4 Text Processing

Abstract

In the previous chapter we were able to automatically process structured data to retrieve biomedical text about any chemical compound, such as *caffeine*. This chapter will provide a step-by-step introduction to how we can process that text using shell script commands, specifically extract information about diseases related to *caffeine*. The goal is to equip the reader with an essential set of skills to extract meaningful information from any text.

Keywords

NLP: Natural Language Processing · Text mining · Pattern matching · String matching · Word matching · Evaluation metrics · Regular expressions · Tokenization · NER: Named-Entity Recognition · Relation extraction

In the previous chapter we were able to automatically process structured data to retrieve biomedical text about any chemical compound, such as *caffeine*. This chapter will provide a step-by-step introduction to how we can process that text using shell script commands, specifically extract information about diseases related to *caffeine*. The goal is to equip the reader with an essential set of skills to extract meaningful information from any text.

Pattern Matching

We used the `grep` command in the last chapter to find a disease in the text, since `grep` receives as argument a pattern to find an exact match in the text, like any search functionality provided by conventional text editors. However, we may need to search for multiple patterns even when interested in a single disease. For example, when searching for mentions of *malignant hyperthermia*, we may also be interested in finding mentions using related expressions, such as:

MH	– acronym
MHS	– acronym for *malignant hyperthermia susceptible*

Since we already know how to deal with multiple patterns by using the `-e` option, we may easily solve this problem by executing:

```
$ grep -e 'malignant
     hyperthermia' -e 'MH' -e '
     MHS' chebi_27732.txt
```

Case Insensitive Matching

When dealing with text, using a case sensitive search is usually a good approach to avoid wrong matches. For example, acronyms are normally in upper case, while the full name is usually in lowercase having sometimes the first letter of

F. M. Couto, *Data and Text Processing for Health and Life Sciences*,
Advances in Experimental Medicine and Biology 1137,
https://doi.org/10.1007/978-3-030-13845-5_4

each word (or only the first word) in uppercase. So, instead of using a full case sensitive `grep`, we might think on performing a case sensitive `grep` for the acronyms and a case insensitive `grep` for the disease words using the `-i` option:

```
$ grep -e 'MH' -e 'MHS'
     chebi_27732.txt
$ grep -i -e 'malignant
     hyperthermia' chebi_27732.
     txt
```

The equivalent long form to the `-i` option is `--ignore-case`. We should note that each execution of `grep` will produce two separate lists of matching lines that might be overlapped.

Alternatively, we can also convert it to just one case sensitive `grep`, if we are sure that *Malignant hyperthermia* is the only alternative case to *malignant hyperthermia* present in the text. So, we can add it as another pattern:

```
$ grep -e 'Malignant
     hyperthermia' -e '
     malignant hyperthermia'
  -e 'MH' -e 'MHS' chebi_27732.
     txt
```

Number of Matches

To be sure that we are not losing any match, we can count the number of matching lines for both cases. First we execute a case insensitive `grep` and then we execute a case sensitive `grep`, both using the `-c` option:

```
$ grep -c -i 'malignant
     hyperthermia' chebi_27732.
     txt
$ grep -c -e 'malignant
     hyperthermia' -e '
     Malignant hyperthermia'
     chebi_27732.txt
```

The equivalent long form to the `-c` option is `--count`.

In our case, the output should show 96 and 95 matching lines for the insensitive and sensitive patterns, respectively.

This means that there is a line that is not caught by the case sensitive pattern. To identify which one is, we can manually analyze each of the 96 matching lines one by one. But the goal of this book is exactly avoiding these type of tedious tasks. One thing we can do to solve this issue is to find from the case insensitive matches the one that do not match the case sensitive patterns.

Invert Match

Fortunately, the `grep` command has the `-v` option that inverts the matching and returns the lines of text that do not contain any matching. The equivalent long form to the `-v` option is `--invert-match`.

Thus, if we apply the inverted match with the case sensitive patterns to the output given by the case insensitive matching, we will get our outlier mention:

```
$ grep -i 'malignant
     hyperthermia' chebi_27732.
     txt | grep -v -e '
     Malignant hyperthermia' -e
      'malignant hyperthermia'
```

From the output, we can easily identify the missing matching line:

```
...gene are associated with
   Malignant Hyperthermia (MH)
   and...
```

We were missing the case where both words have the first letter in uppercase.

Thus, to obtain all the matching lines in a case sensitive match we just have to include the missing match as another pattern:

```
$ grep -c -e 'malignant
     hyperthermia' -e '
     Malignant hyperthermia' -e
      'Malignant Hyperthermia'
     chebi_27732.txt
```

File Differences

Another alternative to compare different matches, is to use the `diff` command that receives as input two files and identifies their differences. So, we can create two auxiliary files and then apply the `diff` to them:

```
$ grep -i 'malignant
     hyperthermia'
    chebi_27732.txt >
    insensitive.txt
$ grep -e 'Malignant
    hyperthermia'
    -e 'malignant hyperthermia'
    chebi_27732.txt > sensitive
       .txt
$ diff sensitive.txt insensitive
     .txt
```

The output should be the same text.

A problem that may occur with case sensitive matching is that some acronyms are defined with lowercase letters in the middle, such as ChEBI, and humans are not consistent with the way they mention them. The same acronym may be mentioned in their original form or with all letters in uppercase, or just some of them. Moreover, these inconsistent mentions sometimes may even be found in the same publication. We hope not in this book ! ⌣̈

Evaluation Metrics

These inconsistencies made by humans when mentioning case sensitive expressions, is one of the reasons that most online search engines use case insensitive searches as default. This type of approach favors recall, while case sensitive search favor precision[1].

Recall is the proportion of the number of correct matches found by our tool over the total number of correct mentions in the texts (found or not found). Case insensitive searches avoid missing mentions, so they favor recall.

Precision is the proportion of the number of correct matches found by our tool over the total number of matches found (correct or incorrect). Case sensitive searches avoid incorrect matches, so they favor precision.

[1]https://en.wikipedia.org/wiki/Precision_and_recall

Normally, there is a trade-off between precision and recall. Using a technique that improves precision, most of the times, will decrease recall, and vice-versa. To know how good the trade-off is, we can use the F-measure, which is the harmonic average of the precision and recall[2].

Word Matching

Acronyms (or terms) may also appear inside common words or longer acronyms. For example, when searching for *MH*, the word *victi**mh**ood* will produce a match:

```
$ echo "victimhood" | grep -i '
     MH'
```

The problem with *victi**mh**ood* could be easily solved by using case sensitive matching, but not for a longer acronym. For example, the acronym NEDMHM for *neurodevelopmental disorder with midbrain and hindbrain malformations* will produce a case sensitive match:

```
$ echo "NEDMHM" | grep 'MH'
```

One way to address this problem is to use the `-w` option of `grep` to only match entire words, i.e. the match must be preceded and followed by characters that are not letters, digits, or an underscore (or be at the beginning or end of the line). The equivalent long form to the `-w` option is `--word-regexp`.

Using this option, neither *victimhood* or *NEDMHM* will produce a match:

```
$ echo "victimhood" | grep -w -i
     'MH'
$ echo "NEDMHM" | grep -w -i 'MH'
```

Word matching improves precision but decreases recall, since we may miss some less common acronyms that we are not aware of, but are still relevant for our study. For example, consider that we may also be interested in the following acronyms:

[2]https://en.wikipedia.org/wiki/F1_score

MHE – acronym for *malignant hyperthermia equivocal*
MHN – acronym for *malignant hyperthermia normal*

If we apply word matching, we will not get a match, since both exact matches are followed by a letter:

```
$ echo "MHE and MHN" | grep -w -
    i 'MH'
```

These are not trivial problems to solve by exact pattern matching, we may need regular expressions to address some of these issues more efficiently.

Regular Expressions

When dealing with natural language text we may need more flexibility than the one provided by exact matching. Regular expressions are an efficient tool to extend exact matching with flexible patterns, that may find different matches. As an example, we may be interested in finding all the mentions of the acronym MHS or MHN in a text. For doing that, regular expressions provide the alternation operator that helps us to solve this issue easily by specifying multiple alternatives to match in a specific part of the pattern, in this case an *S* or an *N* as the last character.

Regular expressions can be better understood by clearly separating three distinct components:

input – any string where we want to find something
pattern – a string that specifies what we are looking for
match – a fragment of the input (a substring) where the pattern can be found

In our examples, the input is the text file *chebi_27732.txt*, but it can be the amino acid sequences that we previously extracted from the UniProt file entries. Until now the pattern has represented an exact string to look for, where each match is an exact replica of the pattern occurring at a given position of the input string. When using regular expressions, the pattern contains special characters, whose purpose are not to directly match with the input but instead have a special meaning. These special characters represent operators that specify which different types of strings we want to find in the input. For example, strings that start with *MH* and end with *S* or an *N*. By using regular expressions, the matches are not replicas of the pattern, they can be different strings as long as they satisfy the specified pattern.

Extended Syntax

The `grep` command allows us the possibility to include regular expression operators in the input pattern. `grep` understands two different versions of regular expression syntax: basic and extended[3]. We will use the extended syntax for two reasons: (i) the basic does not support relevant operators, such as alternation; (ii) and to clearly differentiate exact matching from regular expression matching. Thus, instead of the `-e` option previously used in the `grep` command, we will start to use the `-E` option, which makes the command interpret the pattern as an extended regular expression. The equivalent long form to the `-E` option is `--extended-regexp`. We should note that this option does not affects the matching when using a pattern without any regular expression operator, such as `MH`. For example, the following commands will produce the same results:

```
$   echo -e 'MHS\nMHN' | grep -e
        'MH'
$   echo -e 'MHS\nMHN' | grep -E
        'MH'
```

Note, that we use the `-e` option so the `echo` command interpret the `\n` characters as a newline. Thus, the `echo` command outputs two lines, that are given as input to the `grep` command. We should note that the `grep` command filters lines.

[3] https://www.regular-expressions.info/posix.html

Alternation

The first regular expression operator we will test is the alternation, which we introduced above. An alternation is represented by the bar character (|) that specifies a pattern where any match must include either the preceding or following characters. The preceding and following characters can be enclosed within parentheses to better specify the scope of the alternation operator. For example, the pattern for finding strings that start with *MH* and end with *S* or an *N* can be written as:

```
$ echo -e 'MHS\nMHN' | grep -E
      'MH(S|N)'
```

Basic Syntax

If we use the basic regular expression syntax no match will be found, since the alternation operator is not supported:

```
$ echo -e 'MHS\nMHN' | grep -e
      'MH(S|N)'
```

We will have a match only if the | and the parentheses are in the input string, since it is not interpreted as an operator:

```
$ echo -e 'MH(S|N)' | grep -e
      'MH(S|N)'
```

Scope

To better understand the scope of an alternation, we can remove the parentheses from the pattern and add the -w option:

```
$ echo -e 'MHS\nMHN' | grep -w
      -E 'MHS|N'
```

We only get the first line. This is explained because the alternation operator is applied to all the preceding characters, i.e. the `grep` will search for the *MHS* word or the *N* word. If we add a single *N* to the input string we already get another match:

```
$ echo -e 'MHS\nN' | grep -w -E
      'MHS|N'
```

We can also move the opening parenthesis one character to the left:

```
$ echo -e 'MHS\nMHN' | grep -E
      'M(HS|N)'
```

Only *MHS* is now displayed, since the alternative now represents *MN* without the *H*.

Multiple Alternatives

We are not limited to two alternatives, we can have multiple | operators in a pattern. For example, the following command will find any of the three acronyms *MHS*, *MHE* or *MHN*:

```
$ echo -e 'MHS\nMHN\nMHE' | grep
      -E 'MH(S|N|E)'
```

We can now transform our previous `grep` command with multiple case sensitive patterns:

```
$ grep -c -e 'Malignant
     hyperthermia' -e '
     Malignant Hyperthermia' -e
      'malignant hyperthermia'
     chebi_27732.txt
```

in a `grep` command with a single pattern using alternation:

```
$ grep -c -E '(M|m)alignant(H|h)
     yperthermia' chebi_27732.
     txt
```

And we will obtain the same 96 matches.

Multiple Characters

A useful regular expression feature is that we can use the dot character (.) to represent any character, so if we want to find all the acronyms that start with *MH* we can execute the following command:

```
$ grep -o -w -E 'MH.'
     chebi_27732.txt | sort -u
```

We should note that we use the `-o` option of the command `grep` so it just displays the matches and not all the line that includes the match. The equivalent long form to the `-o` option is `--only-matching`.

The output will be the following three-character lines:

```
MH
MH)
MH,
MH.
MH1
MH2
MHE
MHN
MHS
```

If we really want to match only the dot character, we have to precede it with a backslash character (\):

```
$ grep -o -w -E 'MH\.'
     chebi_27732.txt | sort -u
```

Now only the *MH.* will be displayed.

We can check that there are some matches that are not really acronyms, such as *MH)* and *MH,*.

Spaces

We should note that *MH* appears because the space character can also be matched. For example, the following text includes a word match with *MH␣* since the parenthesis is considered a word delimiter character (not a letter, digit or underscore):

```
... susceptible to MH (MHS) ...
```

On the other hand, the following text does not include a word match with *MH␣*:

```
... markers and MH
    susceptibility ...
```

Thus, what we really want is matches where the third character is a letter or a numerical digit.

Sometimes, the text includes other characters that also represent horizontal or vertical space in typography, such as the tab character. All these characters are known as whitespaces and can be represented by the expression `\s` in a pattern[4]. The following command demonstrates that both the space and the tab characters are matched by `\s`:

```
echo -e 'space: :\ntab:\t:' |
    grep -E '\s'
```

[4]https://en.wikipedia.org/wiki/Whitespace_character

Groups

Fortunately, the regular expressions include the group operator that let us easily specify a set of characters. A group operator is represented by a set of characters enclosed within square brackets. Any of the enclosed characters can be matched.

For example, the previous command to find any of the three acronyms can be replaced by:

```
$ echo -e 'MHS\nMHN\nMHE' | grep
       -E 'MH[SNE]'
```

We should note that only one of the three letters, *S*, *N* or *E* will be matched in the input string.

Ranges

Still, this is not solving our need to only match letters or digit. However, we can also specify characters ranges with the dash character (-). For example, to find all the acronyms that start with *MH* followed by any alphabet letter:

```
$ grep -o -w -E 'MH[A-Z]'
     chebi_27732.txt | sort -u
```

This will result in only three acronyms:

```
MHE
MHN
MHS
```

We should note that `A-Z` represents any alphabet letter in uppercase, a lowercase letter will not be matched:

```
$ echo -e 'MHS\nMHs' | grep -E '
     MH[A-Z]'
```

If we intend to keep the usage of a case sensitive `grep` and at the same time find lowercase matches, then we need to add the `a-z` range:

```
$ echo -e 'MHS\nMHs' | grep -E '
     MH[A-Za-z]'
```

We should note that the dot character inside a range represents itself and not any character:

```
$ echo -e 'MHS\nMH.' | grep -E '
     MH[.]'
```

Additionally, to include the acronyms that end with a numerical digit we need to add the `0-9` range:

```
$ grep -o -w -E 'MH[A-Z0-9]'
     chebi_27732.txt | sort -u
```

Finally, we have the correct list of all three character acronyms starting with *MH*:

```
MH1
MH2
MHE
MHN
MHS
```

Negation

Another frequent case is the need to match any character with a few exceptions. For example, if we need to find all the matches that start with *MH* followed by any character except an alphabet letter. Fortunately, we can use the negation feature within a group operator. The negation feature is represented by the circumflex character (^) right next to the left bracket. The negation means that all the characters and ranges enclosed within the brackets are the ones that cannot be matched. Thus, a solution to the above example is to add the `A-Z` range after the circumflex:

```
$ grep -o -w -E 'MH[^A-Z]'
     chebi_27732.txt | sort -u
```

We can see that all of the three acronyms *MHS*, *MHE* or *MHN* will be missing from the output:

```
MH
MH,
MH.
MH)
MH1
MH2
```

If we do not want the *MH*␣ acronym, we can add the space character to the negative group:

```
$ grep -o -w -E 'MH[^A-Z ]'
     chebi_27732.txt | sort -u
```

The output should now contain one less acronym:

```
MH,
MH.
MH)
MH1
MH2
```

Quantifiers

Above we were interested in finding acronyms composed of exactly three characters. However, we may need to find all acronyms that start with *MH* independently of their length. This functionality is also available in regular expressions using the quantifiers operators.

Optional

The simplest quantifier is the optional operator that is specified by an item followed by the question mark character (?). The item can be a character, an operator or a sub-pattern enclosed by parentheses. That item becomes optional for matching, i.e. a match can either contain that item or not.

For example, to find all the acronyms starting with *MH* and followed by one alphabetic letter or none:

```
$ grep -o -w -E 'MH[A-Z0-9]?'
     chebi_27732.txt | sort -u
```

Given that the third character is optional the output will include the two-character acronym *MH*, but not the *MH*␣ match:

```
MH
MH1
MH2
MHE
MHN
MHS
```

We can add the space character to the group:

```
$ grep -o -w -E 'MH[A-Z0-9 ]?'
     chebi_27732.txt | sort -u
```

Now the output includes the two-character acronym *MH* and the *MH*␣ match:

```
MH
MH
MH1
MH2
MHE
MHN
MHS
```

Multiple and Optional

To find all the acronyms independently of their length, we can use the asterisk character (*). The preceding item becomes optional and can be repeated multiple times. For example, to find all the acronyms starting with *MH* and which may be followed any number of alphabetic letters or numeric digits:

```
$ grep -o -w -E 'MH[A-Z0-9]*'
     chebi_27732.txt | sort -u
```

The output now includes the four-character acronym *MHS1*:

```
MH
MH1
MH2
MHE
MHN
MHS
MHS1
```

We should note that the `grep` command uses a greedy approach, i.e. it will try to match as many characters as possible. For example, the following command will match *MH1* and not *MH*:

```
$ echo 'MH1' | grep -o -E 'MH
     [0-9]*'
```

Multiple and Compulsory

To make the preceding item compulsory and able to repeat it multiple times, we may replace the asterisk by the plus character (+). For example, the following pattern will find all the acronyms starting with *MH* followed by at least one alphabetic letter or numeric digit:

```
$ grep -o -w -E 'MH[A-Z0-9]+'
     chebi_27732.txt | sort -u
```

We should note that the output does not contain the two character acronym *MH*:

```
MH1
MH2
MHE
MHN
MHS
MHS1
```

All Options

The above quantifiers are the most popular, but the functionality of all of them can be reproduced by using curly braces to specify the minimal and maximum number of occurrences. The item is followed by an expression of the type `{n,m}` where `n` and `m` are to be replaced by a number specifying the minimum and maximum number of occurrences, respectively. `n` and `m` may also be omitted, which means that no minimum or maximum limit is to be imposed.

Using curly brackets, the question mark character (?) can be replaced by `{0,1}`. Thus, the following two patterns are equivalent:

```
$ grep -o -w -E 'MH[A-Z0-9]?'
     chebi_27732.txt | sort -u
$ grep -o -w -E 'MH[A-Z0
     -9]{0,1}' chebi_27732.txt
     | sort -u
```

The asterisk character (*) can be replaced by `{0,}`. Thus, the following two patterns are equivalent:

```
$ grep -o -w -E 'MH[A-Z0-9]*'
     chebi_27732.txt | sort -u
$ grep -o -w -E 'MH[A-Z0-9]{0,}'
      chebi_27732.txt | sort -u
```

The plus character (+) can be replaced by `{1,}`. Thus, the following two patterns are equivalent:

```
$ grep -o -w -E 'MH[A-Z0-9]+'
     chebi_27732.txt | sort -u
$ grep -o -w -E 'MH[A-Z0-9]{1,}'
      chebi_27732.txt | sort -u
```

On the other hand using `{1,1}` is the same as not having any operator. Thus, the following two patterns are equivalent:

```
$ grep -o -w -E 'MH[A-Z0-9]'
     chebi_27732.txt | sort -u
$ grep -o -w -E 'MH[A-Z0
     -9]{1,1}' chebi_27732.txt
     | sort -u
```

The previous commands display the all the three-character acronyms:

```
MH1
MH2
MHE
MHN
MHS
```

For example, if we are looking for acronyms with exactly 4 characters then we can apply the following pattern:

```
$ grep -o -w -E 'MH[A-Z0
     -9]{2,2}' chebi_27732.txt
     | sort -u
```

We should note that we use 2 as both the minimum and maximum since *MH* already count as 2 characters.

The output of the previous command is now the four-character acronym:

```
MHS1
```

Position

Sometimes besides the match, we are also interested in limiting the matches to specific parts of the input string. For example, to identify start and stop codons in a protein sequence, we need to limit the matches to the beginning or the end of the sequence. In text, we may for example be interested in lines starting with a name of a disease. To take in account the position of a match regular expressions patterns can start with the circumflex character (^) and/or end with the dollar sign character ($).

If the pattern starts with a circumflex then only matches at the beginning of the line will be considered. On the other hand, if the pattern ends with a dollar then only matches at the end of the line will be considered.

Beginning

For example, if we are looking for lines starting with *Malignant Hyperthermia* we can use the following pattern:

```
$ grep -E '^(M|m)alignant (H|h)
     yperthermia' chebi_27732.
     txt
```

The output will include the list of lines beginning with a mention to *Malignant Hyperthermia*:

```
...
Malignant hyperthermia (MH) is a
    potentially fatal autosomal
  ...
Malignant hyperthermia (MH) is a
    pharmacogenetic disorder ...
```

To check how many of the matching lines were filtered, we can count the number of occurrences when using the circumflex and when not:

```
$ grep -c -E'^(M|m)alignant(H|h)
     yperthermia' chebi_27732.
     txt
$ grep -c -E'(M|m)alignant(H|h)
     yperthermia' chebi_27732.
     txt
```

The output will show that only 23 of the 96 matches were considered.

Ending

If we are looking for lines ending with a mention to *Malignant Hyperthermia*, then we can add the dollar character to the end of the pattern:

```
$ grep -E '(M|m)alignant (H|h)
     yperthermia.$' chebi_27732
     .txt
```

To allow a punctuation character before the end of the line, we added the dot character before the dollar character in the pattern. The dot character matches any character, including the dot itself.

The output will be the list of lines ending with a mention to *Malignant Hyperthermia*:

```
Novel mutation in the RYR1 gene
   (R2454C) in a patient with
   malignant hyperthermia.
```

```
Identification of a novel
    mutation in the ryanodine
    receptor gene (RYR1) in
    patients with malignant
    hyperthermia.
Novel skeletal muscle ryanodine
    receptor mutation in a large
    Brazilian family with
    malignant hyperthermia.
...
```

We can check how many lines were filtered by using again the `-c` option:

```
$ grep -c -E '(M|m)alignant(H|h)
    yperthermia.$' chebi_27732
    .txt
```

```
$ grep -c -E '(M|m)alignant(H|h)
    yperthermia' chebi_27732.
    txt
```

The output will show that only 15 of the 96 matches were at the end of the line.

Near the End

Sometimes we do not want the mention ending exactly at the last character. We may be more flexible and allow a following expression, or a given number of characters. For example, to allow 10 other characters between the end of the line and the mention of *Malignant Hyperthermia*, we can add a quantifier to the dot operator:

```
$ grep -c -E '(M|m)alignant (H|h
    )yperthermia.{0,10}$'
    chebi_27732.txt
```

The output will show that we have 20 matches.

If we remove the `-c` option, we will be able to check that words, such as *families* and *patients*, are now allowed to appear between the mention of *Malignant Hyperthermia* and the end of the line:

```
...
Novel mutations in C-terminal
    channel region of the
    ryanodine receptor in
    malignant hyperthermia
    patients.
...
Novel missense mutations and
    unexpected multiple changes
    of RYR1 gene in 75 malignant
    hyperthermia families.
...
```

Word in Between

To allow a word in between, independently of its length, we can add to the pattern an optional sequence of non-space characters (the word) preceded by a space:

```
$ grep -c -E '(M|m)alignant(H|h)
    yperthermia( [^ ]*)?.$'
    chebi_27732.txt
```

The output will show that we have 24 matches. We should note that the `[^ ]` operator avoids having two words.

If we remove the `-c` option, we will be able to check that lengthy words (with more than 10 characters), such as *susceptibility*, are now allowed to appear between the mention of *Malignant Hyperthermia* and the end of the line:

```
...
Ryanodine receptor gene point
    mutation and malignant
    hyperthermia susceptibility.
...
```

Full Line

If we want lines that start with a mention to *Malignant Hyperthermia* and end with an acronym, *MH* or *MHS*, then we can execute two `grep` commands. The first gets the lines starting with *Malignant Hyperthermia* and the next filters the output of the latter with lines ending with an acronym:

```
$ grep -E '^(M|m)alignant (H|h)
    yperthermia' chebi_27732.
    txt | grep -w -E 'MHS?.$'
```

Alternatively, we can add both the circumflex and dollar operators to the same pattern. However, we cannot forget to add `.*` to match anything in between them, since we are asking full line matches:

```
$ grep -w -E'^(M|m)alignant(H|h)
     yperthermia.*MHS?.$'
     chebi_27732.txt
```

We can see that both commands match all the text of the abstract since each abstract is stored in a single line of the file:

```
Malignant hyperthermia (MH) is a
   pharmacogenetical
   complication ... as for

   genetic diagnosis of MH.
Malignant hyperthermia
   susceptibility (MHS) is a
   subclinical pharmacogenetic
   disorder ... been tested
   positive for MHS.
```

This demonstrates the problem of tokenization, since usually what we really need is to match a full sentence or a phrase. And in that case each line should represent a sentence or phrase from the abstract.

Match Position

For more advanced processing, we may be interested in knowing the exact position of the matches in a given line. This can be done by using the `-b` option of `grep`, which provides the number of bytes in the line before the start of the match:

```
$ echo 'MHS MHN MHE' | grep -b -
     o -w -E 'MH[SNE]'
```

The equivalent long form to the `-b` option is `--byte-offset`.

The output shows the list of matches preceded by their position in the given line:

```
0:MHS
4:MHN
8:MHE
```

Tokenization

As we have shown in the previous section, sometimes we need to work at the level of a sentence and not use a full document as the input string. Tokenization is a Natural Language Processing (NLP) task that aims at identifying boundaries in the text to fragment it into basic units called tokens. These tokens can be sentences, phrases, multi-word expressions, or words.

Character Delimiters

In most languages, some specific characters can be considered as accurate boundaries to fragment text into tokens. For example, the space character to identify words; the period (.), the question mark (?) and the exclamation mark (!) to identify the ending of a sentence; and the comma (,), the semicolon (;), the colon (:) or any kind of parenthesis to identify a phrase within a sentence. However, this problem may be more complex in languages without explicitly delimiters, such as Chinese (Wu and Fung 1994).

A common approach to tokenization is to use regular expressions to replace these delimiters by newline characters. This will result in a token per line. For example, we can replace the characters specifying the end of a sentence with a newline by using the `tr` command and then count the number of lines:

```
$ tr '[.!?]' '\n' < chebi_27732.
     txt | wc -l
```

We get 1493 lines from the original 248 lines:

```
$ wc -l chebi_27732.txt
```

Unfortunately, this is not just so simple. We need to analyze the output:

```
$ tr '[.!?]' '\n' < chebi_27732.
     txt | less
```

Wrong Tokens

We can check that: (i) many lines are empty because an extra newline character will be added to the last sentence, and (ii) the dot character is also used as a decimal mark in a number, then some sentences are split in multiple lines because they have decimal number in them. For example, the original sentence:

```
These 10 mutations account for
    21.9% of the North American
    MH-susceptible population
```

is split in two lines:

```
These 10 mutations account for
    21
9% of the North American MH-
    susceptible population
```

String Replacement

This means that looking at just one character is not enough, we need some context. For performing this, we will use the `sed` command that we may consider as a more powerful version of the `tr` command. The `sed` command is a stream editor that can receive as input a string and perform basic text transformations, such as replace one expression by another, that are available in almost all text editors. For example, we can use a simple `sed` to convert every mention of *caffeine* by its ChEBI identifier:

```
$ sed -E 's/caffeine/CHEBI
      :27732/gi' chebi_27732.txt
```

The `-E` option allow us to use extended regular expressions, like we used before in `grep`. The `s` option has the following syntax `'s/FIND/REPLACE/FLAGS'`, where: `FIND` is the pattern to find in the input string; `REPLACE` the expression to replace the matches; `FLAGS` are multiple options, such as `g` to replace all matches in each line and not just the first one, and `i` to be case insensitive.

For example, the original fragment of text:

```
... link between the caffeine
    threshold and tension ...
```

will be converted to:

```
... link between the CHEBI:27732
    threshold and tension ...
```

Multi-character Delimiters

To replace the delimiter characters by a newline when followed by at least one space character, we can use the following command:

```
$ sed -E 's/[.!?] +/\n/g'
      chebi_27732.txt
```

We should note that by making compulsory a space character, we avoid: (i) empty lines by splitting a sentence that is already at the end of the line (assuming there are no ghost space characters at the end of each line), and (ii) decimal markers because they are followed by numerical digits and not spaces.

We now get 1067 lines from the original 248 lines:

```
$ sed -E 's/[.!?] +/\n/g'
      chebi_27732.txt | wc -l
```

Keep Delimiters

The previous `sed` command is removing the delimiter characters from the text, and this may cause other problems. The best solution is to keep the delimiter characters and just add the newline. The `sed` command allows us to keep each match for a specific part of the pattern (sub-pattern) by enclosing it within parentheses. To include the match of a sub-pattern in the replace expression, we can use the backslash and its numerical order. Thus, we can improve our `sed` command by using this technique so we do not remove any delimiter character:

```
$ sed -E 's/([.!?])( +)/\1\n\2/g
      ' chebi_27732.txt
```

However, other common issues may still persist. For instance, there are some sentences starting right after the delimiter characters without any space in between:

```
... bulk.Fetal ...
... sequencing.Whole ...
```

These sentences include a delimiter character directly followed by an alphabetic letter:

```
$ sed -E 's/([.!?])( +)/\1\n\2/g
    ' chebi_27732.txt | grep -
    i '[.!?][a-z]'
```

To minimize this issue, we can change the pattern so the compulsory space character become optional, but requiring a following uppercase alphabetic letter:

```
$ sed -E 's/([.!?])( *[A-Z])/\1\
    n\2/g' chebi_27732.txt |
    wc -l
```

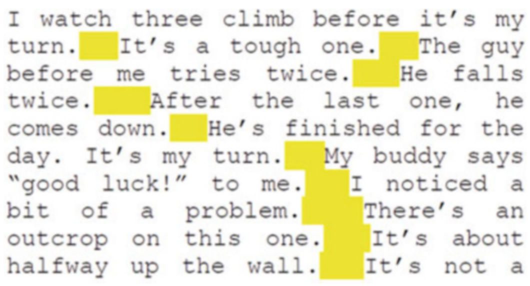
We now get 1127 lines, i.e. this pattern is more flexible and was able to split more 60 sentences. This does not mean that is free of errors. It is almost impossible to derive a rule that covers all the possible typos humans can produce.

As an example, Fig. 4.1 show a complex pattern adapted from Wikipedia. The pattern is equivalent to `\. {2,}[A-Z]`, and identifies multiples spaces at the beginning of a sentence. The pattern requires at least two spaces to be matched, but only after a period and before an uppercase letter.

Sentences File

Using our previous pattern, we can update our script named *gettext.sh* to provide the text already split in sentences by adding the `sed` command:

```
ID=$1 # The CHEBI identifier
    given as input is renamed
    to ID
grep -e '<title>' -e '<rdfs:
    comment>' chebi\_$ID\_*.
    rdf | \
```



```
I watch three climb before it's my
turn.  It's a tough one.  The guy
before me tries twice.  He falls
twice.  After the last one, he
comes down.  He's finished for the
day. It's my turn.  My buddy says
"good luck!" to me.  I noticed a
bit of a problem.  There's an
outcrop on this one.  It's about
halfway up the wall.  It's not a
```

Fig. 4.1 Identifying multiple spaces at the beginning of a sentence using regular expressions (Adapted from: https://en.wikipedia.org/wiki/Regular_expression)

```
gawk -F'[<>]' '{ print $3 }' |
    \
sed -E 's/([.!?])( *[A-Z])/\1\
    n\2/g'
```

To save the output as a file named *chebi_27732_sentences.txt*, we only need to add the redirection operator:

```
$ ./gettext.sh 27732 >
    chebi_27732_sentences.txt
```

Each line of the file *chebi_27732_sentences.txt* represents a sentence.

Entity Recognition

To select the sentences with one of our acronyms, we can use the `grep` command and our sentences file:

```
$ grep -w -E 'MH[SNE]?'
    chebi_27732_sentences.txt
```

The output will only include matching sentences:

```
...
Interestingly, the data suggest
    a link between the caffeine
    threshold and tension values
    and the MH/CCD phenotype.
```

Alternatively, we can use the `-n` option to get the number of the line and the `-o` option to get the acronym matched:

```
$ grep -n -o -w -E 'MH[SNE]?'
      chebi_27732_sentences.txt
```

The equivalent long form to the -n option is --line-number. The output should be something like this:

```
...
1106:MH
1106:MH
1108:MH
1110:MH
1111:MH
```

We can now make a script that receives a pattern as argument and the input text as the standard input, to display the line numbers and the matches in a TSV format. Thus, let us create a script file named *getentities.sh* with the following lines:

```
PATTERN=$1
grep -n -o -w -E $PATTERN | \
tr ':' '\t'
```

Again we should not forget to save the file in our working directory, and add the right permissions with chmod, as we did with our scripts in the previous chapter.

The first line stores the pattern given as argument in the variable PATTERN. The grep command finds the matches and the tr command replaces each colon by a tab character to produce TSV content.

We can now execute the script giving the pattern as argument and the sentences file as standard input:

```
$ ./getentities.sh 'MH[SNE]?' <
      chebi_27732_sentences.txt
```

The output should be something like this:

```
...
1106   MH
1106   MH
1108   MH
1110   MH
1111   MH
```

We should note that now we have the values separated by a tab character, i.e. the output is in TSV format.

The output can also be saved as a TSV file that we can open directly in our preferred spreadsheet application. For example, to save it as *chebi_27732.tsv*, we only need to add the redirection operator:

```
$ ./getentities.sh 'MH[SNE]?' <
      chebi_27732_sentences.txt
      > chebi_27732.tsv
```

Select the Sentence

If we want to analyze a specific matched sentence, we can use a text editor and go to that line number. A more efficient alternative is to use the print p option of sed to output a given line number. For example, to check the *MHS* match at line 2:

```
$ sed -n '2p'
      chebi_27732_sentences.txt
```

Now we can easily check the context of the match:

```
... in susceptible people (MHS)
   by volatile ...
```

Pattern File

The script created in the previous section only accepts one pattern, however we may need to recognize different entities, or different mentions of the same entity, such as the official name, possible synonyms, and the acronyms. Fortunately, grep allows us to include a list of patterns directly from a file using the -f option. The equivalent long form to the -f option is --file=FILE. For example, we can create a text file named *patterns.txt* with the following three patterns:

```
(M|m)alignant (H|h)yperthermia
MH[SNE]?
(C|c)affeine
```

Then we can execute the previous grep but using multiple patterns specified in the pattern file:

```
$ grep -n -o -w -E -f patterns.
      txt chebi_27732_sentences.
      txt
```

Analyzing the output, we can check that the same sentences may include different entities:

```
...
1110:MH
1110:caffeine
1111:caffeine
1111:MH
```

We can now update our script named *getentities.sh* to receive as input not a single pattern but the filename where multiple patterns can be found.

```
PATTERNS=$1
grep -n -o -w -E -f $PATTERNS
    | \
tr ':' '\t'
```

We can execute the script giving as argument the file containing the patterns:

```
$ ./getentities.sh patterns.txt
      < chebi_27732_sentences.
      txt
```

To save the output as a file named *chebi_27732.tsv*, we only need to add the redirection operator:

```
$ ./getentities.sh patterns.txt
      < chebi_27732_sentences.
      txt > chebi_27732.tsv
```

Using the *patterns.txt* file is very useful if for example we are not focused in a single disease, and we want to find any disease mentioned in the text. In these cases, we have to create a file with the full lexicon of diseases. This topic will be addressed in the following chapter.

Relation Extraction

Finding the relevant entities in text is sometimes not enough. We need to know which sentences may describe possible relationships between those entities, such as a relation between a disease and a compound.

This is a complex text mining challenge, but a simple approach is to construct a pattern that allow any kind of characters between two entities:

```
$ grep -n -w -E 'MH[SNE]?.*(C|c)
      affeine'
      chebi_27732_sentences.txt
```

The following sentence is one of the seven displayed sentences mentioning a possible relation:

```
239: ... MHS families were
   investigated with a caffeine
   ...
```

However, we are missing all the sentences that have *caffeine* first:

```
$ grep -n -w -E '(C|c)affeine.*
      MH[SNE]?'
      chebi_27732_sentences.txt
```

We will be able to see that sometimes *caffeine* comes first:

```
801: ... caffeine-halothane
   contracture test were greater
    in those who had a known MH
   ...
1111: ... caffeine threshold and
    tension values and the MH
   ...
```

Multiple Filters

The most flexible approach is use two `grep` commands. The first selects the sentences mentioning one of the entities, and the other selects from the previously selected sentences the ones mentioning the other entity. For example, we can first search for the acronyms and then for *caffeine*:

```
$ grep -n -w -E 'MH[SNE]?'
      chebi_27732_sentences.txt
      | grep -w -E '(C|c)affeine
      '
```

This will show all the nine sentences mentioning *caffeine* and an acronym.

Relation Type

If we are interested in a specific type of relationship, we may have an additional filter for a specific verb. For example, we can add a filter for sentences with the verb *response* or *diagnosed*:

```
$ grep -n -w -E 'MH[SNE]?'
     chebi_27732_sentences.txt
     | grep -w -E '(C|c)affeine
     ' | grep -w -E 'response|
     diagnosed'
```

We should note that this does not take in account where the verb appears in the sentence. For example, in the following sentence the verb *response* appears first than any of the two entities:

```
50: The relationship between the
     IVCT response and genotype
    was ... the number of MHS
    discordants ... at 2.0\,mM
    caffeine ...
```

If the verb needs to appear between the two entities, we have to construct a pattern that have these words in the middle of them:

```
$ grep -n -w -E 'MH[SNE]?.*(
     response|diagnosed).*(C|c)
     affeine'
     chebi_27732_sentences.txt
```

We can see now that the previous sentence (line 50) is not presented as a match.

Remove Relation Types

We may also be interested in ignoring specific type of relations. To do that, we only need to use the `-v` (or `--invert-match`) option. For example, to ignore sentences with the word *response* or *diagnosed*:

```
$ grep -n -w -E 'MH[SNE]?'
     chebi_27732_sentences.txt
     | grep -w -E '(C|c)affeine
     ' | grep -v -w -E '
     response|diagnosed'
```

All the resulting sentences do not mention *response* or *diagnosed*.

Further Reading

If we want to have a deeper knowledge about text processing tasks and challenges, we may be interested in reading some chapters of the book entitled *Speech and language processing* (Jurafsky and Martin 2014). The book is a highly specialized document explaining in full detail the topics here briefly described.

To have an overview about the state-of-art in text processing tools using biomedical literature, we should consider reading a recent and comprehensive survey (Lamurias and Couto 2019).