# File Types

In computing, **files** are essential for storing data, whether it's a document, image, video, or program. Files can broadly be classified into two categories: **binary files** and **text files**. Understanding these file types, how they store information, and how operating systems recognize them is fundamental to working with computers.

A **text file** contains **human-readable** characters **organized into lines.** The content of a text file typically consists of alphanumeric characters, punctuation, and whitespace (horizontal spaces between words, tabs, i.e. tabulator spaces, and line breaks, also known as vertical whitespace). All of these are encoded using a character encoding system (like ASCII or UTF-8). These files are easily readable by humans and are usually opened using text editors or word processors.

Examples of text files include:

* Plain text files (.txt)
* Source code files (.html, .py, .java)
* Configuration files (.conf, .ini)

In a text file, data is represented using sequences of characters. These sequences are **encoded** into binary data (0s and 1s) **when stored in a file** on disk, since computers are only able to deal with binary symbols directly. But the encoding is standardized in such a way that programs can interpret, i.e. **decode** them as readable characters and render them in this form for the human using the computer.

Unlike text files, **binary files** are not meant to be read by humans. They store data in a format that is optimized for machine processing, consisting of binary digits (bits). Binary files contain a structured arrangement of bits that can encode any data type such as images, videos, executables, and other non-textual content.

Examples of binary files include:

- Executable files (.exe, .bin)

- Image files (.jpg, .png)

- Video files (.mp4, .avi)

- Program data files (.dat)

Binary files often contain both **raw binary data** and **metadata** (information about the file). The structure of binary files can vary depending on the format, and they are often only interpretable by specific software designed to read that particular file type.

## Character Encoding

As mentioned earlier, text files store characters, but these are represented by binary symbols (0s and 1s) when processed internally by a computer, i.e. when a character is stored in a file, in the computer’s memory (RAM) or when processed by its CPU (processor). A system of character encoding is used to convert characters into these binary codes and vice versa. There are many different encoding systems. All of these that are of interest encode characters in blocks of 8 bits, i.e. 8 binary symbols. The most important difference between character encoding systems is whether they encode a single character in one byte (8-bit encoding) or more than one byte (typically two, i.e. 16-bit). Regardless of whether one or more bytes are used, there is always a **mapping** between sequences of binary symbols and characters, which is often referred to as a **code page,** a **code table** or a **character map,** which specifies **for every possible sequence of bits**, e.g. for all 256 possible combinations of 8 bit values, what character this combination of binary values represents.

In order to understand how character encoding works, let us review 8-bit encoding first.

### ASCII

The earliest widely used character encoding was ASCII (American Standard Code for Information Interchange), developed in the early 1960s. It represents each character using **7 bits, allowing for 27 = 128 unique characters.** These include **English letters**, digits, punctuation, and so-called control characters. The latter were historically used to control the output device but are largely irrelevant today. Each ASCII character is assigned a unique number between 0 and 127.

* Control characters received a decimal value between 0 and 31.
* Printable characters, including letters (both uppercase and lowercase), digits, and punctuation marks were assigned values between 32 and 127.

These numbers are stored in binary form within a byte.

Even though ASCII uses only 7 bits, each character is stored in a **full 8-bit byte**. The most significant bit, i.e. the bit with the highest value (128), which is the 8th bit counting from the right, is usually set to 0 in the case of standard ASCII characters.

To illustrate this using a few examples:

* The ASCII code for the upper-case character 'A' is 65 in decimal. To convert 65 to binary, we get: 1000001 on 7 bits, i.e. 64 + 1. When stored in a byte, the 7 bits are padded with a leading 0 to make a full 8-bit byte: 01000001.
* The ASCII code for 'a' is 97 in decimal. To convert 97 to binary, we get: 1100001 (64 + 32 + 1) on 7 bits. Adding the leading 0, the binary code represented as a byte is 01100001.
* The ASCII code for the space character (' ') is 32 in decimal. In binary, 32 is 0100000 on 7 bits. Thus the 8-bit representation is 00100000.

### Extended ASCII

As computing spread globally, there was a need to represent characters beyond the basic English alphabet. This led to extended ASCII, which also the extra 8th bit in a byte to represent an **additional 128 characters** (values from 128 to 255) while keeping the original 7-bit ASCII values the same (0-127).This extended set could include characters for accented letters, symbols, and other special characters used in different languages.

Different regions and platforms developed their own versions of extended ASCII, using the extra 128 code points for various language-specific characters. This led to compatibility issues between systems, as the same byte value (128-255) could represent different characters in different encoding systems, making it hard to share text files between systems. For example, the character 'ä' (the letter 'a' with a German umlaut) would have different codes in the ISO-8859-1 (Western European) and Windows-1252 character sets (Microsoft's version). 8-bit encodings worked well for a single language or alphabet, but failed to handle documents with multiple languages or scripts. This created challenges in regions using non-Latin scripts or where multiple languages needed to be supported simultaneously within the same document.

Some of the most important 8-bit code tables include:

* The ISO-8859 family of character encoding standards: The ISO-8859 series includes several 8-bit character encodings designed to cover different regional character sets. Each variant reserves the first 128 code points for standard ASCII and uses the remaining 128 for region-specific characters.
  + ISO-8859-1 (Latin-1): Supports Western European languages (e.g., French, German, Spanish). It includes characters like é, ü, and ñ.
  + ISO-8859-2 (Latin-2): Supports Central and Eastern European languages (e.g. **Hungarian**, Polish, Czech).
  + ISO-8859-5: Designed for Cyrillic alphabet languages (e.g., Russian).
  + ISO-8859-6: Used for Arabic script.
  + ISO-8859-7: Used for Greek script.
* Windows-1252, also known as CP-1252, was developed by Microsoft. It is based on ISO-8859-1 but includes additional characters in the range 128–159 (like smart quotes, the euro symbol (€), and other typographic symbols. that are not part of ISO-8859-1. Windows-1252 was the default encoding for Western European languages used in many Microsoft products (e.g., Windows and Office).
* Windows-1250, also known as CP-1250, is another Microsoft character encoding used to represent text in Central European languages that use Latin script, including Hungarian. It is similar to ISO-8859-2 and includes all the printable characters of the latter, but some are rearranged.
* These two Windows code pages are often referred to as “ANSI” code page, which is a loosely used technical term that in different locale versions of Microsoft Windows these code pages were used as the default “ANSI” encoding (although they do not strictly conform to the American ANSI standard.
* Even older than the Windows code pages, 437 was the character encoding supported by the original IBM PC hardware which ran the DOS operating system. It included all printable ASCII characters, but replaced most of the control characters by decorative symbols (smileys, card suit symbols, arrows, triangles, etc.). Later, MS DOS used code pages 850 (DOS Latin-1, Western European) and 852 (DOS Latin-2, Central European) depending on country and keyboard setting on PCs for languages using a Latin script. Apart from the letters of the English alphabet, these code pages were very different from ISO-8859 and the Windows code pages.
* MacOS Roman was developed by Apple and used on Macintosh computers to encode Western European characters. Like Windows-1252, it was an extended version of ASCII but not compatible with other systems.

Today all 8-bit character encoding systems are considered obsolete. They are no longer the default encodings in current Windows systems like 10 and 11. UTF-8 has become the default encoding due to its flexibility and support for a wide range of languages. Legacy 8-bit encodings like Windows-1252 and 1250 are still supported, but primarily for backward compatibility with older applications or documents.

8-bit character encodings were an important step in expanding the capability of computers to represent characters beyond the basic English alphabet, but they introduced limitations, particularly in representing multiple languages and ensuring compatibility across systems. Some common 8-bit encodings like ISO-8859-1 and Windows-1252 became widely used, but they couldn't address the growing complexity of global communication. This led to the development and eventual dominance of Unicode, which provides a more comprehensive and flexible solution for encoding text.

### Difficulties Due to the Use of Alternative Encodings

When opening a text file, choosing the correct character encoding (code table) is crucial because it determines how the sequence of bytes in the file will be interpreted as readable characters. If the wrong encoding is used, the file may display as garbled or unreadable text. The process of selecting the correct code table is complicated for several reasons.

Most plain text files (with the notable exception of XML) do not include any metadata that specifies which character encoding was used when the file was created. This means that when opening the file later, the system or software has no explicit indication of how to interpret the bytes. As a result, the program must either guess the encoding, the user must specify it manually.

As we mentioned earlier, different languages require different character sets, and no single 8-bit encoding can accommodate all languages. A file created in one language-specific encoding will display incorrectly when opened in another. This is particularly problematic when working in multilingual environments or when files are transferred between users in different regions.

Even within the same geographic region, different systems might use different encodings. In particular, different operating systems and software platforms historically favoured different encodings. For instance, Windows often used Windows-1252 as its default encoding for Western European languages, whereas Linux and Unix systems preferred ISO-8859-1 or UTF-8. A text file created on one platform might be displayed incorrectly on another if the encoding is not recognized or supported.

A file created on Windows with Windows-1252 might show incorrect characters if opened on a Unix-based system assuming ISO-8859-1, as the same byte value might represent different characters in these two encodings, making it difficult to know which one to choose when opening a file. In the most widely used 8-bit encodings, the first 128 characters (0-127) correspond to standard ASCII, which includes basic Latin letters, digits, and punctuation. Since all such encodings agree on these characters, these do not cause problems. However, the characters from 128 to 255 differ between encodings. For example, the decimal byte value 128 represents different symbols in various encodings. In Windows-1252, it corresponds to the "euro" symbol (€), whereas in ISO-8859-1 it is undefined. If the wrong encoding is used, characters like these might turn into nonsense symbols, or text in non-Latin scripts (e.g., Russian, Chinese) could become unreadable.

The emergence and increasingly universal adoption of Unicode (especially UTF-8) as a standard for encoding multilingual text has solved many of the problems of 8-bit encodings. However, many legacy systems and files still use older 8-bit encodings. When opening older files, users must manually ensure the correct encoding is selected, and there can be confusion between Unicode-based encodings (like UTF-8) and older 8-bit ones.

### Unicode

As computing spread globally, ASCII proved insufficient to represent all the characters of different languages. **Unicode** was developed as a **universal character set,** and one of its most common **encodings** is **UTF-8.** UTF-8 is **variable-length,** using **1 to 4 bytes** to represent a character. It is backward compatible with ASCII and can represent characters from most languages, including symbols and emojis. In theory, a file saved in UTF-8 can be opened on different systems without character corruption. In practice, it is not quite as simple, and a number of further conditions need to be met in addition to selecting the right character from the Unicode character set and encoding it using UTF-8. Let us briefly summarise all necessary prerequisites:

UTF-8 is a character encoding that defines how characters are represented in bytes. It can encode the entire Unicode character set, which includes **characters from virtually all languages** and **symbol sets** used around the world (punctuation, currency, mathematical, logical, typographic symbols, lines and arrows, diacritic markings that can be combined with letters, etc.). When a file is saved in UTF-8, the bytes in the file can represent any Unicode character, ensuring that characters are encoded in a standardized way.

However, even if the file is correctly interpreted as UTF-8, the characters may not be displayed correctly if the **font** used by the program does not support the specific characters in the text. No single font contains all of the over 140,000 characters defined in Unicode, so whether or not a character can be properly displayed depends on the font in use and its character coverage. For example, if a text contains a character from the CJK Unified Ideographs block (used in Chinese, Japanese, and Korean), and the font used does not include those characters, the system might display a "missing character" symbol like � (often shown as a square or box: □), even though the underlying byte data is correct. Fonts like Arial, Times New Roman, or Courier New support only a subset of Unicode characters, often covering basic Latin and common characters. Specialized fonts, like Noto or DejaVu, provide broader support but still don’t cover all Unicode characters. Most modern systems use font fallback mechanisms. This means that if a font doesn't support a particular character, the system tries to substitute it with another font that does. If no available font on the system can display the character, you see a character indicating the glyph is missing.

Furthermore, although UTF-8 ensures the characters in a file can be correctly interpreted, this only works if the system or application that is being used supports it. If Unicode is supported, the final display depends on the fonts installed and how the operating system or application manages missing glyphs. Some older or minimal systems lack broad font support, causing issues with displaying certain characters. Programs like text editors, web browsers, and terminal emulators behave differently when dealing with missing glyphs, depending on their handling of fonts and character sets.

A problem that persists to this day is that it is not always self-evident whether a piece of software that we use supports UTF-8 encoding at all and whether it is used by default by the application or must be selected explicitly. For example, the Python programming language opens files for input and output using UTF-8 encoding by default on Linux but not on Windows. On Windows, the 8-bit encoding that corresponds to the system locale is used as default input and output encoding by Python, which is Windows-1250 in Hungary, Windows-1252 in Western Europe. UTF-8 encoding is supported, but must be specified explicitly every time a file is opened for reading or writing. If a Python program was written for Linux, and the programmer did not set UTF-8 encoding explicitly for ‘open’ commands, this program will not work properly on a Windows system if the input or output file is encoded with UTF-8 and contains any characters outside of the basic ASCII character set. In such cases, the program will either open the file incorrectly or crash with a character decoding error.

## Identifying the Type of a File

An operating system (OS) and its programs must recognize the type of a file to know how to process or execute it. There are several methods by which file types are identified. When the user of a computer tries to open a file, the operating system or application typically first checks its file extension and may also look at its header to determine its type. Based on this information, the OS will choose a program that can open or run the file, e.g. a text editor for a text file, an image viewer for an image file, or the OS directly runs it if it is an executable file. If the operating system does not recognize the file type, it may prompt the user to choose a program to open the file or display an error message.

### File Extensions

The most common method of determining a file type is through its file extension. A file extension is a suffix added to the filename, usually following a period (e.g. `.txt`, `.jpg`, `.mp3`). The extension helps the operating system associate the file with the appropriate program or application to open or execute it. Text files typically have extensions like `.txt`, `.html`, `.xml`, `.csv`. Binary files have as many different extensions as there are different binary file formats. Executable binary files often have the extension `.exe` in Windows (but typically not in Linux). The extension `.dll` indicates a binary file that contains compiled program code that is not executed directly but run by a different executable program. Image files are mostly stored in a binary format such as `.png` or `.jpg`. Files that have been compressed into a special binary format can have the extensions `.zip`, `.gz`, `.7z` or `.docx`. The latter, which is the default file format of Microsoft Word, is a zip-compressed archive consisting of several XML files.

However, file extensions are not always a reliable indicator. Extensions can be changed by users, and files can be misnamed. Some extensions like `.3`, `.bak` (for ‘backup’) or `.old` only indicate that the file in question is a renamed older version of a more recent file that replaces it, but does not express directly whether it is a text or a binary file. Furthermore, some file extensions can be ambiguous. The extension `.doc` typically indicates an old binary Microsoft Word document format, but sometimes it is chosen for a plain text document. For reasons like these, operating systems often use additional methods for determining file types in addition to file extensions.

### File Headers

Many common, standard file formats that are typically not application-specific include a **file header**, a small section of the file that contains specific information about the file, including its type. This header often includes a “magic number” — a unique identifier that helps the OS or software determine the type of the file.

For example:

- A PNG image file starts with the bytes `89 50 4E 47`, which is the magic number for PNG files.

- An executable file (.exe) on Windows starts with the bytes `4D 5A`, which identifies it as a DOS/Windows executable.

Even if the file extension is incorrect or missing, the magic number in the file header can be used by the OS or an application to recognize the file type.

The magic numbers in the above examples are represented as hexadecimal values, which is a common way of specifying the value held in a byte using two hex digits. Hexadecimal notation represents the numbers in a base-16 number system. It is very commonly used to represent values in computing. It represents numbers using 16 symbols: the numbers 0-9 represent values from 0 to 9, whereas the letters A-F (either upper or lower case, with no difference in meaning) represent values from 10 to 15. In addition to the digits of a hexadecimal number, the prefix 0x is often added, which is a convention used to indicate that the number is in hexadecimal format, e.g. 0x4e. This value, which is equivalent to the hexadecimal number 4E, is equivalent to the decimal number 4 \* 16 + 14 = 78. A byte is 8 bits long, and each hexadecimal digit corresponds to 4 bits, since 24 = 16. Therefore, two hexadecimal digits, which can represent 16 \* 16 = 256 different values, are equivalent to exactly one byte (8 bits).

A different type of header that is not a sequence of essentially meaningless bytes and is used relatively commonly in Unix and Linux environments is the so-called **shebang line,** also known as hash-bang and by some other names. This is a line put at the beginning of a script file, i.e. a text file containing instructions in some scripting language. This line starts with a number (hash) sign followed by an exclamation mark and the absolute path to the location of an interpreter application on the machine’s file system, e.g. #!/bin/sh or #!/usr/bin/env python3. The shebang line functions as an interpreter directive, which means that it tells the operating system what interpreter to run the script with (passing the script’s path as an argument to the interpreter application) when the script is run as an executable file from the system’s command line or file manager.

### Metadata

Some operating systems use metadata, or additional information stored about the file, to identify its type. For instance, macOS uses a file's **type and creator code** stored in the file system metadata, which helps associate a file with its application regardless of the file extension.

### MIME Types

**MIME** (Multipurpose Internet Mail Extensions) types are a way to describe the content type of a file or data being transferred, typically used in web browsers and servers. A MIME type consists of a type and a subtype, such as `text/html` for HTML files or `image/jpeg` for JPEG images. When downloading files from the web or handling email attachments, the MIME type helps the system know how to handle the file.

# Processing Text Files

Working with plain text files plays an important role in several situations when working with a Linux system and on bioinformatics data. The main reason for this is that generating and storing data of various kinds in plain text format has numerous advantages.

## Advantages of Working with Text Files

### Readability

Most importantly, plain text files are easily readable by both humans and computers. Whether it's code, configuration files, or documentation, plain text is transparent and accessible without the need for special software. For example, a plain text file’s content might look like this:

Meeting Notes - 2024-09-18

--------------------------------

1. Discuss project timeline.

2. Review budget for Q4.

3. Assign tasks to team members.

4. Finalize proposal draft.

Action Items:

- John to update timeline by 09/20.

- Maria to check budget and provide feedback by 09/21.

- Entire team to finalize proposal by 09/25.

A text file like this can be straightforwardly opened in any basic text editor (e.g., Notepad, Vim, or Nano) and the user can read its content without the need for special software. Plain text files are simple to work with (to create, view and edit), lightweight, and universally supported across all operating systems, thus portable. They are not bound to proprietary formats, making them extremely flexible for storing configuration, log data, code, scripts, or documentation.

By contrast, proprietary binary file formats like the legacy Microsoft Word .doc format are only readable with special software, most typically the software that was used to create it. When such files are opened in a text editor, the editor tries to interpret the file as plain text, displaying the raw binary content. Since most of the data is not encoded as human-readable characters, you will see garbled symbols, random characters, and ASCII control characters similar to this:

Normal Times New Roman ... ÿÿÿÿ ... Word.Document.8 ... Š! ... CompObj ... ÐÏà¡±á

The strange symbols that appear in this view of such a file represent binary data that Word uses to store metadata, formatting, and control information (such as styles, embedded images, or tables) and more generally the structure of the document. You may recognize parts of font names (“Times New Roman”), styles (“Normal”) or file structure elements (like “Word.Document.8”), but these are only small pieces of the whole document, interspersed with unreadable control codes.

You might also come across small fragments of the document’s actual text content embedded within the binary data. Since the text is part of the .doc file, it’s stored within this structure, but it’s interspersed with binary formatting instructions. For example, if your .doc file contained the sentence "Meeting at 10:00 AM," you might see parts of it like this:

ÿÿÿÿ ÐÏà¡±á ... M@€eting a$$t 1Ã0:00 AM@@€€

From a practical processing perspective, the problem with such binary data files is not just that humans cannot read their content, but also that it is difficult to process them using any other piece of software apart from the one that was used to create it (like Microsoft Word or a compatible word processing software).

Although the source code of computer programs is typically stored in plain text files, problems similar to the one described above might arise in connection with them as well. Scripts written in languages like Perl or Python are saved as plain text files with extensions like .pl (Perl) or .py (Python). These files are both human-readable and machine-readable, i.e. they can be opened and edited using any text editor. The following is a typical example:

# Python script to calculate square of a number

def square(num):

return num \*\* 2

number = 5

result = square(number)

print(f"The square of {number} is {result}")

There are, however, computer programs that are not stored in a straightforwardly readable way. Macros in Microsoft Word represent a typical example. A Word Macro is a piece of code embedded within a Word document, typically written in VBA (Visual Basic for Applications). Macros are often hidden from regular view, and accessing the code requires opening a specific developer mode in Microsoft Word. Viewed within this developer mode, a macro has a source code that consists of essentially text not very different from the Python script above, just written in a different programming language:

Sub CalculateSquare()

Dim num As Integer

Dim result As Integer

num = InputBox("Enter a number:")

result = num \* num

MsgBox "The square of " & num & " is " & result

End Sub

However, this program is always embedded in a Word document, making it inaccessible unless opened in Word with macro viewing enabled. The user cannot simply open the macro in a basic text editor, making it less portable and harder to work with across systems. It is not even completely straightforward to transfer a macro from one Word document to another.

To summarise, plain text files (like .txt or scripts) are easily readable by humans and computer programs without needing additional software. They present data and code in a straightforward manner. Proprietary formats like .doc files or Word macros require special software, and opening them in non-compatible tools leads to garbled output or is not directly possible at all (as in the case of the Word macros).

### Compatibility

Text files are compatible across different systems and programs. They are universally accepted by virtually all current generally used operating systems, programming languages, and text editors. This makes them ideal for tasks such as sharing scripts, log files, or configuration settings between machines, even when different software or platforms are used. This property of their being compatible with several different software of hardware environments is often also referred to as portability or platform independence.

This general compatibility of plain text format contrasts with the lack of ease of use of proprietary formats, which generally require specific applications (e.g., Word or Excel) to work with, limiting portability across different systems, especially in environments where such software is unavailable or incompatible. In particular, software that relies on proprietary binary formats is often only available for different versions of Windows. A further obstacle to transferability of proprietary-format files between computers is that the applications that read and process them are often commercial software products that require an expensive licence to be purchased for every machine that they are installed on.

### Ease of Automatic Processing

Text files can easily be manipulated, searched, edited, or parsed directly from the operating system shell. This is especially true of Unix-like systems such as Linux, which provide a wide range of utilities like *cat, grep, sed, awk* and *less* that are designed to work with plain text.

### Version Control

Text files are easy to track and manage with version control systems like Git. This is especially important for developers, as source code is stored as plain text. Changes can be easily compared, merged, and tracked over time.

On the other hand, changes in files that are entirely or partly binary are much harder or sometimes even impossible to track, and even if they have been tracked, the changes are more difficult to inspect and to interpret. For example, while changes in a Python script saved as plain text are straightforward to review using a version control system, changes in a Python script that is saved within a Jupyter Notebook, which is a partly binary file format, cannot be effectively examined.

## Functions of Text Files

Beside storing documents in the everyday sense, i.e. text written in a human language like English, such as email messages, research articles, user guides or recipes, plain text files often contain many different kinds of content.

Automating tasks often involves creating, reading, or writing plain text files. The programs that carry out the automated processes are typically stored in this way, either in the form of a **shell script** (essentially a list of OS shell commands to be executed one after the other) or **source code** written in a programming language like C, R or Perl. Records of the steps of such automated processes, their results and any warnings or errors encountered while executing them are written to **log files**.

Configurations of the operating system’s processes as well as installed applications are stored as plain text files. This is true for the most part of Windows as well, but is applied as a general principle even more systematically by Linux. For Linux system administrators, working with text files is essential for configuring services, managing user permissions, and customizing environments.

Finally, and most importantly from the perspective of bioinformatics, data files that contain e.g. measurements, DNA, RNA or protein sequences, or metadata on these (when, where, by whom and with what method was a measurement taken or a biological sample collected) are most frequently stored as plain text files. Furthermore, all or at least part of the output of bioinformatics software is plain text, which is either written to the standard output stream (the terminal) or to a file.

## Common Tools for Working with Text Files

This course will introduce a variety of tools which can be used to automatically search within, extract and reorganise data from, and to modify text files. To present a general overview of these tools, we review their most important groups in this section.

### Text Editors

Although we will not discuss text editors in this course further, it is worth listing the most important ones that are at the users’ disposal on either Linux or Windows.

Linux:

* **Nano** is a simple editor that works from the command line and allows simple editing of text files.
* **Emacs** and **vim** (as well as its simpler cousin **vi**) are highly customizable and extensible text editor with advanced editing capabilities, but a steep learning curve. These editors are also used from the command line and are entirely controlled from the keyboard using cryptic sequences of key presses. This is difficult to get used, especially for users who have only worked with editors and word processors featuring icon- and menu-based graphical user interfaces earlier.
* There are several Window-based editors with a graphical user interface (GUI) that are installed by default in various Linux distributions. Some commonly used ones are **gedit, jedit** and **kate**. All of these are extensible to different degrees using plugins and offer feature used for programming like syntax highlighting. Gedit is the most simple and minimalist of these, while jedit focuses on extensibility and kate on developer-focused features.

Windows:

* There is no command-line-based editor in Windows that is part of the operating system like nano or emacs.
* **Notepad** is a very basic Window-based text editor for plain text files.
* **Notepad++** is feature-rich editor that is extensible, offers syntax highlighting for many programming and markup languages, and has excellent support for different character encodings, search and replacement using regular expressions, etc. It has a graphical user interface with menu-based commands.

Both systems:

Several **integrated development environments (IDEs)** are available for both operating systems. These are primarily text editors and can edit any plain text file, but additionally offer features that support programming in various programming languages including syntax highlighting, commenting out lines, code folding, code completion, and access to the language’s documentation. Some are specific to **one or a few** **particular languages**, such as **RStudio** (which is specifically designed for the R language), **NetBeans** and **Eclipse** (these are primarily used for Java development, but also supports C, C++ and PHP). Others are **language-independent** and can support multiple languages through plugins. The most important multilingual IDE at present is **Visual Studio Code,** also known as VS Code.

It should be stressed that text editors in the above sense need to be distinguished from word processors. Word processors and text editors are both tools used to create and edit text, but they serve different purposes, target different users, and offer distinct features.

**Word processors** like Microsoft Word (for Windows and Mac), LibreOffice Writer (for Linux) and Google Docs (browser-based, works with any operating system) are designed for creating and editing documents that require complex formatting and layout. They are primarily used for producing polished documents like reports, essays, letters, and other professional or academic work written in a human (natural) language. Word processors support rich text formatting such as bold, italics, underlining, font selection, color changes, and other style modifications. They allow for defining page layouts, including control over margins, alignment, headers, footers, and page numbering. They offer the ability to insert images, tables, charts, and other non-text elements. To support working in a human language, they have a built-in spell checker, grammar checker and thesaurus. To help in producing complex documents, they include tools for creating footnotes, references, and table of contents. They typically save files in proprietary formats such as .docx (Microsoft Word) or .odt (OpenOffice Writer), which **retain both the text and the formatting**. This course does not deal with word processors and text formatting.

Text editors, by contrast, are simpler tools designed to work with plain text files, focusing on text content without much or any formatting. They are primarily used by programmers, system administrators, and users working with configuration files, source code, or documents that do not need complex styling.

### Operating System Utilities

The **Unix and Linux** operating systems are bundled with a set of utilities that are designed to support the processing of text files. These are especially relevant for handling system files such as configuration and log files as well as structured data files, and are generally less useful for natural-language documents. In other words, they are not meant to replace word processors.

Unix-like systems (which include Linux, BSD, macOS, etc.) were designed with the philosophy of providing small, modular, command-line tools that do one thing well and can be combined in powerful ways. From its inception in the 1970s, Unix was designed to be a text-oriented, command-line-driven system, where files were often treated as plain text and manipulated with tools like grep, awk, sed, and cat. Text files and text streams became a central concept of the operating system, allowing users to easily work with and process data from the command line.

Because of this, tools like grep, awk, sed, cut, sort, uniq and tr are core components of Unix-like system, built to work together via piping (|) and redirection to process streams of text data. There are tools for:

* **viewing** the contents of text files, especially cat, less and more;
* **finding** files containing particular string patterns: grep in combination with find;
* **manipulating** text, such as sed for replacing words or other patterns, tr for replacing characters, cut, sort and uniq to work with tabular data in plain text format;
* **extracting specific lines** from long text files: head and tail;
* **comparing** files and displaying their differences: diff;
* **counting** lines, words and characters in a file: wc.

Such text processing utilities are missing from the **Windows** operating system, which is due to differences in the historical design goals and philosophies of the two operating systems.

Microsoft’s DOS (Disk Operating System), and later Windows, had a different target audience and was designed to be a simpler system for single-user personal computers. Windows started out as a graphical user interface (GUI) including a window manager for DOS in the 1980s and gradually evolved to become an independent operating system. It was designed to move users away from the command line and encourage interaction through graphical applications (like Microsoft Word and Excel) instead of text-based command-line tools. These applications became the primary ways users created and edited content. The expectation was that users would work with formatted text in word processors, not plain text at the command line.

The DOS operating system had very limited command-line tools for working with text, mainly just type (which is DOS’s equivalent of cat) and more for displaying the content of documents, and very basic text editors (edlin in early MS-DOS versions, complemented by edit later). Similarly the language to write shell scripts, which are called **batch files** in DOS/Windows, is very limited compared to the possibilities of shell scripting in Linux, which comes close to a full-featured programming language. The Windows command-line interface CMD has retained these basic tools but has not expanded them with further utilities. Users are expected to rely on GUI applications for working with text files or can write scripts in programming languages such as Perl or Python to achieve such goals.

With the introduction of PowerShell in 2006, Microsoft took steps to improve the command-line environment in Windows, but PowerShell's model was based on objects rather than plain text. PowerShell commands return structured data objects, not text, which can be more powerful in some scenarios but also less suited for traditional text-based workflows. PowerShell is far more powerful than the DOS-based command-line interface and batch scripting for managing system configurations and automating administrative tasks, and is based on a full-featured object-oriented programming language that is at least as expressive as that of Unix shell scripts, but is not geared toward text manipulation.

For these reasons, Unix and its derivatives like Linux are the operating systems of choice for professionals who need to work with data stored in text files, including bioinformaticians and developers working with many programming languages. Text processing utilities in Linux will be the focus of chapter 5 of this textbook.

### Regular expressions

Regular expressions, commonly abbreviated as regex or regexp, are patterns used to search, match, and manipulate text. They are available in many scripting languages and tools, making them indispensable for working with plain text.

A regex defines a pattern that can match sequences of characters in text. These patterns can be simple (like matching a word) or complex (like matching an email address or extracting parts of a log file).

A regex pattern is like a blueprint or recipe that describes a way to match specific parts of text. It is essentially a set of instructions that tell a program exactly what kind of text you are looking for—whether it's a word, a number, or a more complex structure like an email address. A regex pattern can consist of several different types of component:

**Literal characters:** These straightforwardly specify the **exact character** that the user is looking for. For example, if they are looking for the word "cat", the corresponding regular expression pattern is just cat. This would match any place in the text where the word "cat" appears.

**Character classes:** These match a specific **set of characters** and are mostly specified using square brackets. For example, the pattern [aeiou] matches any English vowel, and the more complex pattern c[aeiou]t would match *cat, cot, cut,* etc. It tells the program, “I'm looking for a *c*, followed by either *a, e, i, o* or or *u*, and finally a *t*.”

The **wildcard** character: A . (dot) in regex stands for any character. So the pattern c.t would match *cat, cot, cut,* or *c9t*—anything that fits the "c + any character + t" pattern.

**Alternatives:** Regex patterns allow you to **group** sequences of characters, including character classes and wildcards, and specify that a string may contain a longer string that has a substring which matches either one or the other subpattern. For example, the pattern qui(te|et) matches either *quite* or *quiet.*

**Repetition and optionality:** Regex patterns can match individual characters or groups (sequences) of characters that repeat or are optional.

When used in combination, such subpatterns allow users to define very complex patterns that can match any formally identifiable part of a text document. Such matches can be used for **searching** in text files, i.e. extracting the contexts in which the matching string appears, most commonly using the grep utility in Unix or Linux. Alternatively, they can be used for **replacing** parts of text, such as removing unnecessary whitespace, e.g. extra spaces at the end of a line. The most commonly used tool for this is the sed utility. Regular expressions are crucial for **extracting data** from unstructured text, such as log files, CSV files, or email addresses from a body of text. They are also indispensable in many natural language processing workflows.

Regular expressions are especially powerful when used within computer programs, which make it possible to manipulate the matching parts of a document in a much more flexible manner than relatively simple utilities like grep and sed*.* Most programming languages support regular expressions, but the language most clearly designed for working with them is Perl. Chapter 3 will deal with regular expressions in much more detail.

### Programming Languages

Several programming languages, most importantly Perl, Python and Awk, are widely used for automating tasks and processing text files. Since these languages are particularly suited for writing short, simple scripts that automate small tasks, they are often referred to as scripting languages. They come with powerful features that make handling plain text and pattern matching straightforward, especially compared to languages like C or Java, which have a different focus.

**Awk** is a specialized language designed specifically for text processing in Unix/Linux environments. It is lightweight and excels in tasks like pattern scanning, report generation, and simple transformations of structured text (like CSV files or log files). Awk reads a file line-by-line, applying specified patterns or commands to each line. This makes it ideal for analyzing logs or tabular data. It automatically divides each line into fields, based on a delimiter (by default, spaces or tabs), making it particularly useful for working with CSV or tab-delimited files. The awk language will be the topic of chapter 5.

**Perl’s** native support for regular expressions allows for complex pattern matching and substitutions. It excels at handling large amounts of text efficiently, making it ideal for tasks like log file parsing, data extraction and text filtering. In the 1990s and 2000s it was accordingly the most important language for writing bioinformatics and natural language processing applications. While it has been mostly replaced by Python in these areas more recently, it is still very useful for working with regexes and for writing scripts for system administration. Perl will be the topic of chapter 6 of this textbook.

**Python** is the most popular programming language today, especially in software development outside of enterprise contexts (where other languages such as Java and C++ dominate), especially among hobby programmers and in scientific computing, including bioinformatics and natural language processing, as well as in artificial intelligence. It is very similar to Perl in many respects and differs from it mainly in its syntax. Although Python has very good support for working with string data, like handling character encoding and regular expressions, and has extensive libraries for dealing with text files of different formats including CSV, JSON and XML, it is not primarily designed for text processing, and will not be dealt with in this course systematically.

## Dealing with Natural-Language Text

### Natural Versus Formal Languages

Processing normal documents written in a natural language presents specific challenges. The notion of natural language refers to languages spoken, read and written by humans, such as English, German or Hungarian, and contrasts with formal languages, such as the languages of formal logic like propositional or first-order (predicate) logic, as well as programming languages. The main difference between natural and formal languages from a processing perspective is that the latter have a precisely defined syntax, i.e. a set of rules to build more complex sentence-like expressions out of simpler, word-like expressions (tokens), and a precisely defined semantics, which determines exactly how a complex expression built up in a particular way out of basic parts is interpreted. By contrast, while complex expressions in natural languages like sentences are similarly built up out of simpler parts according to certain syntactic regularities, these regularities are far more complex and chaotic and less rigidly defined than the rules of formal languages. Furthermore, while natural language semantics also determines how the meaning of complex expressions like phrases or sentences are built up out of the meanings of their parts, the meanings of both the parts and the complex structures are generally both vague, i.e. unclear rather than well-defined, and ambiguous, i.e. the same expression can mean different things in different contexts, or sometimes even in the same context. For these reasons, parsing and interpreting natural language text is a very hard problem especially using an algorithmic approach.

### Structured Versus Unstructured Data

Documents written in a natural language are also difficult to process automatically because the document as a whole is not structured in a systematic and rigid way, i.e. it represents unstructured data.

**Structured data** refers to the sort of data in the context of data processing that is highly organized and formatted in a clear, predefined manner. This usually means that it is stored in tables or databases, where each data point is represented as a record (table row) divided into a specific number of clearly defined fields (table columns), each of which holds data of a specific type to be interpreted in a specific way (e.g., numerical data, dates, addresses). Typical examples of structured data include spreadsheets, forms and relational databases. Structured data are easily processed by algorithms because of their consistent format. This format allows a program to retrieve the required information (e.g. the address of a person with a specified given name, family name and birth date) in a deterministic and unambiguous way via a simple sequence of processing steps. The difficulty in handling structured data lies in creating them, i.e. in organising the unstructured information in the real world so that they are finally represented in such a clean structure. When creating a structured data set, care must be taken to avoid anomalies such as duplicate or incomplete entries, entering data in the wrong field or in the wrong format, inconsistencies such as sometimes entering month numbers as Roman and sometimes as Arab numerals, etc. Handling structured data and retrieval from such data sources is very sensitive to anomalies like these.

**Unstructured data** are not organized in a predefined way and lack a specific structure. For example, while a textbook like this is divided into chapters and sections and thus is not entirely devoid of structuring, it is not possible to specify a sequence of simple steps to retrieve a specific piece of information from this text, such as what the -E option of the Linux grep utility is for. Moreover, it is perfectly normal for a natural language text to pick up the same topic or mention the same person several times in different contexts throughout a longer document, and this is not considered an anomaly, as opposed to a similar situation in a structured database.

Typical types of unstructured data include documents containing natural language text such as emails, images, videos and mixed-media content, e.g. research papers containing both text and images (e.g. diagrams or photographs) or social media posts consisting of static and / or moving pictures, spoken and / or written language possibly accompanied by sounds or music. Such data are difficult to search and process automatically, and retrieving specific pieces of information from a large amount of these data typically requires advanced tools. For example, although full-text search using regular expression pattern matching is a possible way to find information in unstructured text, this approach does not always lead to success due to factors such as synonymy, i.e. the fact that the same concept can be expressed with many different words and complex expressions in natural languages, and the flexibility of syntactic structures, i.e. the fact that the search terms of interest to us can appear in different orders and distances from each other. For example, there is no straightforward way to divide the problem of finding the frequency of a fatal outcome of an e. coli infection among healthy adult patients in a set of research papers into a sequence of well-defined steps so that this series of steps yields exactly the contexts in these papers that answer this question. There might not even be a single correct or best correct answer to a problem, such as searching for the best-fitting image in a large collection of images based on another image (as in an image search service such as Google Lens) or on a natural-language search terms (e.g. searching in the Getty Images image and video database).

Solutions to such problems are necessarily of a **heuristic** nature, i.e. they provide answers that are not guaranteed to be correct, or to be the best possible answer to the question at hand, but rather which are useful and “good enough” most of the time. They generally rely on natural language processing and machine learning techniques.

### Text Mining

Text mining is the process of extracting useful information and patterns from unstructured text data. It applies techniques from data mining, natural language processing (NLP), and machine learning to understand and analyse large amounts of text. More specifically, text mining can be defined as a specialized form of data mining that focuses on large volumes of unstructured text. While **data mining** traditionally deals with structured data (like numbers and categorical variables in databases), text mining adapts and extends these techniques to handle the more complex, unstructured nature of textual information. Both text mining and data mining aim to uncover useful patterns, trends, and relationships within **large amounts of data** that would be impossible or at least extremely costly in terms of time and money to process and make sense of using human labour. Their general goal is to **support decision-making, predict outcomes or gain insights** grounded in these data.

Data mining generally involves a **structured workflow** consisting of steps that successively build on the results of the preceding steps. First, data needs to be **preprocessed**, then **features** need to be extracted from raw data. Based on these features, the data are **analysed** using various **machine learning techniques** that serve to **group similar data** quickly and automatically into clusters (clustering), assign them to **predefined classes** (classification), determine **relationships** between them (association rule mining), etc.

In the course of text mining, such steps are applied in ways that are relevant for unstructured natural language texts. Text data is first **cleaned and prepared** for analysis, which involves tasks like tokenization (splitting text into words), stopword removal (removing common words like "the" or "and") and stemming/lemmatization (reducing words to their base or dictionary form). The preprocessed text then needs to be **converted into** **numerical features** that machine learning algorithms can process, such as bag of words, TF-IDF (Term Frequency-Inverse Document Frequency) to identify keywords that are characteristic of certain documents as opposed to others, or low-rank continuous vector space representations commonly known as word embeddings (like Word2Vec) to capture the meanings of words. Once the documents have been transformed into numeric feature vectors, the subsequent **analysis** can involve:

* grouping similar text documents together, e.g. clustering news articles by topic;
* categorizing text into predefined classes, e.g. classifying emails as spam or not;
* categorizing individual words into predefined classes, e.g. identifying entities mentioned in the text like names of drugs, illnesses, pathogens, etc.;
* identifying relationships within a sentence or longer section of text, e.g. what drug was used with what dosage to treat what illness appearing in what organ with what outcome;

and many more.

In the biomedical domain, text mining plays a crucial role in extracting valuable insights from the vast amount of unstructured text data available in research articles, clinical records, patents, and other medical documents. For example, text mining can help verify whether certain relationships between genes, proteins, diseases and drugs have been previously discovered by analysing large volumes of research papers. It can help retrieve known associations between certain genes and specific diseases, or the effect of drugs on certain conditions. It might be applied to biomedical literature, clinical reports or social media data to identify adverse drug reactions in real-world settings. By extracting relevant information from large repositories of clinical notes, electronic health records and medical publications, it can identify patterns that can be used to support personalized treatment plans tailored to individual patients. It can be applied to social media, news reports and public health databases to monitor disease outbreaks or emerging health trends as well as to track the spread of diseases.

# GNU/Linux Tools for Text File Processing

As summarised briefly in section 2.3.2, the Unix and Linux operating systems (OSs) have a range of useful tools that are specifically designed to **facilitate working with text files**. The main reason for this is that both configuration and log data, which are essential for the functioning of both the OS itself and applications installed on it, are stored in the form of text files within these OSs (as opposed to systems like Windows which rely on binary files for the same purposes).

However, the ability to view, search, and manipulate text files is not only crucial for system administrators, but also for developers and scientists working with data, including bioinformaticians. Many tasks in all of these fields involve working with various kinds of data stored in text format. GNU/Linux offers a rich set of **command-line (CL) tools** to perform these tasks, enabling users to work with text files quickly and effectively.

## Why Use Command-Line Tools?

For users accustomed to graphical user interfaces (GUIs), transitioning to command-line interfaces (CLI) for text file manipulation can be a daunting experience. The main challenge lies in the stark difference in the way the user interacts with the computer. GUIs offer intuitive visual elements — such as buttons, menus, and windows — that allow users to perform actions through point-and-click operations. By contrast, CLIs require users to learn and remember specific commands, their syntax, and various options. This reliance on **memorization and typing** can be intimidating especially for those who are not familiar with programming or technical environments.

However, despite this hurdle that needs to be overcome to start using CL tools for text processing, it is unfortunately unavoidable to work with them. This is because although GUIs provide an accessible way for users to interact with software, they also come with several **drawbacks,** particularly when it comes to processing **large amounts** of data or performing **repetitive tasks**

GUIs often restrict users to **predefined actions** that arefrequently used by most users, making it challenging to perform certain operations that go beyond things like copying, pasting and deleting. Many features available in CLI tools are missing entirely from graphical interfaces or require special software to be installed, such as mass-renaming files, or searching within a specific set of files.

GUI applications can be **slower and less efficient**, especially when dealing with **large files or datasets**, or with a relatively **large number** (e.g. tens of thousands) **of files**. GUI-based text editors in general are not optimized for handling large files. When opening such files, these editors consume substantial amounts of RAM and CPU. Most editors simply crash even when opening text files of moderate size (a few megabytes). Their features which are very convenient when working with small files, such as rendering the text in a scrollable window, highlighting search results, automatic and real-time spell checking, etc., lead to sluggish performance and frequent crashes even if a file is successfully opened. The time taken just to update the display of a moderately large file can lead to delays in scrolling or typing. The responsiveness of the editor diminishes as the size of the file grows, affecting overall productivity and resulting in a frustrating user experience. Features such as search and replace are also optimised toward small files and become cumbersome and slow when applied to large documents.

In fields such as bioinformatics and other scientific disciplines, the advantages of using command-line tools become particularly pronounced.

Bioinformatics often involves processing **massive datasets**, such as genomic sequences or protein structures. CLI tools can efficiently handle these large files, whereas GUI applications face the difficulties that have been outlined above.

Scientists frequently need to apply the **same analysis across multiple files** or datasets. CLI tools can be easily scripted to perform **batch processing**, saving time and minimizing errors. In contrast, GUIs would require manual intervention for each file, leading to **inefficiency** and potential **inconsistencies**.

CLI tools can be seamlessly **integrated into data pipelines**, allowing researchers to combine multiple tools in a single workflow. This flexibility is often difficult to achieve with GUI-based programs, which may not easily interoperate or may require cumbersome data export/import steps.

Using CLI commands can **enhance reproducibility** in scientific research. Researchers can document their analyses in scripts, ensuring that methods are **transparent** and can be **easily repeated** by others. This level of documentation is often harder to achieve with GUI operations, which may not record the steps taken.

Finally, most bioinformatics tools and libraries are **designed primarily for command-line use**. Relying on GUIs would generally limit researchers to a narrow range of tools, potentially hindering their research capabilities.

## What is GNU/Linux?

The fact that Linux CL tools have the same name and work in largely the same way as the corresponding tools of the Unix operating system is rooted in the history of Unix and its influence on Linux development.

The development of Unix began at the end of the 1960s, and its first version was released in the early 1970s. It was designed as a multi-tasking, multi-user operating system featuring a powerful CLI. Unix adopted a design philosophy that emphasized simplicity, modularity, and the idea that each tool should do one thing well. This led to the development of many small utilities that could be combined in scripts and pipelines, creating powerful workflows.

Over time, many different variants of Unix were created by different companies. These OSs shared the same architectural philosophy and similar CL tools, but generally had different kernels. The proliferation of Unix variants eventually led to the creation of standards, like the POSIX standard published in the late 1980s, to ensure compatibility of software across different Unix system variants and a consistent user experience.

Linux was created in the early 1990 to serve as a free and open-source alternative to Unix. Users familiar with Unix should be able to switch to this new system effortlessly. At the time of Linux’s creation, there had been a long-running project called GNU (“GNU's Not Unix”), started in the early 1980, which aimed to implement free and open-source alternatives of the tools and utilities of the Unix OS, including compilers (like GCC, the GNU C compiler, or gawk, the GNU awk compiler), text editors (like GNU Emacs) and shells (like bash). Many such components had been finished in the 1980s and early 1990s, but the originally planned kernel component of GNU faced significant delays, so it was a logical step to merge the two projects by combing the Linux kernel and the GNU tools, resulting in GNU/Linux. Although the “GNU” part of the name is often left out, and the OS is simply referred to as Linux, current distributions of the OS still contain the GNU utilities as their crucial components.

As GNU tools were designed to be compatible with Unix, the Linux command-line environment mirrored that of traditional Unix systems. At the same time, the open-source nature of Linux encouraged contributions from a vast community, leading to enhancements and additional tools that continued to align with Unix philosophies. As a consequence, while the core functionality of the GNU tools is identical to that of their Unix counterparts, they tend to offer useful options that extend the capabilities of the latter.

## Viewing and combining the Content of Text Files

In this subsection, we will explore a range of command-line tools that are used for viewing and searching the contents of text files. These tools are indispensable for navigating files in a Linux environment, allowing users to read and analyze information quickly.

### less and more

These two commands let a user read the content of a text file on the terminal:

less doc.txt

views large files one screen at a time, the arrow keys serve to scroll up and down through the text. the q key exits less.

The more command works similarly, but does not allow the user to scroll up. The space key displays the next page (i.e. screen) from the file, the enter key scrolls down one line at a time.

### cat and paste

The cat command is mainly used to display the contents of a file on the terminal. For short text files that do not take up several screens, cat is a quicker choice to view a file than less or more.

Its name is short for “concatenate”, since it concatenates the files that are listed as its arguments.

cat doc.txt

prints the content of the file doc.txt to the terminal.

doc1.txt:

First sentence.

doc2.txt:

Second sentence.

cat doc1.txt doc2.txt

concatenates the two files, i.e. combines the content of two text documents into a single string, and prints:

First sentence.

Second sentence.

Printing concatenated files to the terminal is usually not very useful. However, instead of printing to the terminal Linux’s **redirection operator** > allows the concatenated string to be written to a file like this:

cat doc1.txt doc2.txt > combined\_doc.txt

This creates a new file combined\_doc.txt or overwrites that file if it already exists, and prints the concatenated content of the two files specified as the arguments of cat. Any number of files can be specified as arguments of cat, and wildcard characters \* and ? can be also used, e.g.

cat doc?.txt

An alternative to the redirection operator is the append operator >>, which also redirects the output of a command to a file, but does not replace an existing file but rather appends the output of the command to an existing file. Appending means concatenating a string to the end of a file.

Thus the following sequence of commands:

cat doc1.txt > combined\_doc.txt

cat doc2.txt >> combined\_doc.txt

is equivalent to

cat doc1.txt doc2.txt > combined\_doc.txt

Knowing line numbers can be useful, especially in combination with the grep tool. The -n option adds the line number for every line as a first column to the output of cat.

The tac command is similar to cat, but outputs input lines in reverse order, i.e. last line to first, per file.

While cat concatenates the content of files „vertically”, the paste command does the same horizontally: it concatenates the specified files line by line, separated by a tab character by default. The first line of the first input file is combined with the first line of the second input file, then the third input file if there is one, and so on, to yield the first line of the output file, and similarly for all other lines. If the input files do not have the same number of lines, the shorter files are padded with empty lines by paste to match the length of the longest file.

paste\_file1.txt

apple

banana

cherry

orange

paste\_file2.txt

22

11

53

8

paste paste\_file1.txt paste\_file2.txt

apple 22

banana 11

cherry 53

orange 8

The -s option results in the opposite behavior: first the lines of the first file are concatenated horizontally on the first output line, then the lines of the second file horizontally on the second output line, etc. Thus the output is the transposed version of the output of the default setting of this command.

paste -s paste\_file1.txt paste\_file2.txt

apple banana cherry orange

22 11 53 8

### head and tail

The head command displays the first 10 lines of a file, which is useful for quickly checking what is in the file and to display just the header of a file, for example of a table. The number of lines to be displayed can be set using the -n option:

head -n1 combined.txt

prints just the first line. A negative number of lines can also be specified for this option, in which case all lines are printed from the beginning of the files but the last n lines, or in other words, everything is printed, but the last n lines are omitted.

More than one file can be specified as arguments of head. In that case the first lines of each file are printed to the terminal.

The tail command shows the last 10 lines of a file, which can be useful to look at the most recent entries in a log file, for example. It works similarly to head, with the difference that for the -n option instead of a negative number a + modifier can be specified like this:

tail -n+2 combined.txt

with the effect that it prints all final lines of the file starting with the given line (counted from the beginning of the file), in this case line number 2.

Thus while head -n1 prints just the first line, tail -n+2 print everything but the first line, e.g. a table without the header line.

In addition to the -n option, a -NUM or +NUM argument can also be passed to these commands, NUM being an integer.

* head -5 is equivalent to head -n5, it outputs the first 5 lines of a file.
* tail -2 is similarly equivalent to tail -n2, it prints the last two lines.
* tail +3 is equivalent to tail -n+3, i.e. it prints all lines of the file starting from line 3, to the end of the file.

+NUM is not defined for head.

### wc

The wc command does not print the content of a file, it is an important tool to get summary information about what the file contains. It counts three important proporties of a text file: line, word and character count. Line count is defined as the number of newline characters, and word count as non-zero-length sequences of characters delimited by whitespace. The number of words (also known as word tokens) in a text is the usual number by which the size of a text or a collection of texts (a text corpus) is measured in computational language processing.

For example, we can use wc to compare whether Melville’s Moby Dick or Joyce’s Ulysses is longer. We download the two novels respectively.

wget https://www.gutenberg.org/cache/epub/2701/pg2701.txt

wget https://www.gutenberg.org/cache/epub/4300/pg4300.txt

wc pg2701.txt

22316 215838 1276290 pg2701.txt

wc pg4300.txt

33216 268086 1586382 pg4300.txt

The output of wc is the line, word and character count in three columns. We see that Ulysses is significantly longer than Moby Dick. If we are only interested in specific counts, the output of wc can be changed to print only characters using -c, lines using -l and words using -w, or any combination of these.

## Table processing commands on text files

Linux provides many commands that are especially suited to the processing of tabular data in text files.

### sort and cut

The sort command sorts the lines in one or more file and outputs the sorted lines. Given the following example file:

fruit.txt

Banana

apple

Cherry

banana

Apple

cherry

grape

Orange

the command sort fruit.txt outputs the lines sorted in so-called lexicographical order. This means that the first line is the one with the first character the ASCII (or Unicode) code of which is smallest, or if the code of the first character of two lines is identical, then the line of these two that has the second character with the lowest code, or if these also match, then the third character, and so on. Recall that upper-case letters come first in the character code tables, thus the output for the example file above is:

Apple

Banana

Cherry

Orange

apple

banana

cherry

grape

The sort command has many options that allow for much more complex uses. The -r option reverses the sort order, i.e. sorts in descending order:

sort -r fruit.txt

grape

cherry

banana

...

The -f option activates case-insensitive sort, i.e. it ignores whether the letters are upper- or lower-case.

sort -f fruit.txt

Apple

apple

Banana

...

The -u option stands for unique, it removes duplicate lines. In this example, it is useful in combination with -f:

sort -fu fruit.txt

apple

Banana

Cherry

grape

Orange

When sorting numbers, lexicographical order is generally not useful. However, sort also provides an option for numerical sorting with -n. Example:

numbers.txt

10

2

30

1

251

sort -n numbers.txt

1

2

10

30

251

Note that lexicographical sorting would yield:

sort numbers.txt

1

10

2

251

30

If a table in a text file contains several columns, the -k (short for key) option can be used to sort by specific column. The format for this option is -k START[,END], where START and END are numbers, START is the first in a range of columns by which the table is sorted, and END is the last column. If END is omitted, sorting is performed from the starting field to the end of the line. If only a single column should be used for sorting, then START and END both need to be specified with the same number like this:

fruit\_data.txt

apple 10 4.5

banana 20 1.2

cherry 15 2.3

apple 25 4.0

sort -k 2,2n fruit\_data.txt sorts this by the second column numerically:

apple 10 4.5

cherry 15 2.3

banana 20 1.2

apple 25 4.0

More than one column can be specified to sort on the first key initially, then break ties using the second (, third, etc.) column. For example, sort -k 1,1 -k 3,3n fruit\_data.txt sorts by the first column lexicographically and then by the third column numerically:

apple 25 4.0

apple 10 4.5

banana 20 1.2

cherry 15 2.3

By default, any sequence of whitespace (space and tab) characters functions as a column delimiter. Alternatively a custom delimiter character can be specified using the -t option. For csv files, this is set to the comma character like this:

fruit\_data.csv

apple,10,3.5

cherry,15,2.3

banana,20,1.2

apple,25,4.0

sort -t, -k 2,2n fruit\_data.csv

apple,10,3.5

cherry,15,2.3

banana,20,1.2

apple,25,4.0

The cut command also mainly serves to manipulate tabular data, although annoyingly the same functionalities such as column selection and delimiter specification are assigned to different options. The cut command extracts specific columns from a text file using the -f option. Either a single column can be specified, or a range of subsequent columns, and only one -f option is allowed. Furthermore, cut assumes a tab-separated input file; a custom delimiter is specified using the -d option.

cut -f1 -d' ' fruit\_data.txt

apple

banana

cherry

apple

cut -f2-3 -d, fruit\_data.csv

10,3.5

15,2.3

20,1.2

25,4.0

The selection can be inverted using the --complement option. In addition to selection of column, a range of characters to be extracted from each line can also be specified. This can be used to crop lines longer than a certain limit, for example.

It is worth noting that Linux also has a tool to shuffle the lines of a file randomly, called shuff. It has various options that

### uniq

This command assumes that the text has already been sorted. It detects repeated (duplicate) lines that occur directly one after the other. It does not detect repeated lines unless they are adjacent. With no options, matching lines are merged to the first occurrence.

uniq\_example.txt

apple

banana

banana

apple

apple

cherry

cherry

cherry

banana

orange

orange

Note that this has not been sorted. The default behavior is:

uniq uniq\_example.txt

apple

banana

apple

cherry

banana

orange

To print only unique lines (which are not repeated), the -u option can be used:

uniq -u uniq\_example.txt

apple

banana

Conversely, the -d option only prints duplicated lines, once for every repeated group:

uniq -d uniq\_example.txt

banana

apple

cherry

orange

The -c option prefixes each line with the number of times it appears consecutively.

uniq -c example.txt

1 apple

2 banana

2 apple

3 cherry

1 banana

2 orange

Furthermore, there are also options to ignore upper vs. lower case, ignore the first *n* columns or characters, or ignore all but the first *n* characters when comparing lines.

### join

The join command is used to combine lines from two files based on a common key. It functions similarly to SQL joins. Both files must be sorted on the join field. By default, join uses the first field in each file as the key for joining. This can be changed using the -1 and -2 options.

sample\_collector.txt

101 Alice

102 Bob

103 Alice

104 Alice

sample\_date.txt

101 2022-08-11

102 2023-09-02

103 2023-10-14

104 2023-11-28

join sample\_collector.txt sample\_date.txt

101 Alice 2022-08-11

102 Bob 2023-09-02

103 Alice 2023-10-14

104 Alice 2023-11-28

By default, the column separator is any sequence of whitespace characters. This can be changed using the -t option. Also by default, all columns from both files are included in the output. This can be changed using the -o option by specifying a comma-separated list of columns from each file to include. For example, -o 1.2,2.2 means that the second columns from the first file and the second column from the second file are included, i.e. the key is omitted in the output:

join -o 1.2,2.2 sample\_collector.txt sample\_date.txt

Alice 2022-08-11

Bob 2023-09-02

Alice 2023-10-14

Alice 2023-11-28

### column

The column command aligns the columns of a table properly using sequences of spaces. By default, it assumes that the columns in the input file are separated by whitespace, but it also has a -s option to specify a different column delimiter. In this way, it can transform a csv file into a visually more readable table (assuming that the columns fit on a screen horizontally):

column fruit\_data.csv -s, -t

apple 10 3.5

cherry 15 2.3

banana 20 1.2

apple 25 4.0

It has numerous options, e.g. it can add column headers using the -N option or transform the input table into JSON format using the -J option. It is also possible to transform the input into

## Comparing two files

Linux has several tools of various complexity that serve to compare two files and determine whether they are identical, and if not, where they differ.

We will use the following example files to illustrate how these commands work:

compare\_text1.txt

Alice in Wonderland

Through the Looking Glass

Jabberwocky

The Red Queen

The White Rabbit

Mad Hatter's Tea Party

The Cheshire Cat

A Garden of Talking Flowers

The Pool of Tears

compare\_text2.txt

Alice in Wonderland

Through the Looking-Glass

Jabberwocky

The Red Queen

The White Rabbit

Mad Hatter's Tea Party

The Cheshire Cat

A Garden of Singing Flowers

The Pool of Tears

A Curious Oyster

### cmp

The simplest such tool is the cmp command. It takes two files as input and by default outputs the line and byte (i.e. character) number of the first difference:

cmp compare\_text1.txt compare\_text2.txt

compare\_text1.txt compare\_text2.txt differ: byte 40, line 2

If there is no difference, its output is empty. Exit code 0 is returned if there is no difference, 1 if there is at least one.

If the number of character-level differences is relatively small, it might also make sense to check its long output with the -l option:

cmp -l compare\_text1.txt compare\_text2.txt

This prints the character positions (from the start of the file) in which the files differ, and the differing character codes in the second and third column for each position:

40 40 55

142 124 123

143 141 151

144 154 156

145 153 147

cmp: EOF on text\_processing/compare\_text1.txt after byte 175

EOF stands for end of file, as always.

### diff

This is probably the most standard CL tool on Linux to compare two files and view their differences. It has an option to ignore case and several options to ignore various types of whitespace (everything, only spaces at the end of lines, blank lines, etc.). Its output looks as follows. The numbers of differing lines are indicated, as well as the type of difference (c for change, a for addition):

2c2

< Through the Looking Glass

---

> Through the Looking-Glass

8c8

< A Garden of Talking Flowers

---

> A Garden of Singing Flowers

9a10

> A Curious Oyster

An alternative output format is the two-column output that is activated by the -y option:

Alice in Wonderland Alice in Wonderland

Through the Looking Glass | Through the Looking-Glass

Jabberwocky Jabberwocky

The Red Queen The Red Queen

The White Rabbit The White Rabbit

Mad Hatter's Tea Party Mad Hatter's Tea Party

The Cheshire Cat The Cheshire Cat

A Garden of Talking Flowers | A Garden of Singing Flowers

The Pool of Tears The Pool of Tears

> A Curious Oyster

### comm

This command finds common and unique lines between two sorted files. By default, it outputs a table with three columns. Column one contains lines unique to the first file, column two contains lines unique to the second file, and column three contains lines common to both files. These columns can be **suppressed** by -1, -2 and -3 respectively. Therefore, for example, -12 only prints the lines common to both files, -23 lines that only appear in the first file, and -3 all differences:

comm -3 compare\_text1\_sorted.txt compare\_text2\_sorted.txt

A Curious Oyster

A Garden of Singing Flowers

A Garden of Talking Flowers

Through the Looking Glass

Through the Looking-Glass

## Using pipes for more complex text processing

The outputs of the text processing commands that we have discussed are generally text files themselves. These text files reside in the computer’s memory and can either be printed to the terminal or saved to a file using output redirection, as we have seen. However, they can also be sent (“piped”) to another text processing tool for further processing using the pipe operator |. All commands that take a single file as argument have a version with zero file name arguments. In this case, the input to the command is expected from standard input. The pipe operator sends the standard output of the command directly to its left to the standard input of the command directly to its right. In this way, several commands can be chained one after the other using a series of commands connected by pipe operators.

Let us assume that we need the total word count of all files in a directory. wc \* produces the following output:

3730 40952 237115 219.txt.utf-8

22316 215838 1276290 pg2701.txt

33216 268086 1586382 pg4300.txt

6782 51257 306595 pg64317.txt

66044 576133 3406382 total

Thus we need the value of the second column from the last line of this output. We can achieve this using the following complex command using three pipe operators:

wc \* | tail -1 | column -t -o, | cut -d, -f2

The column command is necessary here because cut by itself cannot handle the sequences of spaces of various lengths as column delimiters, whereas column parses the table correctly and converts it into a csv structure that cut can deal with.

Another application of piping is to select a specific line from a file. This can be achieved using a combination of head and tail. Often cat is used to start the processing of a text file using a pipeline, although tail could directly take the file name as argument as well. In this case, cat also serves to add the line number to check that we have indeed selected line number 3 using head and tail:

cat fruit\_data.txt -n | tail +3 | head -1

3 cherry 15 2.3

It is worth noting that although more and less do not output a text file and thus cannot be used to the left of a pipe operator, they can be used at the end of the pipeline to view the resulting output if it is long and does not fit on a screen. For example, if we want to remove the Project Gutenberg header and footer from Moby Dick using head and tail, it is useful to check which lines the header and footer end and start at respectively. This can be done using

cat -n pg4300.txt | less

We can see this way that the header ends and the text itself starts on line 24 and the novel ends and the footer starts on line 32865. Thus we can clean the text like this:

cat pg4300.txt | head -32865 | tail +24 | cat -s > pg4300\_clean.txt

Text cleaning is very often a fundamental part of the preprocessing workflow that is required for text mining and other natural language processing tasks and can be effectively carried out using the Linux CL tools introduced in this chapter.

Another example of piping, combined with a simple Linux shell loop, transposes a csv table, assuming that the table has three columns. We iterate over each of the three columns (thus seq 3) of the csv table and select them using cut -f, then transpose each of them using paste -s column by column. The result is printed to the terminal, but could be piped on to another command such as column for further processing or formatting or redirected to an output file.

for i in `seq 3`; do cut -d, -f$i fruit\_data.csv | paste -s -d, ; done

apple,cherry,banana,apple

10,15,20,25

3.5,2.3,1.2,4.0

When piping a text file to a command that requires two or more text files as argument, such as paste or diff, the hyphen character - is used to signal that one of the files comes from standard input instead of being read from disk. In the following simple, although not very useful example, we first remove the first and last line from a file, and then compare the result to the original file:

cat compare\_text1.txt | head -n-1 | tail +2 | diff - compare\_text1.txt

0a1

> Alice in Wonderland

7a9

> The Pool of Tears

The output shows that the first and last line are still present in the input file but missing from the processed output of the pipeline that is passed to diff using the - argument.

# Searching and Replacing Using Regular Expressions

This chapter introduces regular expressions (regexes), a fundamental tool for efficient text processing. Although regular expressions are supported in most programming languages, including Awk and Perl, which we will discuss later, as well as R, Python, and even most good text editors and IDEs such as Notepad++ or Visual Studio Code, in this chapter we will focus on the integration of regular expressions with two essential Linux utilities which we have not discussed in the previous chapter: grep for searching and sed for text replacement. We will discuss how to construct regex patterns to locate and manipulate text across files and streams. Regex concepts have a wide range of practical application in solving common text-processing tasks, such as filtering log files, extracting data, or performing batch updates to text.

In the first section, we will use grep to explore the basics of searching text using regex patterns. We will introduce topics like character sets, placeholders, anchors, greedy versus non-greedy quantifiers, grouping, alternation, and lookahead, which will allow us to write patterns for complex search tasks. In the second part of the chapter, we will move on from searching to modifying text with `sed` systematically. This section will introduce the concept of captured groups and show how to reference them in replacement patterns, enabling text transformations like reformatting dates or rearranging names.

## Searching Using Regular Expressions With grep

Regular expressions (regex) are a tool that enables users to search and manipulate strings based on specified patterns. They serve as a compact and flexible way to define search criteria for text processing tasks. Unlike simple keyword searches, regexes allows you to describe complex patterns, such as finding all email addresses, identifying lines containing dates, or extracting specific log entries. Their utility spans numerous fields, including system administration, data analysis, software development, web scraping and, most relevantly to us, processing of natural language texts.

In the context of Linux, regex plays a central role in command-line tools, most importantly grep and sed. Thanks to regular expressions, these tools are highly effective at sifting through large volumes of text or automating repetitive editing tasks.

The usefulness of regexes lies in their ability to handle a wide variety of tasks with a concise syntax, but this same feature can make it seem confusing at first, much like using a command-line interface. We will gradually introduce more complex regex concepts gradually after reviewing simple patterns. The difficulty in learning regular expressions lies in their rather compact syntax which can make them resemble a random assortment of characters with no obvious structure. For this reason, learning regular expressions can appear like a daunting task to someone not familiar with them.

grep is a command-line utility for searching plain-text files for **lines** that **match a regular expression** pattern. Its name comes from a command in the ed Unix line editor application, g/re/p (**g**lobally search a **r**egular **e**xpression and **p**rint), which works like grep. The grep command has a huge number of features that could fill a whole textbook. Its most common usage is finding lines in the input using a specified regex. Its basic syntax is

grep [options] 'pattern' file

As we have seen for other Linux text-processing applications, the file argument can be omitted or replaced by a simply hyphen -. In this case, grep processes the lines that it receives from standard input. Also, instead of a single file argument, an arbitrary number of file names or names with the wildcards ? or \* can be specified. The simplest regular expression is just a sequence of characters. These characters are interpreted **literally**, and lines containing this string are returned by grep. All other lines are omitted.

grep\_ex1.txt

apple

banana

mango

fig

tango

every

Any

an

an-organic

grep 'an' grep\_ex1.txt

b**anan**a

m**an**go

t**an**go

**an**

**an**-org**an**ic

The -i option activates case-insensitive search:

grep -i 'an' grep\_ex1.txt

b**anan**a

m**an**go

t**an**go

**An**y

**an**

**an**-org**an**ic

The -w option searches for whole words, i.e. if the pattern matches only part of a word in a line, that line is filtered out. For grep, a word is defined as a sequence of so-called word characters, where word characters are essentially defined as letters and numbers. Thus punctuation and other symbols are not word characters and thus delimit a word in the sense of grep and more generally in the sense of regex.

grep -w 'an' grep\_ex1.txt

**an**

**an**-organic

Similarly, -x matches whole lines.

grep -x 'an' grep\_ex1.txt

**an**

-c returns the number of lines that matched the regex rather than the lines.

grep -c 'ango' grep\_ex1.txt

2

Other options include: -v for inverting results, i.e. only printing lines that do not contain the pattern; -l for printing the names of files that contain the pattern, which is useful if many files are being searched in, e.g. grep -l 'mango' \*.txt. If the file searched in is a directory (or there are several files, some or all of which are directories), the -r option recurses through the content of all these directories.

The -n option adds line numbers, and -o, prints the matching substrings on the lines rather than the whole lines. The latter option is irrelevant for literal patterns but is useful for patterns that contain so-called metacharacters. Metacharacters stand not for themselves but have some special function in a regex. So-called “basic” regular expression syntax in grep understands the following metacharacters:

* ^ stands for the **beginning of a line**
* $ stands for the **end of a line**
* . stands for **any character**
* [ ], i.e. a pair of square brackets, surrounds a **specific** **set of characters**. Such a bracketed group of characters stands for any **one** of the characters contained in the bracket.
* **\*** is a **quantifier** that means that the preceding character or metacharacter can occur any number of times, from zero to arbitrarily many times, in a string that matches the regular expression.

Examples:

grep '^Even ' pg2701.txt finds all lines starting with “Even”, followed by a space in Moby Dick.

Even as it was, I thought something of slipping out of the window, but

Even now I lose time. Good-bye, good-bye. God bless ye, man, and may I

Even Ahab is a braver thing—a nobler thing than \_that\_. Would now the

grep ' d[iuo]g ' pg2701.txt finds all lines containing the words dig, dug or dog between a pair of spaces.

all over like a Newfoundland dog just from the water, and sat up in

the same sort of life that lives in a dog or a horse.

elephant stands in much the same respect to the whale that a dog does

though at times found on the sea-coast, is also dug up in some far

and ignores you, though you dig foundations for cathedrals. Yet was

up the sea air as a sagacious ship’s dog will, in drawing nigh to some

grep ' k[aeiou]\*t' pg2701.txt matches words that start with a k, followed by 0 or more vowels, followed by a t:

to the kitchen, and bawling out “clam for two,” disappeared.

However, a warm savory steam from the kitchen served to belie the

would try a little experiment. Stepping to the kitchen door, I uttered

Hussey! apoplexy!”—and with these cries, she ran towards the kitchen, I

harpooneers—all kith and kin to noble Benjamin—this day darting the

the eye, as that sometimes he is loathed by his own kith and kin! It is

a sunbeam! Hurrah!—Here we go like three tin kettles at the tail of a

of his dignity, and kitten-like, he plays on the ocean as if it were a

Brackets not only allow a list of characters, but also a range of characters to be specified. A range is interpreted as all characters in the ASCII (or Unicode) code page the code of which falls between the first and the last character specified. Ranges are specified in the form c-d, where c and d are single characters, with a hyphen character between them. The most commonly used ranges are a-z, A-Z (all lower-case and upper-case ASCII letters respectively) and 0-9 (all digits), but other ranges where the first character’s code (before the hyphen) is smaller than the second character’s code are permitted, e.g. d-x.

grep -o ' [hH]arpoon[a-z]\* ' pg2701.txt | sort | uniq finds all word forms starting with *harpoon* in Moby Dick:

harpoon

harpooned

harpooneer

harpooneers

harpooner

harpooning

harpoons

If an opening square bracket is followed by a ^ character, the bracketed part refers not to the set of characters listed, but rather to the **complement** of this set, i.e. any character but the ones listed. For example, the following regex finds all words in Moby Dick that start with *wh,* end with *le,* and have any character but *i* or *o* in between:

grep -o ' wh[^oi]le ' pg2701.txt | sort | uniq

whale

Regex also defines special notations for certain character classes. The notation \s stands for any whitespace character, most importantly space and tab in the case of grep. \S stands for any non-whitespace character. \w stands for any word character (recall that these are the letters of the alphabet and numbers, plus the underscore character \_). \W stands for non-word characters. The interpretation of \w is somewhat similar to that of [a-zA-Z0-9\_], but not quite the same. Notably they differ for Unicode letters. While the latter pattern obviously does not match Hungarian letters with diacritics, for example, \w does match them; note the different between the two outputs:

echo "árvíztűrő" | grep '\w'

**árvíztűrő**

echo "árvíztűrő" | grep '[a-zA-Z0-9\_]'

á**rv**í**zt**ű**r**ő

Finally, one further group of special characters that start with a backslash (such notations are often called “escape sequences”, and prepending a character with a backslash is often referred to as “escaping” that character) are the literal counterparts of the symbols that are used as metacharacters in regex syntax. Thus while ^, $, [, ], . and \* have a special interpretation within a regex pattern, their escaped versions \^, \$, \[, \], \. and \\* refer to the hat, dollar, bracket, dot and asterisk (star) characters themselves in a pattern. Of these, the \. character is most frequently used, the others are less commonly needed usually.

The following two patterns match times specified in 24-hour HH:MM format:

[01][0-9]:[0-5][0-9]

2[0-3]:[0-5][0-9]

Note that the first pattern only works up to 19:59. If we added 2 to the first set, that would not only match hours up to 23, but also non-hour strings from 24 to 29. The basic regex syntax defined in grep does not allow us to express all and only the well-formed hours using one single pattern.

The so-called **extended** regex syntax is activated using grep’s -E option. It adds a number of further metacharacters: | ( ) { } ? and +. (In fact, GNU grep’s basic regex syntax allows us to express the same functions as these metacharacters, but we have to escape these symbols with a backslash if we want to use them not as literal characters but for grouping, disjunction or quantification without adding the -E option.)

The vertical line character | expresses **disjunction** (“or”). If used without adding parentheses, it matches sequences that match either to the left of |, or to the right of |, or between a pair of | characters if more than one appears in the pattern string:

grep -E ' snow | rain ' pg2701.txt

all, one grand hooded phantom, like a snow hill in the air.

such fervent rays, that it seemed to have melted the packed snow and

occasional squall of sleet or snow would all but congeal his very

floor-screwed chair; the rain and half-melted sleet of the storm from

fac-similes of magnified Arctic snow crystals. I mean no disparagement

of alpine land lying along the snow line. Few are the foreheads which

track the snow prints of the deer. But in the great Sperm Whale, this

**Alternative** subpatterns can be **grouped** within a pair of **round parentheses**, like in the following example, in which we combine our previous two HH:MM patterns into one:

([01][0-9]|2[0-3]):[0-5][0-9]

The other metacharacters are used as **quantifiers**. + and ? are added to the right of a character, a character set, or a parenthesised group, just like the \* quantifier. While \* means ‘any number of occurrences, including zero’, + means ‘any number of occurrences, but at least one, and ? means ‘at most one, i.e. zero or one occurrence’.

grep -Eo ' s[aeiou]+nds? ' pg2701.txt | sort | uniq finds words that contain at least one vowel between the starting letter *s* and *nd,* which is optionally followed by an *s*:

sand

send

sends

sound

sounds

The **curly braces** are used to specify the **exact number** of repetitions, either as a **single number or a range**. They are added immediately after the character, character set or group that they quantify, just like the other quantifiers. '[0-9]{3}' matches exactly three numbers, one after the other.

grep -Eo '\w\*[aeiou]{4,}\w\*' pg2701.txt | sort | uniq matches words that contain at least four subsequent vowel letters:

Hawaiian

obsequious

terraqueous

The following matches all year numbers from 1 up to 2024. In the first and second parts, the {,2} and{,3} match at most two or three elements of the number set. The first part before the first vertical bar matches all numbers between 1 and 999, the second matches 1000 to 1999, the third 2000 to 2019, and the last 2020 to 2024.

[1-9][0-9]{,2}|1[0-9]{,3}|20[01][0-9]|202[0-4]

Finally, the next pattern matches both HH:MM and HH:MM:SS time strings. Note that in this case a parenthesized group was created out of a literal character : and two number character ranges. The grouping only serves the purpose of adding the {1,2} quantifier to this three-character substring.

([01][0-9]|2[0-3])(:[0-5][0-9]){1,2}

The grep extended regex syntax is still quite limited compared to regexes supported by applications like Notepad++ or VS Code or programming languages, especially Perl. Perl provides an extremely powerful regex syntax that we will not be able to review here in detail, but a few functionalities are worth pointing out here, since the GNU grep command has a -P option that activates support for Perl-compatible regular expressions.

For languages other than English, matching Unicode characters e.g. in names is an important consideration when using regexes. Although we have seen that the \w word character class available in basic grep syntax supports Unicode letters, this does not allow for more fine-grained searches. Perl regex syntax allows users to match so-called Unicode character properties. Unicode characters are subdivided into several non-disjoint classes that are called character properties. These can be specified in patterns using the escape sequence \p{}, which the designation of the property added within the curly braces. The three most important properties are \p{Ll}, meaning lower-case letter, \p{Lu} for upper-case letter, and \p{L} for any letter. These replace the basic regex [a-z], [A-Z] and [a-zA-Z] patterns, respectively, for alphabets containing any non-English characters. Perl regex also introduces a shorthand notation \d for the digit class of characters, which is equivalent to and thus can be used to replace the much longer and less readable [0-9].

Perl regex add two important concepts that control the way the regular expression engine matches strings: non-greedy quantifiers and lookaround. Normally the quantifiers \* and + function in a greedy way, which means that they match as many characters as possible. In many situations, this is not ideal, as we want to specify the right boundary of a pattern (e.g. a space, a tab, some punctuation) but do not want to restrict what characters might precede it. In such situations it would be more comfortable to use a very general quantified pattern like .+\t or .\*\t (the \t escape sequence stands for the tab character in Perl-compatible regex). The problem with these is that they match up to the last tab in the string, not the next. The non-greedy version of these quantifier is denoted by +? and \*? respectively.

Compare the two outputs of grep -o with the greedy \* quantifier and its non-greedy version on the same text:

grep -Po 'whaler.\* ' pg2701.txt | sort | uniq

whaler

whaler are protected from the inclement weather of the frozen seas.

whaler at sea, and long absent from

whaler best fitted

whaler in any sort of decent weather? She has a “\_Gam\_,” a thing

whaler is

whaler we

whaler wonders soon wane. Besides, now and then

whaler! What does the whaler do when she meets

whaler,

whaler, who, in

whaler.

whaler. While other hulls are loaded down with alien stuff, to

whalers

whalers I know of—not all though—were such famous,

whalers frequently touch to augment their

etc.

grep -Po 'whaler.\*? ' pg2701.txt | sort | uniq

whaler

whaler!

whaler,

whaler.

whalers

whalers,

whalers.

whalers;

Lookaround (more specifically lookahead and lookbehind depending on the direction) allows the user to specify the left and right boundary of a regex pattern, especially in combination with non-greedy quantification. In previous examples when we tried to match words, the match included the preceding and following space surrounding the word that we were interested in, since this was the only way to specify that the word was surrounded by spaces. Lookaround makes it possible to separate the pattern to be matched and the context in which we want it to appear. Lookbehind specifies the left context and is denoted by the special character sequence (?<=PATTERN), lookahead the right context and is denoted by (?=PATTERN). Negative lookbehind specifies that the pattern is not allowed to appear as the left context of the pattern and is denoted by (?<!pattern), whereas negative lookahead (?!PATTERN) excludes the specified pattern from the right context. The following pattern matches words delimited to the left by a space, to the right by whitespace or various punctuation marks.

grep -Po '(?<= )whale.\*?(?=\s|[.,;?!—)])' pg2701.txt | sort | uniq

whale

whale-boat

whale-boats

whale-boat’s

whale-bone

whale-books

whale-craft

whale-cry

whale-e

whale-fastener

whale-fish

whale-fishers

whale-fishery

whale-fleet

whale-hater

whale-hunt

etc.

## Replacing Strings Using sed

The name of the sed tool is short for stream editor. Stream refers to the input stream being passed to the tool via shell pipes. This indicates that the command's primary functionality is to modify text read from standard input and print it to standard output. However, a file name can be also specified as its argument, and the tool can save the changes back to the same file if needed.

The basic syntax of this command is sed 's/regex\_pattern/replacement/flags' file. The s at the beginning of the string is fixed. The regex patterns to be used with sed are essentially identical to those we have discussed in connection with grep. They can contain literal characters, character classes, paranthesised groups and quantifiers. The character span of the input that matches the regex pattern is replaced by the specified replacement string. The replacement string typically consist of literal characters and references to strings that matched the regex pattern or that matched the groups within the pattern. Input lines that do not match the regex pattern are printed to the output unchanged, i.e. sed does not filter the lines contrary to grep.

A standard use of sed, in particular in language processing tasks, is to remove trailing whitespace from the ends of lines, leading whitespace from the beginning of lines, and sequences of more than one whitespace within lines. This is achieved using the following command, which prints the result to standard output.

sed -E 's/\s+$//; s/^\s+//; s/\s{2,}/ /g' pg2701.txt

Extended regex syntax needs to be activated using the -E option, like for grep. Several substitute commands can be issued within a single string to sed, separated by semicolons, as an alternative to chaining several sed commands one after the other using pipes. The g flag is usually specified for all sed replacements. It stands for global replacement, which means that all occurrences of a pattern are replaced on each line. If it is not specified, then only the first occurrence of the pattern is replaced per line. For leading and trailing whitespace this is irrelevant since there can only be at most one such sequence on each line, but there might be more than one line-internal long whitespace sequence.

The entire matching substring can be referenced in the replacement string using the & character. In the following example, a $ symbol is added to the left and a zero appended to the right of the values in the third column of fruit\_data.txt:

sed 's/[0-9]\.[0-9]/$&0/' fruit\_data.txt

apple 10 $4.50

banana 20 $1.20

cherry 15 $2.30

apple 25 $4.00

Parenthesised groups in the regex pattern can be backreferenced using \NUM references, where NUM is the number of the group, i.e. \1 for the first group, \2 for the second group, etc.

The following example transforms the date format YYYY/MM/DD into YYYY. MM. DD.:

echo '2012/05/30' | sed -E 's#([0-9]{4})/([0-9]{2})/([0-9]{2})#\1. \2. \3.#g'

Note that in this example the / separator of the string passed to the sed command was replaced by #. Although the / character is used usually by default, essentially all special characters except the backslash can be used instead as separator. In particular, # is commonly used if either the regex or the replacement string would include a / character, since otherwise this would have to be escaped as \/, which makes the sed string hard to read and is best avoided.

Another example to transform DD/MM/YYYY to YYYY-MM-DD:

echo '30/05/2012' | sed -E 's#([0-9]{2})/([0-9]{2})/([0-9]{4})#\3-\2-\1#g'

Such numbered backreferences to groups are also permitted within the regex part of the sed string as well as in grep regexes, as in the following example, where we select lines that contain the same word repeated twice:

grep -E '( \w+)\1 ' pg2701.txt

that the landlord, after all, **had had** no idea of fooling me—but at the

the old Mogul has fixed him, too. I twigged it, knew it; **had had** the

suppose **that that** poor fellow there, who this moment perhaps caught by

fellow-critters, **dat dat** woraciousness—’top dat dam slappin’ ob de

or two **that that** society passed a resolution to patronize nothing but

apprised **that that** individual’s intention was to land him in the first

Whale’s on the larboard; did you never hear, Stubb, **that that** ship can

Stubb, do you suppose **that that** devil you was speaking of just now, was

As good luck would have it, they **had had** a whale alongside a day or two

Um-m. So he must. I do deem it now a most meaning thing, **that that** old

The -i option overwrites the input file with the changes. This should be used with care, as it will lead to loss of data otherwise. If an argument is specified for this option, a backup of the original file is saved with this argument string appended to the input file’s name, e.g. fruit\_data.txt.old in this example:

sed -i.old 's/apple/&s/' fruit\_data.txt

Apart from the replacement functionality discussed, sed can also be used to filter lines exactly like grep if only a regex pattern is provided using a pair of delimiters instead of three. The -n option disables printing of all lines, and the p flag prints matching lines.

sed -n '/apple/p' fruit\_data.txt

In addition, sed also allows the user to specify a regex condition in addition to the substition string, with the effect that the replacement is only triggered if the line matches the regex condition. Furthermore, apart from the g flag, other flags can be specified, e.g. a number like 3, with the effect that only the third occurrence matching the regex pattern on a line is replaced. These options are of little practical utility in most cases, and we therefore do not go into them further. For situations where more advanced patterns are required than sed functionality, it is generally recommended to use perl instead of sed. Perl substitution with regexes offers other useful features, such as the ability to specify named groups in a regex, making it possible to reference these groups by name instead of by number. This can make complex replacement patterns far more transparent.

# Using the awk programming language to manipulate tabular data

## What is awk?

awk is a powerful, versatile text-processing tool that is both a special-purpose programming language for various text processing tasks, and a command-line utility. It was created in the 1970s by Alfred Aho, Peter Weinberger, and Brian Kernighan, three eminent computer scientists. Its name is derived from their initials and is pronounced like the name of the bird auk. This language was designed specifically to handle pattern-based text processing and manipulation, making it ideal for extracting, transforming and analysing structured text data such as logs, reports, and CSV files.

awk operates by scanning input text line by line, applying specified patterns and actions to it. It is particularly useful for dealing with data that is organized into fields (columns) and records (rows). While the tools we have discussed earlier, such as grep and sed, work with individual lines to filter and edit them respectively, awk combines both searching and text transformation capabilities, and furthermore adds support for field-based processing, conditional logic and arithmetic operations. It can filter, transform and summarize text, and can thereby replace complex pipelines.

There are several situations in which awk is a more adequate choice than other Linux tools. Using a very succinct but still intuitive syntax, awk can extract specific fields, compute values, and reformat text efficiently. Thus it is possible to write compact one-liner programs in awk and execute them from the Linux command line. For example, the following command extracts and prints the first and third columns from the specified file.

awk '{print $1, $3}' fruit\_data.txt

apple 4.5

banana 1.2

cherry 2.3

apple 4.0

awk inherently recognizes and processes fields (columns) separated by delimiters such as spaces, tabs, or commas. This makes it particularly useful for structured data like CSVs. In addition, it includes programming features like loops, conditional statements and associative arrays and therefore provides scripting capabilities. This can be used, for example, for aggregating data, such as summing values grouped by a category. It is also excellent for summarising and analysing data – e.g. computing average, min, or max values from a column of numbers – directly from the command line without needing further tools like Excel or Python.

Typical uses of awk include log file analysis, where it is used to extract error messages, summarise response times or count specific events in server logs. It can be used for basic data cleaning and transformations, like preparing CSV files for further analysis by reordering, filtering, or modifying columns. It also has applications in report generation, as it can automate the creation of tabular or summary reports from raw data.

Although awk can be used to write longer programs in a text editor, it is typically used for one-liners, and thus in this chapter we will focus on running very short awk programs by passing them to the awk interpreter from the command line.

## Command Structure and Default Behaviour

The fundamental structure of an awk command is as follows:

awk 'pattern {action}' input\_file

“**Pattern**” specifies a **condition** that determines which lines of the input file are processed. Patterns can include text matches, regular expressions, or logical conditions. If omitted, **all lines** are processed. “**Action**” is a block of code, enclosed in {}, that specifies **what to do** with the matching lines. If omitted, the default action is to **print the entire line**, which is referred to by the variable $0 in awk.

awk '{print}' file.txt prints all lines of a file, as no condition has been provided. Note that like in most languages, the command to print is just print.

awk '/error/' file.txt prints only lines containing the string „error”. Note that this command only consists of a pattern, and therefore the default action of printing the line is triggered.

An awk program consists of a **sequence of pattern-action** statements. If a program contains more than one pattern-action statement, these can be separated by semicolons to improve readability, but a semicolon is not required between these structures. Each action block can contain more than one command statement. Command statements within an action block must be separated by semicolons. The final statement within an action block need not be closed by a semicolon, but one can be added optionally.

By default awk **splits** all lines it reads from the input **into fields**, which can be accessed using the variables $1, $2, etc., and printed, for example, like in the following command, which prints the first and third field from all lines.

awk '{print $1, $3}' file.txt

The notions of field and record are crucial for awk. A **field** is a column within a line. Fields are separated by a delimiter. By default, awk assumes that the delimiters are sequences of whitespace, i.e. spaces or tabs. As already mentioned, fields are referred to by special numbered variables. $0 represents not a field, but the entire line. $1, $2, ... $NF: Represent the first, second, ... last fields, respectively.

**Records** are **rows** of data, separated by a record separator. By default, awk assumes that the record separator is the newline character.

The following built-in variables are used by awk:

* FS input field separator
* OFS output field separator
* NF number of fields
* RS input record separator
* ORS output record separator
* NR number of records (i.e. line number) that awk has processed since the beginning of the execution
* FNR number of records per file

These can be referred to like this:

awk '{print $NF}' file.txt prints the last column in each line.

awk '{print NF}' file.txt prints the total number of fields in each line.

The values of the build-in variables can be changed during the execution of a program using the assignment operator = like in the following example:

awk 'BEGIN {FS=","; OFS="\t"} {print NR, $1, $3}' fruit\_data.csv

1 apple 3.5

2 cherry 2.3

3 banana 1.2

4 apple 4.0

Here the input field separator is changed to comma, with the effect that awk separates the columns of a csv file. The output field separator is changed from space to tab, i.e. a tab-separated (tsv) file is output. The line number is printed on each line using the NR built-in variable. Commands within a block are separated by a semicolon, as shown in the BEGIN block.

An awk program has special BEGIN and END blocks to execute code before or after processing the input respectively, which was seen in the previous example. The **BEGIN block** is for setting up variables or printing headers, e.g.

awk 'BEGIN {print "Name", "Score"} {print $1, $2}' file.txt

Note that in this case the print command prints two strings as output fields in the BEGIN block to the first output line and two fields extracted from the input document on all subsequent output lines.

The **END block** can be used to print summaries. The following awk program uses the += combined addition and assignment operator to increment the value of the (non-built-in) variable sum, which is silently initialized to 0, by the numerical value of the second column on every input line, and prints the final value of the variable after all input has been processed:

awk '{sum += $2} END {print "Total:", sum}' fruit\_data.txt

Total: 70

## Patterns

The awk language defines the following patterns:

* BEGIN { *statements* }: as we have seen, *statements* are executed once before any input has been read.
* END { *statements* }: *statements* are executed once after all input has been read.
* *expression* { *statements* }: *statements* are executed at each input line where the expression is true.
* /*regular expression* / { *statements* }: *statements* are executed at each input line that contains a substring matched by the *regular expression*
* *compound pattern* { *statements* }: *statements* are executed at each input line where the compound pattern is true. A compound pattern consists of basic expressions (conditions) combined by the logical operators && (meaning AND), || (OR) and ! (NOT).
* *pattern1 , pattern2* { *statements* }: This is called a range pattern. It matches each input line from a line matched by *pattern1* to the next line matched by *pattern2*. The *statements* are executed at each line within this matching range.

### Expressions

Expressions used as patterns can be built up out of string and number **literal values**; **variables**; **associative** **arrays;** **fields;** and **function calls**, all of which possess a value. These value-denoting expressions can be combined using the **arithmetic** **operators** +, -, \*, /, % (remainder of division) and ^ (exponentiation); the **comparison operators** < (less than), <= (less than or equal to), == (equal to), != (not equal to), >= (greater than or equal to), > (greater than); **regex** **matching operators** ~ (matches) and !~ (does not match); and **logical operators** && (and) , || (OR), ! (NOT).

Arithmetic operations on numbers work as expected. Identity of two string values can be checked using the == and != operators:

awk '$1 == "apple" {print $0, $2 \* $3}; $1 != "apple" {print "other fruits", 0, 0, 0} ' fruit\_data.txt

apple 10 4.5 45

other fruits 0 0 0

other fruits 0 0 0

apple 25 4.0 100

Comparison operators like > and < are defined for strings but not very useful in most cases. The value of the < operation is true if the first string comes before the second string in lexicographical ordering, and accordingly for the other three relational operators.

There is only one operation in awk that is specific to pairs of strings, which is **concatenation**. It does not have an explicit operator. String concatenation is achieved by writing string literals, string-valued variables, fields, array elements, function values, and other expressions next to one another, like in this example:

{ print NR ":" $0 }

This prints each line preceded by its line number and a colon, with no spaces in between. The number NR is converted to its string value (and so is $0 if necessary); then the three strings are concatenated and the result is printed.

As opposed to languages like C and Java, arrays in awk are **associative arrays** in which values are mapped to keys using a so-called hash map. This is the same concept as the one that is called dictionary (dict) in Python, hash in Perl and map in Java. Arrays in awk are not declared, their elements are initialised by being mentioned and being assigned a value like this:

x[NR] = $0

This statement assigns the current input line to element NR of the array x. This element functions like a variable from now on and can be printed, for example, using print x[1] or print x[2], etc., for each line. (Recall that the variable NR takes values from 1 to the number of records in the input as awk processes the input line by line.) The keys of associative arrays need not be numbers, they can be strings as well. Thus the following is a legal assignment statement:

pop["Asia"] += $3

In the action part of an pattern-action statement, a loop can be used to iterate over all keys in an associative array like this:

END { for (line\_number in x) print line\_number, x[line\_number] }

The following program stores every line of the input memory and prints them using such a loop after all lines have been read:

awk '{x[NR] = $0} END {print "Lines read:"; for (line\_number in x) printf "Line %d: %s\n", line\_number, x[line\_number] }' fruit\_data.txt

Lines read:

Line 1: apple 10 4.5

Line 2: banana 20 1.2

Line 3: cherry 15 2.3

Line 4: apple 25 4.0

### Numerical and String Functions

awk has several built-in functions to work with numbers and strings. The numerical functions are, in alphabetical order:

* atan2(y,x) arctangent of y/x
* cos(x) cosine of x, with x in radians
* exp(x) ex
* int(x) integer part of x
* log(x) natural (base e) logarithm of x
* rand() random number r, where 0 <= r < 1
* sin(x) sine of x, with x in radians
* sqrt(x) square root of x
* srand(x) set x as the new seed for rand( )

String functions are:

* gsub(r,s) substitutes s for r globally in $0, returns number of substitutions made
* gsub(r,s,t) substitutes for r globally in string t, return number of substitutions made
* index(s,t) return first position of string t in s, or 0 if t is not present
* length(s) return number of characters in s
* match(s,r) test whether s contains a substring matched by r, return index or 0
* split(s,a) split s into array a on FS, return number of fields
* split(s,a,fs) splits into array a on field separator fs, return number of fields
* sprintf(fmt,expr-list) return expr-list formatted according to format string fmt
* sub(r,s) substitutes for the leftmost longest substring of $0; matched by r, return number of substitutions made
* sub(r,s,t) substitute s for the leftmost longest substring oft; matched by r, return number of substitutions made
* substr(s,p) return suffix of s starting at position p
* substr(s,p,n) return substring of s of length n starting at position p

These numeric and string functions can be legitimately used in both the pattern and the action part of a pattern-action statement. Most string functions, in particular the substitution and the split function, are of little use in the pattern part and are typically used in action statements.

### Regular Expressions as Patterns

Regular expressions can be used to match lines similarly to grep if a regular expression pattern is specified. Regexes must be delimited by a pair of / characters. GNU awk supports extended regular expressions like the grep -E option, i.e. all greedy quantifiers, numerical quantifiers, grouping and the disjuncition operator, but not additional capabilities like non-greedy quantifiers, \d or Unicode properties. However, as mentioned above, a regular expression matching operator can also be used to build up and expression pattern. The regular expression is placed to the right of the ~ or !~ operator, and the string to be matched with to the left.

The following example selects lines that contain a date from August or September by matching a regex with the second (date) field:

awk '$2 ~ /-0[89]-/' sample\_date.txt

101 2022-08-11

102 2023-09-02

## Actions

The action part of a pattern-action statement can consist of either a single statement, which include:

* **assignment statements** which assigns a **value to a variable**. These are formed with a variable name (to which a new value is assigned) and one of many various assignment operators. The assignment operators include the **increment and decrement operators** ++ and -- (increase and decrease the value of a variable by 1 respectively); the simple **assignment operator** = (preceded by a variable name, followed by an expression the value of which is assigned to the variable), and **assignment operators combined with an arithmetic operation** +=, -=, \*=, etc., which are preceded by a variable name and followed by an expression that has a numerical value.
* a print command, which takes a list of expressions and prints their value; or a printf command, which takes a format string as argument and as many further expressions as there are placeholders in the format string.
* a **conditional statement**, which can be either if ( *expression* ) *statement* or if ( *expression* ) *statement* else *statement*
* **looping statements**. awk has normal for loops like C or Java; foreach loops which have the form for (*variable* in *array*) *statement*; while loops; and do ... while loops.

Alternatively the action can be a **block of several statements** enclosed by { }.

We do not discuss conditional and looping statements in more detail, but refer to the next section for examples that contain such constructs to illustrate what they can be used for in awk.

## Running awk Scripts from a File

Although in this chapter we focus on one-line awk programs that can be passed as an argument to the awk command on the command line, it can be useful to write more complex programs. We illustrate this with a program that prints a summary based on dates appearing in a table after all lines have been read, and uses an associative array to count relevant lines.

Although awk supports substitution based on regexes using the built-in function gsub, it might be easier in most cases to use sed instead and pipe the results to awk. Accordingly, this example uses sed to transform the dates into a format more easily processed by awk:

cat sample\_date.txt | sed 's/-/ /g' | awk 'BEGIN {OFS="\t"; print "year", "month", "day"}; {print $2, $3, $4; samples[$2] += 1}; END {OFS=""; for (year in samples) printf "\n%d sample(s) from %d", samples[year], year; print ""}'

year month day

2022 08 11

2023 09 02

2023 10 14

2023 11 28

1 sample(s) from 2022,

3 sample(s) from 2023

sed was used to replace the month and day separator hyphens by spaces, which awk treats as input field separators by default. This example shows that even relatively simple awk scripts can become long and hard to read on a single line. Thus, although it is technically possible to pass a program like this to awk on the command line, it is not necessarily good practice and it definitely stretches the notion of a one-liner. If an awk program is more complex than what would fit on a single line, it is better to save it as a script file. The program above would look like this if formatted in a nicer way:

BEGIN {OFS="\t"

print "year", "month", "day"}

{print $2, $3, $4}

{samples[$2] += 1}

END {OFS=""

for (year in samples) printf "\n%d sample(s) from %d", samples[year], year

print ""}

If we save this program in a script file named *reformat\_dates.awk*, we can run it using the -f option like this:

cat sample\_date.txt | sed 's/-/ /g' | awk -f reformat\_dates.awk

The output is the same as above.

## Some examples

The following one-liners serve as examples to demonstrate some useful use cases of awk. They are taken from the original manual of the language titled *The AWK Programming Language* by A, K and W.

END { print NR } prints the total number of input lines.

NR == 10 prints the tenth input line.

{ print $NF } prints the last field of every input line.

{ field = $NF} END { print field } prints the last field of the last input line.

NF > 4 prints every input line with more than four fields.

$NF > 4 prints every input line in which the last field is more than 4.

{ nf = nf + NF } END { print nf } prints the total number of fields in all input lines.

/Beth/ { nlines = nlines + 1 } END { print nlines } prints the total number of lines that contain Beth.

$1 > max { max = $1; maxline = $0 } END { print max, maxline } prints the largest first field and the line that contains it (assumes some $1 is positive).

NF > 0 prints every line that has at least one field.

length($0) > 80 prints every line longer than 80 characters.

{ print NF, $0 } prints the number of fields in every line followed by the line itself.

{ print $2, $1 } prints the second field, then the first field of every line.

{ temp = $1; $1 = $2; $2 = temp; print } exchanges the first two fields of every line and then print the line.

{ $1 = NR; print } prints every line with the first field replaced by the line number.

{ $2 = ""; print } prints every line after erasing the second field.

{ for (i = NF; i > 0; i = i - 1) printf("%s " $i); printf ( "\n" ) } prints the fields of every line in reverse order. Note that this program contains a for loop, which is an awk programming construct that we have not touched upon. The awk for loop is a true for loop like the one in C or Java, not a foreach loop like the one in Python or Perl.

{ sum = 0; for (i = 1; i <= NF; i = i + 1) sum = sum + $i; print sum } prints the sums of the fields of every line.

{ for (i = 1; i <= NF; i = i + 1) sum += $i } END { print sum } adds up all fields in all lines and prints the sum.

{ for (i = 1; i <= NF; i = i + 1) if ($i < 0) $i = -$i; print } prints every line after replacing each field by its absolute value

# Text Processing using the Perl Programming Language

## General Characteristics of Perl

Perl is a high-level, general-purpose programming language that was originally developed in 1987 by a computational linguist. It was specifically designed to be a powerful text-processing tool and became very popular for tasks that involve manipulating and extracting data from text files especially in Unix/Linux environments to complement the already substantial capabilities of utilities that are standard components of these operating systems. Historically, Perl was a key language for early web programming, especially with CGI (Common Gateway Interface) scripts that handled web forms and server responses. Perl has since evolved into a versatile language used for a wide range of applications especially related to Unix/Linux system administration, web development, natural language processing and bioinformatics. Its text manipulation strengths are well-suited to all of these fields.

Among all major programming languages, Perl has the best integration of regular expression pattern matching and processing capabilities with other components of the language. The expressivity of the language of regular expression patterns in Perl also surpasses the built-in regex capabilities of other programming languages and utilities like grep and sed. In addition, it also permits the reading and writing of text files line by line in an exceptionally succinct way. Accordingly, it is often used to parse, transform, and extract data from text files, such as log files or reports.

However, Perl has many uses in addition to text file processing. It has a very large repository of third-party modules and libraries available via an online repository called CPAN (Comprehensive Perl Archive Network), which extend the language’s capabilities and offer pre-built solutions for many tasks including e.g. communicating with databases or programming GUIs. Although it is especially closely connected to Unix/Linux, it runs on most platforms including Windows and macOS.

Since Perl has been widely used for several decades, it remains important in some areas, notably including bioinformatics, because of the legacy code that was written in this language and is still being used today. Although Perl is clearly not as important and widely used in this area today as it was 20 or even 10 years ago, and has been mostly replaced by R and especially Python, it is being used by many programmers, especially older bioinformaticians. In rankings of all programming languages across all industries (and notably not within scientific computing, and bioinformatics in particular) such as the TIOBE index, which are based on the number of programmers who state that they are using the language in question, as well as on the number of jobs posted for programmers, Perl holds a respectable position around the 25th place, with Python occupying 1st place and R around 20th. If we examine the number of pull requests and pushes on GitHub (https://madnight.github.io/githut), which reflects the popularity of the language in the open source software development communities rather than their use in corporate contexts, Perl is placed even higher, around 20th, well above R. These numbers confirm that Perl remains very much a living and relevant language today with an active community that is well worth getting acquainted with even apart from its usefulness for text processing.

Perl is a very flexible and versatile language which allows programmers to express their code in several different ways. It is a multi-paradigm language, meaning it supports procedural, object-oriented, and functional programming styles. In addition, it embraces the philosophy of “there is more than one way to do it”, jokingly abbreviated as “TIMTOWTDI”, which means that the same idea can be expressed using more than one synonymous language constructs. This flexibility of expression allows developers to choose the approach to programming that they prefer. While this might appear to be a positive thing, it often means in practice that Perl does not enforce a transparent and easy to read programming style. Instead, it allows and even encourages the programmer to use shortcuts and a highly compact style which tends to result in code that may be quick to write but is also very hard to decipher. Such program code is difficult to debug and maintain more generally. Although this problem is rooted in Perl’s philosophy, best practices for modern Perl programming

3.2.

# A Brief Introduction to Text Mining

Text mining is a general concept that encompasses all types of text processing that deal with **finding, organising and analysing information** from text sources. More specifically, text mining is usually considered to be restricted to text processing that aims to extract summary data from **very large bodies of text**, at least thousands, but typically millions of documents, which cannot be effectively processed with simple text processing techniques like basic search and are too large to be summarised by humans reading these texts.

Extraction of summary data means that **new information** is being created as a result of the text mining process, which is what distinguishes text mining from activities such as text search and information retrieval (the area of computing concerned with building search engines e.g. for text collections such as libraries or the web). The latter do not aim to create new information but rather to find exactly the pieces of information in very large bodies of text that are already contained in it and that are needed by the user. For example, finding the date of birth of a certain celebrity is a difficult problem to be solved by web search engines, but it is a problem for which the answer is either contained in some documents online, in which case it is a solvable problem, or it is not to be found anywhere, in which case the answer to the question is unknowable.

Text mining, by contrast, tries to answer questions the answers to which are **not contained in any single document** (or even in any combination of documents) in the body of text that is being mined. For example, a typical question that is asked in text mining is: What are the main topics discussed in a given collection of documents, and which documents are concerned with which topics? More generally, the new information that text mining tries to create is concerned with a pattern, a trend or a relationship that cannot be established by examining single documents. Text mining might be interested in creating statistical analyses of certain issues and visualising important patterns, such as analysing trends like Google Ngram Viewer and Google Trends (although the latter is based on search terms entered in a search engine rather than texts, but the general principle is the same) or finding sometimes not obvious relations between concepts (for example, by examining word co-occurrence graphs). Word clouds are a popular option to visualise important words in a collection of texts, although typically at a much smaller scale than what text mining is generally interested in.

Apart from determining trends and summarising and visualising the entirety of a collection of texts, another important application of text mining is to **synthesise (summarise)** what is contained in the collection of documents **on a certain topic**. For example, if we ask the question what transcription factors have been shown to regulate the expression of a certain gene, the answer to this question is likely **distributed over a potentially very large number of documents**. A negative answer, i.e. that we do not know of any such transcription factors, would require all remotely related literature to be checked (this paper discussing the gene in question does not mention any such transcription factors, that other paper does not either, etc. for all relevant papers). However, the main point is that a positive answer to such a question is not a search result that points to a specific sentence or a section in a particular paper, or even several such search results, but rather a summary, whether in the form of a natural language text, of a table, or some other form, that presents the overall results of such a complex literature review.

Another similar application of text mining applied to the **processing of scientific literature** would be establishing what the most influential papers published e.g. during the past 5 years are on a certain topic, e.g. antimicrobial resistance. To answer this question would involve several steps, each of which can be automated:

* make a list of what literature is being cited in sections or paragraphs of papers that discuss AMR,
* verify that the papers being cited are in fact concerned with AMR,
* count the citations,
* keep track of why each of these publications is cited, i.e. whether the citing author specifically agrees or disagrees with cited publication,
* determine either based on the citations or by examining the paper itself what aspect of AMR each of these influential publications focus on.

Text mining should be distinguished from other related fields of study and techniques that are also concerned with the processing of language. On the one hand, text mining is related to the scientific discipline of **corpus linguistics** which aims to discover patterns of language use in very large bodies of text. The clear difference between corpus linguistics and text mining is that corpus linguistics is concerned with linguistic patterns themselves, essentially what words are used in what contexts, in what syntactic structures, either appearing together with other words, or in the same contexts that certain other words (e.g. their synonyms or antonyms) would appear in. Text mining is not interested in language as such, but only in the content of documents. The specific syntactic structures and combinations of words by which these contents are expressed are not directly relevant. Secondly, text mining is related to the field of applied computer science known as **natural language processing**, commonly abbreviated as **NLP**, which is concerned with the processing of human languages by computers. NLP aims to create technological solutions for practical problems related to the processing of texts, such as creating summaries of long documents; find the answers to specific questions in a document even if the part of the document that contains the answer is phrased in very different words than the question; categorising a customer service enquiry (by email, chat or on the phone) so that the inquiry can be routed to the department or staff member who is best qualified to process it, etc. Thus, while NLP uses partly similar methods to text mining to process language, and although it makes sense to regard text mining as a branch of NLP, the scale and the goals of these two activities are markedly different.

## Text Processing Pipelines

In general it is not possible to use text directly in either NLP or text mining, but it needs to be run through a processing pipeline first. If this prior processing is omitted, text may be still be processed, but the results achieved may be far from the optimal results that are achievable if the text is properly processed.

### Language Identification

One important prerequisite is to filter texts based on their language. Processing tools are in general only able to process documents in a specific language, and different languages tend to require or at least work best with different tools, thus language identification is a necessary first step. Some documents might contain parts in several languages, e.g. citations, and thus be bi- or multilingual. Parts not belonging to the language currently being processed need to be handled, e.g. filtered out. Language identification is typically carried out using statistical language models that are based on the frequencies of short character sequences, 3-, 4- or 5-grams. A statistical model of e.g. 5-gram frequencies for all languages to be distinguished first needs to be created An alternative is to look for frequent words

### Cleaning up the Texts to be Processed

Texts that we want to mine are often not in the form that can be directly processed. For example, if the text was collected from the internet, HTML or XML markup need to be removed from it. If it was extracted from a pdf, things like page headers, page numbers, image captions, text boxes, footnotes, etc. need to be handled or removed so that they do not interrupt the text flow. Also in complex page layouts like pages containing several columns, e.g. what we see in most magazines or newspapers, it is a difficult task to determine which column comes after which. Words broken at the ends of lines need to be combined. If the document was digitised from printed pages by scanning and optical character recognition (OCR), errors that were introduced during the digitisation process, such as junk characters that are not really on the page but that are due to decorative lines, images, contamination on the page, scanning artefacts, pencil markings, etc. being incorrectly recognised as text, need to be filtered out. These are all difficult subtasks. In ideal situations, such as mining PubMed abstracts, the language identification and clean-up steps can fortunately be omitted or are at least substantially simplified, since these already contain documents that are already in English and served in a machine-readable form. Still some cleanup can be required here as well.

### Dividing the Text into Useful Units

In general it is not ideal to work with whole documents such as whole research papers in one go. For example, if we are interested in papers that discuss a certain biological phenomenon, we typically want to know not just whether the paper as a whole mentions it, but also where, in which sections or sentences of the paper it appears. In addition, most existing NLP tools and techniques expect the input text to be split up into shorter units, mostly sentences, which are then treated as the units of further processing. This subdivision of the text is not a trivial task either. For example, for documents digitised from paper, a line of text is typically not an ideal unit of processing. Sentences or paragraphs, which make more sense, need to be recognised first, since a longer line of text may contain more than one sentence, and a sentence may span several lines, both starting and ending in the middle of a line. Thus several lines making up a paragraph first need to be combined, and then sentence segmentation needs to be carried out. Some text structuring patterns such as bulleted lists, can further complicate this process considerably. For some document types and genres, e.g. mining PowerPoint presentations, a completely different approach is often necessary, since these often do not contain whole sentences. Treating a slide as a unit of processing might be a more reasonable approach here. In the case of biomedical paper abstracts, it might be worth considering other units, such as marked sections on materials and methods, as the units of processing.

### Tokenisation

NLP is almost always concerned with processing individual words that make up a text, so text needs to be divided up into words. Tokenisation is a relatively simple procedure for which there are good standard tools for most languages, but it is not trivial. In general, a word token is something delimited to the left and to the right by whitespace. Edge cases like the first and last word of a document are relatively trivial to handle. However, there are systematic patterns that require substantial effort to get right, such as punctuation attached to a word to the left (e.g. quotation marks, parentheses, etc.) or to the right (comma, period, colon, etc). These are clearly not part of a word token and need to be separated from it. Whether they should be retained as tokens or ignored (deleted) needs to be decided. Most or possibly all languages have word forms that cannot be neatly handled using the simple tokenisation strategy based on whitespace. For example, in English there are contracted forms such as *it’s* or *don’t* which are in a sense two word tokens combined into one, and at the very least they contain a non-alphabetic character that also functions as punctuation in other cases, so it is not trivial how to handle these. Complex word forms like compounds marked by a hyphen, such as anti-vaccination, need to be handled, either tokenised as a single word, or two, or possibly three (including the hyphen). Some contractions are not marked by punctuation, such as German *zum,* which is the contracted form of the preposition + article sequence *zu dem.* A decision needs to be made whether this is considered one token or two. Conversely, compound words in English are often spelled as two or more separate words with spaces in between, such as the compound *sentence boundary detection.* Compound words in other languages, like Hungarian or German, should generally not be spelled as separate word forms with spaces, but in practice many or perhaps most language users spell write them as separate words contrary to the language’s orthography rules. Thus although *mondathatár-felismerés* would be the only correct spelling of this compound, we can expect to encounter non-standard spellings such as *mondat határ felismerés* frequently. Another extreme case is languages that do not use whitespace between words at all, such as Chinese and other East-Asian languages. In these a completely different tokenisation strategy needs to be developed. All of these situations need to be handled one way or the other. Thus although tokenisation appears to be a simple issue, it quickly turns out to be rather non-trivial once one starts working with actual texts.

### Stemming and Lemmatization

Most languages have a morphological system due to which the same dictionary words, called lexemes or (in the context of NLP typically) lemmas, can be used in different forms in different syntactic contexts or with slightly different meanings, such as singular and plural forms of nouns or various tense forms (present, past, future, etc.) for verbs. In some languages words have very few such so-called morphological forms, like in English. In other languages, like Turkish, Hungarian or Georgian, the same lemma can have tens or thousands of slightly different forms. Many NLP techniques have been found to work best in practice if forms of a word are reduced to the word’s lemma or stem (which is the part of the word that remains after its morphological prefixes and suffixes have been removed, e.g. *adopt* is the stem of *adoption*). This process is called stemming or lemmatization and is usually carried out using existing standard morphological processing tools. Stemming is typically simpler, less fine-grained than lemmatization, and it is always either one or the other that is applied to an unprocessed text. It makes no sense to both stem and lemmatise word forms. In addition to stemming or lemmatization, a related procedure of reducing word forms to a more general lexical processing unit is normalisation. As mentioned before, orthographical mistakes may lead to forms that cannot be processed using the usual rule-based lemmatisation and stemming tools or reduce the forms to a stem or lemma different from that of the correctly spelled forms. Normalisation is the process of correcting orthographic anomalies so that texts can be processed properly using existing NLP resources such as machine-readable dictionaries. For example, without normalisation, the frequent misspelled form *pajama* is a completely different word from the correctly spelled *pyjama.* The former needs to be normalised for the NLP application to be able to know that a text talking about *pajamas* and one talking about *pyjamas* are talking about the same topic. Note that at least in the context of NLP, correct spelling is not a matter of being pedantic or a “grammar nazi” (orthography does not have anything to do with grammar for that matter) but decides whether a text can or cannot be processed properly.

### Stopword Removal

The most frequently used words in a language are so-called function words which do not contribute a lot to the meaning of a sentence. Function words contrast with so-called content words which generally express a meaning or a concept. A step that is often found to improve the performance of NLP applications in practice (similarly to the effect of stemming, lemmatisation and normalisation) is the removal of function words, which are called stopwords for some unknown reason in NLP. To find the most frequent words in a corpus of English texts, we can use the following Unix shell command:

cat \* | sed 's/ /\n/g' | sort | uniq -c | sort -r

A typical output for English starts as follows, for a corpus containing half a million word tokens:

28926 the

16058 of

13638 and

11641 a

10624 to

9265 in

5697 I

5624 his

5353 that

4720 he

4573 was

4571 with

3512 it

Just these top 13 function words add up to more than 120,000 word tokens, i.e. about one quarter of the entire volume of text in the corpus.

All of these are typical function words, which would thus be removed as part of an NLP and text mining pipeline. There reason why stopword removal tends to have a positive effect is that stopwords are never characteristic of a certain text, as they are so fundamental elements of the given language that they occur in most texts and in many sentences. Thus they contribute little to what text mining is interested in, which is to identify topics and patterns that are characteristic of specific texts. In fact, stopwords are so frequent that they tend to introduce a significant amount of noise, i.e. they distract the text mining application from the important patterns that it should be focusing on. Note, however, that for more fine-grained applications, at least certain function words can be very relevant, and removing them can have a significant negative impact on the performance of a text processing application. For example, removing negation can be very detrimental in sentiment analysis, which aims to determine whether a review or a comment expresses a positive or a negative attitude. For such an application, it is not at all irrelevant whether the text being processed contains the phrase *not very nice* or just *very nice*, and removing the negation would reduce the former to the latter.

### Further Linguistic Processing

Stemming and stopword removal aim to **simplify** the input text and **remove parts** of it. Depending on the specific implementation, lemmatisation can have the same effect, in case we simply replace the forms by their lemma. An **alternative way** to proceed in NLP and in text mining as well is **not to simplify** the material but rather to **enrich** it with **linguistic annotation**. There are several types of linguistic analysis that can be applied here, including:

* **part-of-speech tagging**, which assigns its lexical category such as noun, verb, adjective, article, preposition, etc. to each word in the text, disambiguating ambiguous words based on context, e.g. whether process appears as a noun or a verb in a phrase like *the entire process* versus *to process the data;*
* **morphological analysis**, which determines the morphological features such as number (singular vs. plural), person (1st, 2nd, 3rd), case (nominative, accusative, genitive, etc.), tense, mood (indicative, subjunctive, imperative, etc.) to each word;
* **syntactic analysis,** usually referred to as parsing, which analyses the syntactic structure of the sentence and thereby determines which word modifies, i.e. belongs to which other word, e.g. that in *my younger brother’s children,* *younger* modifies *brother,* not *children;* orwhat the subject, object, temporal modifier etc. are of a verbal predicate. Such processing can be useful or even indispensible for various analyses such as relation mining, which extracts relations between entities, such as who did what to whom, when, where, etc., from large bodies of text.
* **named entity recognition** is concerned with the identification of proper names and the ontological category of the entity that they denote (such as person, place, institution, etc.) in texts. The crucial point is that the names are not simply looked up in a list of known names, but are rather identified based on linguistic cues in a text, i.e. depending on whether these expressions appear in linguistic contexts that are characteristic of names, whether they are capitalised, etc. In biomedical text mining, named entity types to be recognised might be the names of biological species or other taxonomical categories, drugs, pharmaceutical companies, or authors being cited.

The results of these analyses do not replace or remove the original words in the text, like in the case of stemming or stopword removal, but are rather added to the text in the form of **annotation**, i.e. names denoting drugs are tagged in some form, marking them as such.

The current standard for annotations added through part-of-speech tagging, morphological analysis and syntactic analysis is **universal dependencies**, abbreviated as **UD**, which specifies both tagsets (i.e. list of allowed tags) and tagging guidelines for a very large number of languages on part-of-speech, morphological and syntactic level. UD’s name derives from an approach to syntax called dependency syntax, which is a descriptive syntactic framework that is based on a relatively intuitive assumptions about how sentences are structures, as opposed to syntactic approaches such as generative syntax, which assumes that observable sequences of words are

However, such linguistic analysis steps can serve as input to further processing steps that might be necessary. For instance, when preparing medical or legal records for mining, it might be compulsory to anonymise the input texts. Automatic **anonymisation** presupposes that the names in the texts have been recognised as such, and thus named entity recognition can be an important prerequisite for this stage of the pipeline. The type of entity is likely also be relevant here, e.g. legislation might require the data processor to remove names of persons from texts to be processed, but not names of places, companies or institutions.

It should be self-evident that these linguistic processing steps are **incompatible** with either stemming or stopword removal, since for these analyses to be carried out, the exact forms of the words need to appear in the sentences to be analysed, and function words, which play a crucial role in marking sentence structure, must not be left out. Thus if linguistic analysis is to be applied, then stopword removal and stemming need to be removed from the processing pipeline. Whether the correct approach is to simplify the input or enriching it with linguistic analysis results depends exclusively on the aim of the analysis.

## Mathematical and Statistical Properties of Natural Languages

All natural languages have some general properties that underlie the approaches to NLP and text mining, including the preprocessing techniques that we have mentioned. The most important regularity that can be observed from a quantitative perspective in natural language corpora is a phenomenon called **Zipf’s law**. Zipf's law is a principle that describes the statistical distribution of elements in certain types of datasets, particularly in natural language, but also in certain social and economic systems. It is named after the linguist George Zipf, who observed the phenomenon in the frequency of words in a language. Zipf's law states that in a dataset, the frequency of an item is inversely proportional to its rank when sorted by frequency. For words in a language, this typically means that the most frequent word (in English *the*) appears about twice as often as the second most frequent word *of,* three times as often as the third most frequent word *and,* and so on. Although the numbers are not quite exact, we can see how this works out in practice in the list of stopword frequencies in section 7.1.6.

Zipf's law is a specific example of a **power-law distribution**. Such distributions describe phenomena where there are few occurrences (in this case, words) that occur very frequently and there are many occurrences (words) that occur very rarely, specifically just once. Words that occur just once in a collection of texts (a corpus) are so important in connection with the processing of corpora that there is a technical term for them: they are called hapax legomena, which is Classical Greek for “things that were said (only) once”, **hapaxes** for short. A hapax legomenon (hapax for short) is a word form that occurs only once, as a single token, in the given corpus, i.e. in the collection of texts that are being examined. Hapax legomena make up a significant part of every natural-language corpus. In the specific case of the collection of English novels that the counts were based on, more than 10 percent of all tokens is a hapax token:

cat \* | sed 's/ /\n/g' | sort | uniq -c | grep -c ' 1 '

56938

This number is even more significant when compared to the number of total word forms (as opposed word tokens) in the same corpus:

cat \* | sed 's/ /\n/g' | sort | uniq -c | wc -l

88193

Thus two-thirds of all word forms in this corpus only occur once.

In other languages like Hungarian, the proportion of hapax tokens relative to the total number of text tokens is significantly higher and can approach 50 % of the whole corpus. This is because in Hungarian (and other languages with a rich set of morphological forms for each lemma), the total number of occurrences of a given lemma is distributed not between two or three word forms but typically between tens.

It is worth noting that the proportion of hapaxes is constant for a given language regardless of the size of the corpus. A somewhat surprising consequence of this is that it is impossible to compile a corpus that contains all words in a language. As we add further and further texts to a corpus, we keep adding more and more hapaxes.

Like stopwords, hapaxes are generally **uninteresting for text mining** unless they can be connected to a non-hapax lemma by lemmatization or stemming. This is because text mining as interested in patterns that recur across documents, and hapaxes are necessarily confined to not just a single document but even to one single point of a document and thus uninformative and irrelevant. Thus along with stopwords, hapaxes (or hapax lemmas or hapax stems, depending on the processing pipeline) need to be filtered out as well.

Apart from such general mathematical and statistical regularities, various other **numerical characteristics of texts** can be measured and could allow us to learn about characteristics of individual documents or genres of documents, such as mean length of sentences (measured in tokens or characters), mean number of punctuation symbols per sentence (an indicator of sentence complexity), type to token ratio (how many times does each word form occur on average; the higher this number, the less often words are repeated, and the more “colourful” the text is), mean length of tokens (longer words are characteristic of more complex texts and more technical language).

All of these numbers are highly dependent on language, and thus these numbers are generally only informative when documents (genres, etc.) that are written in the same language are being compared to one another.

A mathematical property of words that is particularly relevant to text mining is **term frequency - inverse document frequency**, standardly abbreviated as **TF-IDF**. This number is an indicator of how specific a given word is to a specific text, which is often called the **keyness** of a word to a text. Term frequency is the number of occurrences of a word form in a specific document, whereas document frequency is the total number of texts that the word form occurs in. TF-IDF is calculated for each word form in a document by the following formula: tf \* log(N/df), where N is the number of documents in the corpus. N/df is exactly 1 if df = N, thus the word occurs in all documents (which is probably the case for the most frequent stopwords). It is never smaller than 1 (df cannot be more than N by definition), it grows as the number of texts that the word appears in increases, and N/DF = N if df = 1, i.e. for a word that only occurs in a single document. Thus the more often a word occurs in the current text, and the less texts it occurs in, the higher TF-IDF is. The resulting number tells us how important a word is for the given text. If a word form occurs in a very large number of documents in the corpus, this decreases its TF-IDF, and indicates that the word is not characteristic of any particular document. A high document frequency, which results in a low TF-IDF, is characteristic of stopwords. A low document frequency, on the other hand, indicates that the word is typical of the document at hand. Generally words with a high TF-IDF can be considered keywords.

## Examining Words in Context

When working with text corpora, it is sometimes useful to examine in what context individual words appear, although this is not usually directly helpful for achieving the goals of text mining. A **concordance** is a **list of all contexts that a word appears in** within the corpus. Essentially when we use grep searching for a specific word in a text corpus, the output is a concordance.

grep "climbing" \* | head | grep "climbing"

pg2701.txt:woods, the last day of the year! Who’d go **climbing** after chestnuts now?

pg2701.txt:leaped from the canoe, swam to the boat; and **climbing** the gunwale,

pg4300.txt:cormorants, vultures, goshawks, **climbing** woodcocks, peregrines,

pg4300.txt:them in the intermediate imagine **climbing** over the railings if anybody

pg4300.txt:**climbing** down into the area if anybody saw him Ill knock him off that

pg64317.txt:the course of it, my underwear kept **climbing** like a damp snake around

When working with text corpora, the most typical kind of concordance is a so-called **KWIC** concordance, where KWIC is short for key word in context. The “key word” here does not have anything to do with keywords in the TF-IDF sense, it just means the word that we are currently interested in, i.e. the search term. A KWIC concordance is a concordance in which the search term is horizontally centred in the middle of the screen on every line, and we thus see its context both to the left and to the right of it with the same width on all lines.

Another concept from corpus linguistics that can be relevant for the analysis of large collections of text is that of collocation. **Collocations** are **statistically highly unexpected co-occurrences** of two or more words. The idea behind collocations is that raw co-occurrence frequency does not tell us much about how closely two words belong together. For example, *to jump* is not a collocation, the two words do not constitute a set phrase. On the other hand, jump overboard can be reasonably considered a set phrase and a collocation. This is despite the fact that to jump occurs more frequently in our texts (or probably any collection of texts) than *jump overboard.*

cat \* | grep -c "\bjump overboard\b"

3

cat \* | grep -c "\bto jump\b"

4

However, what is important is that *overboard* occurs in the corpus only 34 times, *jump* 24 times, and *to* 10508 times. Thus the reason why *to* *jump* is more frequent than *jump overboard* is simply that *to* is extremely frequent, and it appears next to all possible verbs, not specifically with *jump.* *Overboard,* however, is strongly associated with either verbs of throwing *(throw, toss, fling)* or verbs of movement *(go, jump, slide).*

Although this particular example is about linguistic characteristics of specific words and not interesting for text mining, it should be clear that the idea of collocations, i.e. mutual strong associations between words that go beyond their random co-occurrence in the same contexts that is explainable by the high frequency of either one or both of the words in question, can be informative when analysing large amounts of text and can help reveal interesting patterns that are worth investigating. For example, in a biomedical domain if we analyse and compare the collocations of e.g. the names of different drugs in a corpus, we might be able to identify patterns such as side effects or positive effects that are highly associated with the drug.

Although collocations for words are typically identified within a very narrow context window, typically of a maximum distance of three words to the left and to the right of the word for which we want to expects the collocates, this choice has mainly to do with the fact that linguistically relevant set lexical phrases normally appear in the very close vicinity of each other. However, when we are less interested in specific linguistic expressions but rather in thematic or conceptual associations between words, like in the previously mentioned possible biomedical application, it might be better to analyse the frequencies of co-occurrence in a broader context, perhaps within the same sentence or paragraph. Ultimately there are no recipes that are guaranteed to work with all texts for all questions that we are asking in text mining, but like in all areas of data science, an iterative process of trial and error is usually involved that tries to combine various practices that have been established to work well relatively frequently.

## Turning Text into Numerical Vectors

The examination of contexts in which words appear and of collocations assumes that there is a human examining and interpreting the results. This is a time-consuming process that cannot be straightforwardly automated, and is not scalable, i.e. it only works effectively for small collections of texts. However, as we have mentioned, text mining typically aims to extract information from very large document collections containing many thousands and even many millions of documents. For this to become feasible, raw text must be **transformed into a format that computational models can interpret**, a process known as vectorisation.

### Vectorisation and Representing Features as High-Dimensional Vectors

Unlike structured data such as numbers or categorical labels, **text is inherently unstructured**, consisting of sequences of words and phrases that **lack a direct mathematical representation**. **Vectorisation** bridges this gap by converting textual data into numerical vectors, enabling algorithms to process and analyse it. This transformation not only allows for the application of machine learning techniques but also captures essential properties of the text, such as word frequency, context, and semantic meaning. In this chapter, we explore the necessity of vectorisation, its fundamental role in the text mining pipeline, and the diverse methods available, ranging from the both conceptually and algorithmically simplistic, resource-light “bag of words” approach to sophisticated contextual embeddings. Understanding these techniques is crucial for working with text data across a wide range of text mining applications, from sentiment analysis to the grouping of documents through either clustering or classification.

Vectorisation is a process by which units of textual data, which may be words, phrases, sentences, paragraphs or entire documents are converted into vectors in a multidimensional space. On an abstract level, vectors are mathematical objects that are characterised by magnitude and direction. However, in the very concrete sense that we are interested in here, a vector is simply an array, or in other words, an ordered list of a certain length the elements of which are numbers. We often refer to the elements of a vector as its coordinates. The length of the list of numbers corresponds to the number of dimensions in the space they occupy.

Thus in the two- or three-dimensional spaces familiar from geometry, a vector might be defined by two or three numbers corresponding to coordinates like (x,y) or (x,y,z), respectively, which denote the end point of the vector relative to the origin. Vectors can be visualized as arrows in a plane or in three-dimensional space, and the “direction” and “length” of these arrows is strictly determined by the position of the end point of the vector, assuming that the starting point of the vector is the origin of the vector space (or in other words, the origin of the coordinate system). Any pair of positive or negative numbers specifies a point on the two-dimensional plane, and thus a vector in the plain. Similarly, any triple of numbers specifies a point in three-dimensional space. If two different vectors are defined by two points in the plane that are very close to each other, e.g. (2,2) and (2.1, 2.1), both the direction and the length of the two vectors are very similar. Conversely, if the numbers (coordinates) are negated, the vectors point into the opposite direction, e.g. (-2,-2) and (2,2). Two vectors, either in two- or in three-dimensional space, always enclose an angle. If they port in exactly the same direction, the angle between them is 0 degrees; if their direction is opposite to one another, the angle is 180°, etc. The angle enclosed by two vectors can be calculated based on their coordinates.

In text mining, words, documents, etc. are characterised by vectors of much higher dimension than two or three; typical values are hundreds up to hundreds of thousands of coordinates. However, what has been said about two- and three-dimensional vectors is true of these as well: higher-dimensional vectors also have a length, a direction, and two vectors in the same vector space always enclose an angle which characterises similar the direction is in which the two vectors point compared to one another. The only substantial difference is that multidimensional vector spaces are impossible to visualize. In such spaces, each dimension represents a distinct feature of the data. In text mining, each feature of data might typically be some characteristic of a particular word type (a lemma, a stem or a word form). For example, if the vector characterises a word form, then these elements of the vector can characterise the frequency with which the word described by the vector co-occurs with that other word in all sentences of a corpus. If the vector characterises a document, the elements of the vector might be the TF-IDF value of each word form that occurs in that document relative to the entire corpus. Such very high-dimensional vector spaces are not uncommon in other fields of data science either. In image processing, for example, features that characterise an image might be the red, green and blue values of each pixel of the image, for example. In all of these cases, the interpretation of each dimension of the vector space is relatively straightforward, but this is not necessarily the case. Since extremely high-dimensional vectors are computationally difficult (slow and memory-intensive) to work with, the dimensionality of these vector spaces is often reduced using standard dimensionality-reduction techniques familiar from linear algebra. While this makes the resulting lower-dimensional versions of the vectors much more efficient to process, the downside is that this transformation results in a total loss of interpretability of the features.

Despite being intangible and abstract, multidimensional spaces (whether interpretable or not) enable important mathematical operations that have practical uses in the text mining process: Vectors in such spaces can be compared for similarity using techniques like cosine similarity, or clustered based on their relative positions.

### Bag of Words and TF-IDF Vectorisation

Before we turn to more useful but much more complex text vectorisation methods, we will review in this section two very basic, foundational methods for vectorising text: the “Bag of Words” (BoW) model and Term Frequency-Inverse Document Frequency (TF-IDF) vectorisation. Both approaches are widely used for representing textual data as structured numerical data and form the basis for many text mining and NLP tasks, mainly owing to their simplicity.

**Bag of words** represents text as a collection of words, disregarding grammar, order, and context, and focusing solely on the frequency of words within a document. Each document is transformed into a vector where each element corresponds to the count of a particular word from the corpus vocabulary.

The first step in implementing BoW is to **build the vocabulary** from the corpus to be processed. All different word forms (i.e. unique words) from the corpus are collected to form the vocabulary. Note that before this processing is carried out, the preprocessing steps mentioned earlier (stemming, stopword removal, etc.) are typically executed, since BoW omits a lot of information contained in the corpus anyway.

Assuming a very small corpus containing the sentences "I love dogs.", "Dogs are loyal.", and "I love loyal friends like dogs.", the vocabulary would consist of the items "I", "love", "dogs", "are", "loyal", "friends", and "like", assuming that lemmatization and stopword removal were skipped; and alternatively "love", "dog", "loyal", "friend" and "like" including these preprocessing actions.

After the vocabulary has been defined, a so-called **term-document matrix** (TDM) is created based on the corpus. A term-document matrix is a mathematical representation of a corpus in which rows correspond to terms (typically words, but these could also be phrases or n-grams, e.g. bigrams), columns correspond to documents, and the **entries in the matrix represent the occurrence or weight of a term in a document**.

Continuing the above example of creating a BoW matrix based on three sentences, we get the following, where each cell contains the frequency of a word in a document:

*I love dogs are loyal friends like*

*Doc1* 1 1 1 0 0 0 0

*Doc2* 0 0 1 1 1 0 0

*Doc3* 1 1 1 0 1 1 1

As seen in this example, the term **document** does not necessarily refer to a standalone, long-form text like a book or an article in the context of term-document matrices. Instead, it refers to a unit of text analysis that can vary depending on the task or application. A document could be a **paragraph** from a report; a **single sentence or tweet** in a social media dataset; a **product review** in a dataset for sentiment analysis; an entire **book** in a large corpus of literature, etc. However, the definition of what a document is **should be kept consistent** within the same term-document matrix, i.e. it is not correct to mix various types of document in the same matrix, so that one column represents a sentence, another a paragraph, and a third a whole book. This is because the term-document matrix is typically used for comparisons, and textual units of very different sizes can hardly be compared to each other in a sensible way.

The **granularity** of what constitutes a document is determined by the goals of the analysis. For example, in a search engine, documents might be entire web pages to match against queries. In sentiment analysis, a document is typically a customer review which expresses a sentiment as a whole. If the goal is question answering based on a corpus of scientific texts, the correct level is probably either sentence or paragraph, since the answer to a question is typically a single sentence or a paragraph-length sequence of sentences, but not e.g. a whole section of a research paper, or even a paper as a whole.

It is important to recognise that a term-document matrix consists of two types of vectors: The horizontal rows of the matrix are document vectors, whereas the vertical columns are term vectors. Documents with similar contents will typically have similar document vectors, and words with related meanings are also likely to have similar word vectors. Note all word vectors have the same dimensionality (number of elements), and also all document vectors have the same number of elements. Thus the TDM defines a document vector space and a term vector space. Vectors across these vector spaces cannot be compared to each other, i.e. it makes no sense to compare a term vector to a document vector, even if the dimensionality of the two vector spaces would be accidentally identical.

Once the BoW matrix has been created, the document or term vectors that it contains can be used to compare and thus to either cluster or classify documents (which is typically what text mining is concerned with), or terms. How this is done will be briefly outlined in section 7.5.

The BoW approach to text vectorisation has the advantage that it is **very simple**, **straightforward to implement** and understand. Both the term and the document vectors are **trivial to interpret**, as each feature in a document vector directly corresponds to a word in the vocabulary, and each feature in a term vector corresponds to a document in the corpus. However, it also has clear drawbacks: The dimensionality of the vectors is very high, as both the size of the vocabulary and the number of documents can become enormous for large corpora, especially if we are working with “documents” that are sentence-sized. In addition, the document-term matrix is sparse, meaning that most cells in it are zero, making it inefficient to work with or at least requiring the application of special solutions for the storage and handling of sparse matrices / vectors. Furthermore, as mentioned earlier, BoW ignores the order and syntactic relationship between words, potentially missing semantic meaning.

Instead of building up the TDM based on raw frequency counts (i.e. the number of times the given term appears in the given document), an alternative is to weight the words using the TF-IDF value, as defined in section 7.2. In this case, the score recorded in each cell of the matrix indicates the importance of the term in the document relative to the corpus. TF-IDF assigns higher scores to words that are frequent in a particular document but rare across the corpus, thus prioritizing terms that distinguish documents (or, more interestingly, groups of related documents) from each other.

Text mining processes based on TF-IDF matrices tend to yield better results in practice than ones based on BoW. However, they involve the same problems as BoW matrices (size, sparseness, ignoring context). In addition, if further documents are added to a corpus, BoW vectors do not change (at most only further 0-valued columns are added to them for words not seen in the original corpus), whereas TF-IDF scores can change and need to be recalculated.

### Dense Vector Spaces Characterising the Distributions of Words

As text mining evolved, traditional methods like Bag of Words (BoW) and TF-IDF revealed limitations, particularly their inability to capture semantic relationships and contextual meanings. Dense vector representations emerged as a more sophisticated approach to overcome these problems. These methods convert text — whether individual words or entire documents — into **dense, relatively low-dimensional vectors** compared to the previously mentioned term and document vectors. Typically the dimensionality of these vectors is in the **low hundreds** (a usual choice is 300 dimensions), which is much less than either the number of documents or the number of words (forms or lemmas) in a large corpus that is typically of interest for text mining. **Dense** means that there are no zero elements in these vectors; all elements are being used, and all have a real number as their value.

These dense vector representations, commonly known as **word, sentence, documents, etc. embeddings**, are numerical encodings of text that primarily capture the so-called **distribution of words** in the corpus, i.e. **in the context of which other words each word occurs** often, and which words each word does not occur with. It was an extremely important and hugely influential result in natural language processing in the mid-2010s when researchers working for Facebook observed that dense word vectors which were calculated to encode the distribution of words were found to encapsulate word meaning as well: two words that appear in the context of a similar set of other words in the corpus, i.e. two words that have a similar distribution in the corpus, have a similar meaning, according to native speakers’ judgments of semantic similarity. This idea, according to which the meaning of a **word can be characterised by** or is in some sense even determined by **which other words it is used together with**, is called **distributional semantics**, and has been around since the middle of the 20th century, although it has never been a very prominent theory of natural language semantics. It was also a central idea of an approach to text mining and information retrieval that was popular in the 1990s and the early 2000s called **latent semantic analysis** or **latent semantic indexing**, which went out of fashion later since it could not be scaled to very large sets or documents because of the mathematical complexity of this approach. However, the semantic results achieved with distributional word vectors in the mid-2010s were widely regarded as an impressive confirmation that distributional semantics captures at least a very important aspect of natural language meaning correctly.

Without going into technical and mathematical details, the most important and most widely used software implementations of dense distributional word and document vectors, which are called **word2vec** and **fastText**, are based on the following general approach. First, a vocabulary for the entire corpus is compiled, similarly to BoW. Then a separate random vector of the specified dimensionality, typically 300 dimensions, is generated for every single word in the vocabulary, so that every item in the vocabulary has its own vector which will eventually represent its distribution, and thereby its meaning, but initially is just a random vector. As we have seen above, such a vector designates a specific point in the 300-dimensional vector space.

After this initialisation, the entire corpus is processed by sliding a context window of a certain width, usually 5 words to the left and 5 words to the right around a central word, over the entire corpus, from the beginning of the first document to the end of the last document, moving this sliding window to the right by one word on each step. Thus if the corpus contains 1 million word tokens, this process takes 1 million steps to finish, and in each step, we look at the context around a single word token, first the context around the very first word in the corpus, then the context around the second token, then the third, and so on, up to the millionth and last token. On each such step, we modify some word vectors as follows: For each such context window, the vector belonging to the vocabulary item of the central word (e.g. the vector of the vocabulary item *today* if the central word token has the form *today*) is moved a little bit closer to the vector of each of the 10 words that appear in its context, and the vectors of the latter are moved a little bit closer to the vector of the central word. In addition, 10 words from the vocabulary are randomly selected, which do not appear in this context window around the central word. These words are referred to as a negative sample taken from the vocabulary. The vector of the central word is moved a little bit away from each of the vectors in the negative sample, and these vectors in turn are moved a little bit away from it.

After all words in the corpus have been processed in this way, the word vectors converge to a configuration such that words that appear either together in the same contexts, or in similar context to each other, have vectors that are similar, i.e. are situated in a similar region of the vector space, whereas words that occur in completely different contexts are situated in regions that are far from each other.

While word2vec and fastText uses such an iterative process to generate word vectors, there is a third widely known approach called GloVe that creates a word co-occurrence matrix as a first step and then uses matrix factorisation to derive the word vectors from this. This approach is more efficient but requires far larger computational resources, in particular a very large amount of memory during the calculation of the word vectors.

Thus in addition to having far fewer dimensions than vectors in traditional term-document matrices, such word embeddings capture the position and usage of words within sentences, i.e. the vector representation of each word is influenced by the representation of surrounding words. Furthermore, vectors for similar words (e.g. "king" and "queen") or related concepts (e.g. "dog" and "pet") are located close to each other in the vector space. In addition, it has also been established that similarity of word vectors is correlated with word analogies, part-of-speech and morphological relationships.

Apart from these advantages over TDMs, word embeddings have the clean drawback, from a text mining perspective, that they characterise words, not documents, and it is not self-evident how good document vectors can be created based on them. Fortunately, in practice there are straightforward imperfect solutions for this problem that work well enough, as we will see shortly. Another disadvantage is that, like dimensionality-reduced TDMs, the features in word embeddings do not have any intuitive interpretation. While it can be clearly and consistently observed that e.g. the word vectors of country names or number words or names of professions or nouns derived with the suffix -ness cluster together in the vector space, or that the vectors of female counterparts of male words (e.g. king vs. queen, man vs. woman, boy vs. girl, etc.) are relatively shifted in the same direction as compared to the male word, there is no single feature among the 300 vectors, or any combination of a small number of coordinates, which could be identified as encoding a female feature or the meaning of the suffix -ness, or being a number. Instead, each of these semantic factors are spread over all 300 vectors. For this reason, word embeddings are often referred to as distributed semantic representations, which means that the vector representations encode meanings, but each meaning component is distributed across all dimensions of the vector space.

Although software like word2vec, fastText and GloVe can be downloaded from the internet and used to train a word vector space model from scratch on one’s own corpus, pre-trained vector dictionaries for several languages are available for download online and can be used directly without having to invest time and effort in training.

However, since vector space models of words in a corpus represent a special kind of summary of the content of the corpus, an approach to text mining that compares several subcorpora within the corpus of interest through their word vector space models is one possible embedding-based approach to text mining. For example, one could collect social media documents (e.g. tweets, comments, blog posts) from different time periods, e.g. 2010 to 2015, 2015 to 2020, 2020 to 2025, compile a corpus for each of these periods, train a word vector space model on each, and the examine the vector spaces with regard to how associative relationships between various words (and thus concepts) have evolved over time, e.g. in connection with concepts related to vaccination.

Dense vectorisation techniques can **extend beyond words** to phrases, sentences, paragraphs, and entire documents. This can be attempted in different ways. One approach to generating document-level vectors is to combine the dense vectors of individual words within the document. Techniques for aggregation include taking the **average of all word vectors** in the document, or calculating a **weighted average**, assigning weights to word vectors **based on their importance** (e.g., TF-IDF weights). While straightforward, these methods obviously ignore word order and distances between the words that make up the document, i.e. they lead to the loss of a substantial amount of information.

### Generating Document Embeddings Using Neural Networks

More recently invented and more advanced techniques than the word embeddings discussed in the previous subsection involve generating embeddings directly at the sentence or document level using neural language models. These models consider the entire text unit during encoding, enabling them to capture rich semantic relationships, sentence structure, and context.

It is no exaggeration to say that modern language models based on the neural network architecture called transformer, like BERT, GPT and RoBERTa, have truly revolutionized how text is represented and processed for computational tasks since the end of the 2010s. These models generate document embeddings by leveraging deep learning, i.e. very complex neural networks consisting of hundreds of millions up to trillions of trainable parameters, and trained on massive amounts of linguistic data, containing at least billions, or in the case of larger models even trillions of word tokens. The embeddings generated by such immense language models encode not only the individual meanings of words but also their contextual relationships within sentences, paragraphs, or entire documents.

The process of building such an embedding begins with a tokeniser software specifically created for the language model reading in the text and tokenising it, breaking it into smaller units such as words or subwords. Each such word or subword token is mapped to an initial embedding, a numerical vector that serves as a starting point. The language model then processes these embeddings through multiple layers of transformations in a complex neural network, which essentially consist of matrix multiplications, additions and concatenations. During this process, the model captures intricate linguistic patterns, such as syntactic structures, word dependencies, and even the broader semantic meaning of the entire document.

For models like BERT and RoBERTa, which are bidirectional, this involves analyzing the text in both directions, examining the context both to the left and to the right of each word, to understand how each word relates to its context. By contrast, generative models like GPT focus on predicting the next word in a sequence, but they still encode semantic information over longer stretches of the prior context. Once the text has been fully processed, the model outputs a dense vector representation. This vector can either correspond to individual tokens (for fine-grained analysis) or be aggregated into a single embedding that represents the document as a whole.

Neural language model-based word and document embeddings have substantial advantages over either term-document matrix or word2vec-style word embedding representations. Unlike these earlier methods, neural language models encode meaning dynamically based on context. For instance, the word *bank* in *river bank* will have a different embedding than *bank* in the sense of a financial institution, as the model considers the surrounding words. Contrary to the relatively simplistic document embeddings that can be calculated by averaging word2vec word embeddings, neural document embeddings capture not only the meanings of individual words but also their relationships and the overarching themes of the document. This makes them highly effective for complex tasks like document classification, similarity detection, and clustering. Finally, language models are invariably pre-trained on massive and diverse corpora, and therefore have encode very deep and extensive knowledge about the pattern of the language or languages that they have been trained on. Experience has shown that this general language knowledge can be fine-tuned for specific tasks with minimal additional data.

However, these substantial advantages do not come without costs. Generating embeddings with large models like BERT or especially large generative models like GPT requires significant computational resources, both in terms of hardware (e.g., GPUs) and time. This tends to make them impractical for large-scale applications like text mining, where we want to process huge numbers of texts. Secondly, as with distributional word embeddings, their interpretability is limited. The dimensions of embeddings are abstract, they do not generally represent anything that is interpretable in terms of human concepts. Thirdly, the quality of the embeddings depend heavily on the data the model was pre-trained on. For example, if the pre-training corpus lacks representation of certain domains, the embeddings generated by the model will be of relatively little use. This is in fact true of all NLP models, not just neural language models, but it is important to remember that a model that has been shown to work excellently on e.g. general question answering tasks will not necessarily perform nearly as well e.g. on some natural language understanding task as applied to a scientific text from the biomedical domain.

Despite these limitations, they remain a cornerstone of modern NLP, enabling sophisticated insights from text data that were previously unattainable.

## Using Vector Representations of Texts for Classification and Clustering

Regardless of the way they are created, document vectors form the foundation of numerous downstream applications, in part related to text mining. By representing documents in a shared vector space, their relative positions can be analysed to identify similar texts or **group them** into clusters. Dense vectors enable machine learning models to **classify** documents efficiently into predefined classes, e.g. for sentiment analysis (whether the document expresses a positive or a negative attitude to e.g. a product), topic modeling (characterising the topic of a document, i.e. what is it about), or spam detection. All of these are relevant to text mining if they are applied at a large scale to large corpora to extract summary information or trends, although primarily they clearly also have uses on the level of a single text.

Apart from text mining, a widely used application of document vectors is related to search engines. Here, a query entered by a user is first vectorised (e.g. using a TDM or by adding up the embeddings of the words that appear in the query), and then the query is matched to documents indexed by the search engine by comparing each document’s vector representations to that of the query. Using word embeddings, documents can be returned as a search hit for a query even if the exact words that appear in the query do not appear in a document, but it contains synonyms or paraphrases of terms in the query. Similarly, Document embeddings can also be used in recommendation systems to suggest relevant content (e.g., articles, products) based on user preferences encoded in similar vectors.