Chapter 1

Text Data and Regular Expressions

1.1 Introduction

Although binary files allow us to explicitly encode the type of a value, be it an integer, real number, string, etc., much of the data we deal with are given to us as plain text. We input numbers in text files, download text files from Web and FTP servers, and save spreadsheets as comma-separated values in .csv files. In these cases, the data are merely represented by their text form and are easily interpreted by applications. However, there are many examples of more complex situations where the data are not as easily interpreted, and the text must be processed to create the values of interest. A simple example of this phenomenon is when numeric values are embedded into text, but not in a regular or simple format, such as numbers in an HTML table. In this case, we must extract the elements of interest from the text content by identifying the patterns where the values occur. A different sort of example occurs when text itself makes up the data, such as a speech, an abstract, or an email message. Then we must search for the presence of certain words or phrases in particular contexts or places to uncover structure in the data, e.g. we might examine how often each word is used, the names of the author(s), the use of punctuation, etc. Finally, documents and text are sometimes treated directly as data such as in search engines, databases, and so on.

1.1.1 Placing data on a map

To make a county map of the United States (Figure **??** in Chapter **??**) that displays election results, and possibly census data too, requires the various sources of information to be merged together. This merger uses county name, a text field, which needs to be *transformed* into a uniform format across the three sources. The following sample lines of text demonstrate the inconsistencies in how a county's name is represented in these three sources of data; geographic (top), census (middle) and election (bottom). Notice that there is no period after `''St''` in the geographic data; the election results differ from the other two sources in that there is an `&` rather than "and" in Lewis and Clark County; the capitalization is not consistent (e.g. "Qui" vs "qui" in Lac qui Parle County); and the use of "County" and "Parish" is not consistent.

```
"De Witt County",IL,40169623,-88904690
"Lac qui Parle County",MN,45000955,-96175301
"Lewis and Clark County",MT,47113693,-112377040
"St John the Baptist Parish",LA,30118238,-90501892

"St. John the Baptist Parish","43,044","52.6","44.8",...
"De Witt County","16,798","97.8","0.5", ...
```

```
"Lac qui Parle County","8,067","98.8","0.2", ...
"Lewis and Clark County","55,716","95.2","0.2", ...

DeWitt  23      23      4,920   2,836   0
Lac Qui Parle   31      31      2,093   2,390   36
Lewis & Clark   54      54      16,432  12,655  386
St. John the Baptist    35      35      9,039   10,305  74
```

1.1.2 Spam filtering

In Chapter **??** we explore the problem of filtering email to try to differentiate legitimate electronic mail from unsolicited bulk email, i.e. spam. One approach considered *creates* several variables pertaining to the email message and then uses the values of these variables for a mail message to predict whether it is spam or not. A potentially useful variable is one that indicates whether the subject line of the email message contains a "word" with punctuation or a digit in the middle of it, such as "V!agra" or "m0rtgage".

Below are parts of the header from three email messages. The bottom two headers are from spam while the top is not.

```
Date: Tue, 02 Jan 2007 12:17:45 -0800
From: Duncan Temple Lang <duncan@wald.ucdavis.edu>
To: Deborah Nolan <nolan@stat.Berkeley.EDU>
Subject: Re: 90 days

Date: Sat, 27 Jan 2007 16:28:48 +0800
From: remade SSE <glzmeqrxr99@embarqhsd.net>
To: depchairs03-04@uclink.berkeley.edu
Subject: [SPAM:XXXXXXXXX]

Date: Thu, 03 Apr 2008 09:24:53 +0700
From: Faustino Britt <Faustino@sfera.umk.pl>
To: Brice Frederick <nolan@stat.Berkeley.EDU>
Subject: Fancy rep1!c@ted watches
```

Notice that the last subject line contains the "word" `rep1!c@ted` rather than "replicated". These fake words are popular in spam because they are easy for us to read, but they are not likely to be found in a list of "banned" words, i.e. words that a spammer would use. Another variable that may be helpful in detecting spam is a logical that indicates whether or not the subject line of an email message begins with "Re:" Yet another is whether or not the reply-to address contains an underscore or digit.

1.1.3 Weblogs

To analyze the requests made to a Web site, we first *extract* the relevant information from the Web log, such as the machine of requester, the time and date of the request, the name of the file requested, the return status, and the number of bytes returned. Below are two lines from a Web log. The log is a text file where each request appears on a separate line of text, and although the text has a lot of structure, the information does not appear in a simple format such as in comma separated values, nor is it placed consistently in the same columns in the file. For example, the date and time are set off in square brackets. Note that each line in the Web log is broken across four lines here for formatting purposes.

```
169.237.46.168 - - [26/Jan/2004:10:47:58 -0800]
  "GET /stat141/Winter04 HTTP/1.1" 301 328
  "http://anson.ucdavis.edu/courses/"
  "Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0; .NET CLR 1.1.4322)"
169.237.46.168 - - [26/Jan/2004:10:47:58 -0800]
```

Examples of Uses of Regular Expressions on Text Data

- EXTRACT pieces of text that appear in non-standard formats.
- CREATE variables from information found in text.
- CLEAN and TRANSFORM text into a uniform format and resolve inconsistencies in format between files.
- MINE text by treating documents directly as data.

```
"GET /stat141/Winter04/ HTTP/1.1" 200 2585
"http://anson.ucdavis.edu/courses/"
"Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0; .NET CLR 1.1.4322)"
```

1.1.4 Mining the State of the Union addresses

We *mine* the text of the State of the Union Addresses, looking for similarities between presidents' speeches. One approach to doing this would be to compare word frequencies across documents in the corpus of speeches. To do this, we build a word-vector for each speech that tallies the number of occurrences of each word used in the speech, e.g. there was one occurrence of the word "much", the word "debt" was used twice, and the words "nation", "national" or "nations" appeared five times in the December, 1790 inaugural speech of George Washington's (a snippet is shown below). With these word-vectors we can look for similarities between the distribution of words in the speeches. to create the word-vector, we first stem words (i.e. reduce "running" to "run") and remove stop words such as "and', "the", and "of".

```
***

State of the Union Address
George Washington
December 8, 1790

Fellow-Citizens of the Senate and House of Representatives:

In meeting you again I feel much satisfaction in being able
to repeat my congratulations on the favorable prospects which
continue to distinguish our public affairs. The abundant fruits
of another year have blessed our country with plenty and with
the means of a flourishing commerce.
```

The regular expression language is essentially a programming language, and with it we can develop complicated constructs, making use of the rich set of meta characters that it employs, to analyze each of these data sets. The goal of this chapter is to demonstrate the fundamental ideas behind the language, to introduce the basic elements in the language, and show how they can be combined to express patterns. There are many tutorials and examples on the Web that cover different uses and applications of regular expressions. There are also books on the subject that provide many more examples and a greater understanding of how regular expressions work. Most important of all, practice in creating regular expressions and testing them on data is essential to gaining both understanding and experience so that when you need to use regular expressions in handling data, they will be familiar.

1.2 Matching literal strings

The inconsistencies in the various sources of information for making the election map (Section 1.1.1) are easily remedied. For example, "County" or "Parish" can be removed from the end of each county name. Fixing the problem with the missing period in the names such as "St John the Baptist Parish" is slightly more subtle. Many counties in the United States have "St" in their names and we want to make sure that we change all occurrences of "St" to "St." However, we also want to be sure not to change, a name such as, "Stone County" to "St.one County". If we search for the pattern `'St '` and change it to `'St. '`, then that should avoid this problem.

Rather than edit data files manually, it is better to make the changes programmatically in order to keep a record of the changes required in case mistakes are made or the process needs to be repeated when the data change. Using any general programming language such as Matlab, Java, C, Perl, etc., we could develop a function to perform this simple task for operating on strings and patterns within them.

```
> string
[1] "St John the Baptist Parish"
> if ("St " == substring(string, 1, 3))
+     newString = paste("St. ",
+            substring(string, 4, nchar(string)), sep ="")
> newString
[1] "St. John the Baptist Parish"
```

If we are not sure that the pattern will occur at the beginning of the string, then we need a more general approach. Below we split the input string into a vector of single characters, and iterate over these characters looking for the particular string. That is, determine which of these characters are possible starting points of the pattern, i.e. `S`.

```
> characters = unlist(strsplit(string, "") )
> characters
 [1] "S" "t" " " "J" "o" "h" "n" " " "t" "h"
[11] "e" " " "B" "a" "p" "t" "i" "s" "t" " "
[21] "P" "a" "r" "i" "s" "h"
> possible = which(characters == "S")
> substring(string, possible[1], possible[1] + 2)
[1] "St "
```

1.2.1 Fundamental Approach

We can write a more general R function that would determine if an input text *string* contained the argument *pattern*.

```
findPattern = function(pattern, string) {
  lets = strsplit(string,"")
  firstLetter  = substring(pattern, 1, 1)
  possibles = which(lets[[1]] == firstLetter)
  if (length(possibles) > 0)
       any(pattern == substring(string, possibles,
             possibles + nchar(pattern) -1))
  else return(FALSE)
}
```

This function matches a *literal* string given by *pattern* within the given *string* by searching for all the occurrences of the first character in *pattern* and then looking at all substrings with the same length as *pattern* starting from those points. It illustrates the fundamental approach to pattern matching, e.g. when we look for the literal string `St ` in a line of text. When we write down a regular expression pattern like `'St '`,

```
            The Slippery St Frances.
                ||       |||
                ||       |||
Found S ________||       |||
Followed by t?__| No     |||
Is it S? _______| No ...||| Keep looking for an S
                         |||
Found S _________________|||
Followed by t? ___________|| Yes
Followed by blank? _______| Yes  - A Match!
```

Figure 1.1: A diagram of a search for the literal string ‘St ’. The regular expression matching engine looks for the first character ‘S’, immediately followed by t’, immediately followed by blank. When it finds the ‘S’ in Slippery, it then checks if the next character is a ‘t’. Since it is not, it starts over and checks to see if it is an ‘S’, and continues looking for an ‘S’. The most basic building block in a language that supports matching patterns in text is a facility for specifying a short string of text in a literal string as a pattern to match.

what is meant is really the following: find the character `S` immediately (i.e. the next character) followed by `t`, immediately followed by a blank character. What the regular expression matching engine does is, for the target string, start at the first character and check to see if it is an `S`. If not, then move to the second character and start looking there. When it finds an `S`, it then checks if the next character is a `t`. If not a `t`, then start over and check if this character is an `S` and so on. So we can think of the literal string as being made up of three consecutive sub-patterns: `S`, `t` and ‘ ’. Thinking of the pattern `'St '` in this way makes it easier to see how to combine different types of complex patterns to define a sequence (see Figure 1.1 for an example of this search process).

There are two functions in R, *gsub()* and *sub()*, that look for the pattern and replace it within a string with some other text. Each of these functions takes three arguments: the regular expression (pattern) defining what to match, another regular expression to use as the replacement text, and the string(s) on which to do the matching and substitution.

The ‘g’ in the name *gsub()* refers to *global*. This means that it changes all the matches of the regular expression in the text with the replacement pattern. The *sub()* is almost exactly the same as *gsub()* except that it only replaces the first occurrence of the pattern with the replacement text. In our example here, we expect there to be one occurrence of “St ” in the string, and so *sub()* should work fine.

```
> countyNames
[1] "Dewitt County"              "Lac qui Parle County"
[3] "St John the Baptist Parish" "Stone County"
> gsub("St ", "St. ", countyNames)
[1] "Dewitt County"               "Lac qui Parle County"
[3] "St. John the Baptist Parish" "Stone County"
```

As an illustration of how *gsub()* differs from *sub()*, we replace the word (or literal string, actually) “one” with the digit “1” in the following simple character vector.

```
> strings =  c("a test", "and one and one is two",
        "one two three")
> gsub("one", "1",strings)
[1] "a test"  "and 1 and 1 is two"  "1 two three"
```

Notice, there was no “one” in the first string (“a test”), so there was no way to substitute the match with the replacement text (“1”). So it remains unaltered and is returned as is. In the second string, there are two occurrences of the string “one”. Each of these are replaced with the digit “1”. And similarly, the third string has its single occurrence of “one” replaced with “1”. In contrast, the *sub()* replaces only the first occurrence of “one” with “1” in the second string.

```
> sub("one", "1", strings)
[1] "a test"   "and 1 and one is two"  "1 two three"
```

The language of regular expressions is far more powerful than illustrated by this simple example. The next examples build on this pattern matching technique to demonstrate the more advanced features of the regular expression language.

1.3 Character Classes

Searching the subject line for “Re:” in an email message (Section 1.1.2) is a task similar to the pattern search in the previous example; the simple function *findPattern()* should easily handle the job. The other patterns present more of a challenge. For example, if we are looking for email addresses that contain digits, then any digit 0 through 9 found anywhere in the address would be considered a match. The function *findPattern()* cannot handle this more complicated pattern, as it is only capable of performing a simple literal string comparison. Here we want to ask about alternative patterns, i.e. of the form this or that. In this case, we could split the string into separate characters and check whether any of these are digits. Similarly, to find a fake word that contains punctuation in the middle of it, we could split the subject line into individual characters, search for punctuation or a digit, and if we find it, then look at the preceding character and the succeeding character to ascertain whether they are letters of the alphabet (upper or lower case). So any letter–punctuation or digit–letter combination is a match.

We can write a suite of functions to perform the different types of matching and substitutions required of this problem. Since the search for any digit or any alpha character are commonly needed, it is reasonable to guess that others might have already implemented these for their purposes and we might be able to reuse their code. As with all software, we would like to reuse such code as it is likely to be better tested and more efficient than our initial efforts. Ideally, we would be able to use a language to express these patterns in a unified manner. Indeed, the regular expression language is such a general language that has several implementations that are well-tested and efficient.

It provides basic building blocks for specifying patterns that are to be matched in a piece of text. These allow us to match literal strings, a character from a particular set of characters or its complement (character sets), and one sub-pattern or another (alternation). We can also match by position such as at the beginning or end of a line, and we can create sub-patterns from individual patterns by specifying the number of times the pattern should be matched (quantifiers).

1.3.1 Equivalent Characters

For the search in an email address for a digit, the pattern we want to match could contain any digit: 0, 1, 2, 3, 4, 5, 6, 7, 8 or 9. Regular expressions allow us to succinctly express the concept of “match a digit”, by explicitly enumerating the equivalent characters. We use the `[ ]` notation to identify a collection of equivalent characters, called a *character class*, that specify the collection of characters that constitute a match. For example, to match a digit, we us `[0123456789]`. Similarly to match a lower case letter, we use

```
[abcdefghijklmnopqrstuvwxyz]
```

and to match a space or a TAB character, we use `[ \t]`.

Name	Collection of characters
[[:alnum:]]	All alphabetic and numeric
[[:alpha:]]	All alphabetic
[[:lower:]]	Lower case alphabetic characters
[[:upper:]]	Upper case alphabetic characters
[[:digit:]]	Digits 0123456789
[[:punct:]]	Punctuation characters
[[:blank:]]	Blank characters, i.e. space or tab
[[:space:]]	White space
[[:cntrl:]]	Control characters, e.g. new line
[[:print:]]	Printable characters
[[:graph:]]	Printable character except space

Table 1.1: Some useful named character classes.

Basically, we can enumerate any collection of characters within the [] and these are included in the set of characters that constitutes a match. We also adapt this notation very slightly to indicate a match on the complement of the set of characters. That is, we place a caret $\wedge$ as the first character within `[]` to indicate that the equivalent characters are the complement of the characters enumerated within, i.e. anything but these characters is considered a match.

There are many collections of characters that are commonly used. For example, we often want to specify all the letters of the alphabet, lower or upper case or both. And we often want all the digits. And in other cases, we want a subset of these sets. The $-$ character when used within the character class pattern (i.e. the `[]`) typically identifies a range. We can specify the digits 0 through 9 more readily as `[0-9]` and the subset of the digits 3, 4, 5, 6 can be specified as `[3-6]`. Similarly, we can specify `[0-9A-F]` to match all the hexadecimal digits. And you will often see `[A-Za-z]` for all letters (upper and lower case) in the alphabet. Once again, the pattern expresses a higher level concept that is easier to read than explicitly enumerating the elements of the character set.

Note, if we want to include the character $-$ in our set of characters to match, then we must put this at the beginning of the character set, otherwise it is interpreted as a range. For example, to match the basic arithmetic operators $+$, $-$, $*$ or $/$, we can use `[-+*/]` Or to match a digit with either a $+$ or $-$ in front of it, we can use `[-+][0-9]` That is, we have made an overall pattern from a *sequence* of two sub-patterns and each of these sub-patterns is made up using the primitive elements. The first - is for the literal character and the second is for the special character that denotes a range.

1.3.2 Named Character Classes

Character classes are very convenient, and the range operator ($-$) succinctly specifies collections of characters to further simplify their use. The regular expression language also provides a collection of built-in character sets for commonly used collections. Each of these is identified by a short name. See Table 1.1 for a description of some of these named character sets.

We use these collections with the same `[]` notation, but we specify the named character set with an additional `[::]` pair. So, for example, to specify the punctuation characters, use `[[:punct:]]` The `[:punct:]` term is the named character class. Additional characters can be included in the overall set such as `[[:digit:]_]` to consider a match any digit *or* the _ underscore character.

The search for a digit or an underscore in the email can now be easily performed.

```
> Addresses
[1] "Duncan Temple Lang <duncan@wald.ucdavis.edu>"
[2] "depchairs03-04@uclink.berkeley.edu"
[3] "Faustino Britt <Faustino@sfera.umk.pl>"
> grep("[[:digit:]_]", Addresses)
[1] 2
```

The *grep()* function in R takes two arguments, the regular expression specifying the overall pattern to match and then a character vector containing the different text strings on which to search. It returns the indices of the elements of that character vector for which there was a match (or the empty integer vector if none matched). This can be readily used to subset the character vector to get only the elements containing or not containing that pattern. The return value from the call to *grep()* above is 2 because a digit was found in the second element of the character vector *Addresses*. The *grep()* is also available in the shell (see Chapter **??**).

The search for a fake word that contains punctuation or a digit in the middle of it is handled by the following pattern

```
[[:alpha:]][[:digit:][:punct:]][[:alpha:]]
```

Paying careful attention to the square brackets in this pattern, we see that we are looking for three characters. The first can be any letter in the alphabet (upper or lower case), followed by a digit or punctuation mark, followed by another letter. Unfortunately this pattern matches the text string "it's", which we do not want. This problem can be resolved by providing the specific punctuation marks that are acceptable in the character class,

```
[[:alpha:]][[:digit:]!@#$%^&*():;?,.][[:alpha:]]
```

or we could first remove any quotation marks from the search string and then use the original pattern.

```
> s = c(subjectLines, "It's me")
> s
[1] " Re: 90 days"     "[SPAM:XXXXXXXXX]"
[3] " Fancy rep1!c@ted watches"  "It's me"
> newString = gsub("'", "", s)
> grep("[[:alpha:]][[:digit:][:punct:]][[:alpha:]]", newString)
[1] 2 3
```

Note that the search did not match the "Re:" because the colon is followed by a blank, nor does it match the fourth element because the `'` has been removed from the string. It does find a match in the second element and the third element. To find exactly where the pattern was found in these strings, we can use the *regexpr()* function.

```
> regexpr("[[:alpha:]][[:digit:][:punct:]][[:alpha:]]",
       newString)
[1] -1  5 13 -1
attr(,"match.length")
[1] -1  3  3 -1
```

The return value of -1 indicates that the pattern was not found in the first and fourth elements of *newString*. As for the second element, the return value of 5 indicates that the pattern was found beginning at the fifth character in the string, and the value of the attribute *match.length* for this element in the return vector indicates that the match is three characters long. The fifth through eighth characters in `[SPAM:XXXXXXXXX]` are `M:X` and so we have found the pattern we expected to find.

Notice that the match found in the third element of *newString* uncovers one more limitation in our pattern specification: the pattern was found in characters 13-15 in the string, i.e. `c@t`. That is, we did not find `p1!c` because it consists of four characters: a letter, followed by a digit, followed by a punctuation mark, followed by a letter. To search for the more general pattern of any number of digits or punctuation marks between letters, we must change the pattern as follows.

```
[[:alpha:]][[:digit:][:punct:]]+[[:alpha:]]
```

The plus sign between the second and third characters in the pattern indicates that the second character may appear one or more times.

1.3.3 Meta Characters

The characters `[ ]` and `[: :]` and + are given special meaning in a pattern; these special characters are called meta characters. Regular expressions offer a rich set of meta characters for pattern matching. For example, meta characters make easy work of the task to derive is a logical that indicates whether or not the subject line in an email is all capital letters. This phenomena in email is referred to as yelling. Here is a case when the complement of a character set is useful because we allow any character except lower case letters of the alphabet in the subject line of the email. But we face the problem of needing every character in the subject line to *not* be a lower case letter. We want to specify a pattern that consists of non-lower case letters from the beginning to end without knowing how long it is. Again, we can write a specialized function to do the work,

```
> subjectLines
[1] " Re: 90 days"      "[SPAM:XXXXXXXXX]"
[3] " Fancy rep1!c@ted watches"
> all(strsplit(subjectLines,"")[[1]] %in% LETTERS)
> FALSE
```

but regular expressions provide a clean, clear way to express this pattern via meta characters.

```
^[^[:lower:]]*$+
```

To explain, the first character in this pattern, the ^ is the anchor for the beginning of the string, and the last character, `$` is the anchor to specify the end of a string. The asterisk denotes “any number of times” meaning that the character immediately preceding it may be repeated zero or more times. Put all together, the pattern finds a match when the string consists entirely of non-lower case letters from beginning to end. Note that the caret ^ appears twice in the pattern, and each occurrence has a different meaning. The first caret is the meta character for the beginning of line anchor, and the second caret, which is the first character inside the square brackets, represents the complement meta character that says any character that is not a lower case is a match.

```
> grep("^[^[:lower:]]*$", subjectLines)
[1] 2
```

Table 1.2 provides a list of some of the more useful meta characters. Note that the position of a character in a pattern determines whether of not it is treated as a meta character. For example, the `*` is a meta character in the above pattern that searches for one or more non-lower case letters. However, in the example of Section 1.3.1, the `*` in the character class is treated not as a meta character but as the literal asterisk, `[-+*/]`.

1.4 Advanced Notions

In the Web log analysis (Section 1.1.3), our ultimate goal is to transform a line in the Web log into a line of comma-separated values for the IP address, date, file, status, and, bytes. Before we do this, we look at the simpler problem of extracting the date and time only. That is, we want to grab the information between the square brackets, ignoring the time zone piece (e.g. `-0800`). Square brackets do not appear elsewhere in the line, and so a search for a left square bracket will bring us to the date. Below is one line of text from the web log as a character string in R.

Character	Meaning
^	As the first character in the pattern, anchor for beginning of line
	As the first character inside [ ], exclude these characters.
$	End of line anchor
?	Character or sub-pattern occurs zero or one time
+	Character or sub-pattern occurs one or more times
*	Character or sub-pattern occurs zero or more times
.	Any single character
[ ]	Character class
-	Range within a character class
( )	Group or sub-pattern
|	Alternation, i.e. one sub-pattern or another
{ }	Quantifier: `{n}` means exactly `n` repeats of the sub-pattern
	`{n,m}` `n` to `m` repeats
	`{n,}` `n` or more repeats

Table 1.2: Some useful meta characters. Note to search for one of these characters as a literal, it may have to be preceded by a backslash (or two backslashes in R).

```
> weblog
[1] "169.237.46.168 - - [26/Jan/2004:10:47:58 -0800]
\"GET /stat141/Winter04 HTTP/1.1\" 301 328
\"http://anson.ucdavis.edu/courses/\"
\"Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0;
.NET CLR 1.1.4322)\""
```

Notice that quotation marks appear with a backlash that acts as an escape character, so the quotation mark within the character string does not end the character string.

The following regular expression searches for the left square bracket followed by any number of characters followed by a right square bracket:

```
> regexpr("\\[.*\\]", weblog)
[1] 20
attr(,"match.length")
[1] 28
```

To search for a literal square bracket, we need to use the backslash twice, once to escape from R and again to escape from the regular expression use of `[` as a meta character. The pattern contains two meta characters, the verb+.+ which stands for any character, and the `*` which says any character may be repeated many times (actually zero or more times). Essentially, the pattern will produce a match when it finds any string between `[` and `]`.

The function *regexpr()* returns more detailed information than the other R functions we have seen in this chapter for handling regular expressions. It provides a) which elements of the character vector actually contained the pattern in the regular expression, and also b) identifies the position of the substring that was matched by the regular expression pattern.

The return value from our search is the integer 20, the position of the starting character of the match. In this case, it tells us that the left square bracket is the 20th character in the string. Also, the attribute "match.length" is 28 which indicates that the matching string, from `[` to `]` is 28 characters long. This length of the matching pattern is returned in a slightly odd form because it allows us to treat the return value from *regexpr()* directly as a simple integer vector while still carrying around additional information with it. To get the substring in the string that corresponds to the date and time, we can use this return value along with *substring()*:

```
> x = regexpr("\\[.*\\]", weblog)
> substring(weblog, x + 1, x + attr(x, "match.length")-8)
[1] "26/Jan/2004:10:47:58"
```

The *regexpr()* function is a very useful tool for getting an understanding of what a particular pattern actually matches. We can create a pattern to match and then give it different test strings and see which parts actually match. This is a very important exercise to practice to really understand regular expressions. As an example, if we return to the simple example seen earlier where we look for the string ”one”, we can use *regexpr()* to determine where in the strings it occurs.

```
> regexpr("one", c("a test", "a basic string",
  "and one that we want", "one two three"))
[1] -1 -1  5  1
attr(,"match.length")
[1] -1 -1  3  3
```

The return value is an integer vector with an element for each of the elements in the vector. Each element in the return vector gives the position of the starting character of the match, if it exists, and -1 when no match occurs for that string.

1.4.1 Grouping and references

An alternative way to pull out the date and time from the Web log is via references. That is, we locate the substring in the Web log that is of interest and pull it out by reference. The parentheses meta characters `( )` group together a sub-pattern, which can be referred to in a later pattern. The pattern,

```
.*\\[(.*) [-+].*\\].*
```

looks for a string that consists of any characters any number of times, followed by a left square bracket. Then any characters any number of times followed by a blank, then either a _ or + then and characters, a right square bracket, followed by any characters. Notice that the second pattern of “any characters” is contained in parentheses, i.e. `(.*)`, which makes it a sub-pattern. This sub-pattern can be referred to as `\\1` in a substitution string,

```
> gsub('.*\\[(.*)\\].*', '\\1', weblog)
[1] "29/Dec/2003:06:36:18 -0600"
```

Essentially we have substituted the entire line of text in the Web log with the sub-pattern found in `\\1`, which is the sub-pattern found between the square brackets. We need the final `.*` because without it,

```
> gsub('.*\\[(.*)\\]', '\\1', weblog)
[1] "29/Dec/2003:06:36:18 -0600 \"GET /logo.html
 HTTP/1.1\" 200 244 "http://anson.ucdavis.edu/courses/"
"Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0;
.NET CLR 1.1.4322)"
```

the end of the string is not eliminated. The pattern here matches all of the characters in a line in the Web log up to and including the information in the square brackets. It substitutes all of this with the sub-pattern found, i.e. with the characters between the square brackets, but we want to substitute the entire line with the sub-pattern. Further, we can drop the time zone offset by making our pattern a bit more precise,

```
> gsub('.*\\[(.*) [-+][0-9]+\\].*', '\\1', weblog)
[1] "29/Dec/2003:06:36:18"
```

Here, we sub-pattern consists of those characters in the square brackets that appear before the time zone offset, which is specified by a blank, followed by either a plus or minus, followed by one or more digits and then a right square bracket. The + meta character in `[0-9]+` means one or more of the preceding character, i.e. one or more digit in this case.

1.4.2 Alternation

Grouping can be very handy when you want to express the notion of equivalent sub-patterns. For example, in the Web log the command is either GET or PUT, and we see that it appears in quotes along with the file name.

```
...  "GET /logo.html  HTTP/1.1" 200 244 ...
```

In our search for the file name, we see that it occurs between the GET and the HTTP. A regular expression that extracts the file name can use this structure.

```
> gsub('.*"GET (.*)  HTTP.*', '\\1', weblog)
[1] "/logo.html"
```

However, when the GET is a PUT or when the HTTP is an FTP then our substitution will not work as expected. Alternation comes to the rescue. We search for GET or PUT by using the pattern `(GET|PUT)` and similarly we can replace HTTP with `(HTTP|FTP)`,

```
> gsub('.*"(GET|PUT) (.*)  (HTTP|FTP).*', '\\1', weblog)
[1] "GET"
```

We did not get the file name this time. What went wrong? Our regular expression has changed. Now it has three sub-patterns instead of one. The alternation `(GET|PUT)` is the first sub-pattern, and now the one that we want is the second,

```
> gsub('.*"(GET|PUT) (.*)  (HTTP|FTP).*', '\\2', weblog)
[1] "/logo.html"
```

Note that we could express a character class using *alternation*. For example, rather than `[0-9]`, we could use the construction: "(0|1|2|3|4|5|6|7|8|9)" for our pattern. This is tedious to write and becomes difficult to read as the number of characters to be matched becomes lengthy. We are not succinctly expressing the concept of "match a digit", but instead we are explicitly enumerating the characters. This makes maintaining and understanding the regular expression more difficult. Additionally, these single character alternations are not very efficient. They can slow down the speed with which the regular expression automata performs the matching.

1.4.3 Number of matches

We have seen two meta characters that can be used to denote multiplicity in matching. These are `*` for zero or more and + for one or more. Another is `?` for zero or one. Regular expressions also allow the specification of an explicit number of matches via the curly braces `{ }`. For example, `{4}` denote exactly four. Applied to our time zone problem, we could explicitly specify four digits following the plus/minus sign as follows,

```
> gsub('.*\\[(.*) [-+][0-9]{4}\\].*', '\\1', weblog)
[1] "29/Dec/2003:06:36:18"
```

Ranges can also be specified with the curly braces. For example, `{4,}` means four or more matches, and `{4,7}` means four to seven matches.

At last , we have the concepts that will enable us to tackle the original problem to transform a Web log entry:

```
193.188.97.151 - - [29/Dec/2003:06:36:18 -0600]
"GET /logo.html  HTTP/1.1" 200 244 ...
```

into the comma separated values:

```
193.188.97.151, 29/Dec/2003:06:36:18, /logo.html, 200, 244
```

which has extracted the IP address, date, file name, status, and, number of bytes transferred.

As we have seen, each of these pieces of information can be located in the file by carefully examining the structure of the file. The IP address comes first, and is always followed by two dashes. Then comes the date and time between square brackets, followed by either the command GET or POST command, file name, and HTTP or FTP all in quotation marks. Finally, the last two numbers in the file are the status and the number of bytes, respectively. We can construct a regular expression that describes the Web log line and uses parentheses to extract sub-patterns of interest.

```
(.*) - - \\[(.*) [-+][0-9]{4}\\]
 "(GET|POST) (.*) (HTTP|FTP)(/1.[01])?" ([0-9]+) (-|[0-9]+).*
```

The first subgroup will match the IP address, the second will match the date and time, the third will match GET or POST, and so on. The subexpressions that we wish to keep are the first, second, fourth, seventh, and eighth. The substitution string can refer to these sub-patterns and build text that consists of these four values separated by commas: `\\1,\\2,\\4,\\7,\\8`

```
> gsub('(.*) - - \\[(.*) [-+][0-9]{4}\\]
"(GET|POST) (.*) (HTTP|FTP)(/1.[01])?" ([0-9]+) (-|[0-9]+).*',
'\\1, \\2, \\4, \\7, \\8', weblog)
[1] "193.188.97.151, 29/Dec/2003:06:36:18, /logo.html , 200, 244"
```

The substitution string skipped over the groups that we needed for the match, but not for the output. (It is possible to avoid numbering these groups, if necessary.)

1.5 Greedy Matching

To figure out, or disambiguate, the meaning of a word, we can see how it is used in other contexts by other authors. For example in a phrase such as “put a program in place”, we (or software for checking grammar) might not know if the prepositional phrase “in place” modifies the verb “put” or the noun “program”. More generally, with phrases that have the form: verb noun1 preposition noun2, we want to determine if the prepositional phrase modifies the verb or the first noun. One way to determine this is to rearrange these words where the new arrangement clearly implies whether it is the verb or the noun that is being modified. The Web can serve as a corpus of examples, where the occurrence of the particular rearrangement in a Web page would constitute a vote for verb or noun. Search engine can assist us in finding these web entries.

One rearrangement is: preposition noun2 up-to-three-words noun1. The occurrence of this ordering of the words would indicate that the prepositional phrase modifies the verb.

In our example, a Google search for `"in place" program` returns entries such as the following three.

> 2003 Total Aging In Place Program. All rights reserved. Web site design, hosting and maintenance provided by The PCA Group, Inc. ...
>
> I will use the knowledge learned from the program in the daily operations of my job as an In-Place Test Technician. Due to the informative materials of the ...
>
> Additionally, qualified students may participate in an internship in place of one of their courses. (See Madrid Internship Program description for details. ...

Notice that in the first phrase, we have the phrase “in place” immediately followed by “program”. However, in the second phrase, program comes before our prepositional phrase, and in the third there are are more than three words between the phrase and “program”.

The actual text returned is not plain text as shown above, but HTML. That is, the text is marked up with annotations that tell the Web browser how to display the text. For example, we see below that there is extra text, such as `</b>` that we wish to ignore when counting words between the preposition and the second noun.

> Additionally, qualified students may participate in an internship `<b>`in place`</b>` of one of their courses. (See Madrid Internship `<b>`Program`</b>` description for details. `<b>...</b>`

Essentially, we want to strip out the html from the text before we go about checking the order of the noun and prepositional phrase and counting the words between them. Consider the following substitution to do exactly that,

```
> googleText
[1] "Additionally, qualified students may participate in an
internship <b>in place</b> of one of their courses.
(See Madrid Internship <b>Program</b> description for details."
> gsub("<.*>", "", googleText)
[1] "Additionally, qualified students may participate in an
 internship  description for details."
```

This substitution did not give us what we expected. The problem is greedy matching. The pattern `<.*>` searches for a pair of angle brackets with any characters between. Although, `<b>` is a match, so is `<b>in place</b>` and

```
<b>in place</b> of one of their courses.
(See Madrid Internship <b>Program<b>+
```

also constitutes a match. All begin with < followed by *any* characters (including > in this case) followed by >. The regular expression engine performs greedy matching here, and matches the largest substring possible, which is more than we want. we need to exclude the > from the set of all characters. Anything but the > can be matched,

```
> gsub("<[^>]*>", "", googleText)
[1] "Additionally, qualified students may participate in an
internship in place of one of their courses.
(See Madrid Internship Program description for details."
```

That is the result we are after.

1.6 Summary

In this chapter we introduced six basic concepts of regular expressions.

1. **Literal String** – Basic matching occurs one character at a time from left to right. Look for the first character in the pattern, when it is found in the string, see if the next character in the sting matches the second character in the pattern, and so on.

2. **Character Sets** – These are collections of equivalent characters, where a match could be any one of the characters specified in the character set. A character set is the collection of of characters between `[` and `]`. Some of the most common collections are named, such as [[:alpha:]] for the letters of the alphabet.

3. **Repetition** – A match may be repeated a specific number of times, e.g. `{m}` for `m` times. Or a range of times, such as `{m,}` m or more times and `{m,n}` m through n times. In addition the meta characters `*` ?+ denote zero or more, one or more, and zero or one, respectively. These quantifiers modify the character or group of characters that immediately precedes it.

4. **Grouping** – Parentheses can be used to form sub-patterns. Groups are useful for alternation, repetition, and referencing.

5. **Alternation** – Alternate patterns may be provided via the | symbol. For example `this|that` matches either `this` or `that`. Parentheses limit the alternation, e.g. `th(is|at)` has the same effect as the previous alternation.

6. **References** – A sub-pattern may be referred to later in the same pattern or in a substitution pattern. The reference is based on the position of the sub-pattern. The first or leftmost sub-pattern is referred to as `\\1`, the second as `\\2`, and so on.

1.7 Exercises

1.8 Resources

http://regexp.resource.googlepages.com/analyzer.html

1.9 Additions

We should add some material on useful and reasonably common extensions to the basic regular expression language and which are supported in R (and other languages) via the `perl = TRUE` option to use the PCRE (Perl Compatible Regular Expression library). Of note are the

- non-greedy (or lazy) matching using the ? qualifer after a quantifier, e.g.

  ```
  regexpr("^a+?", "aaab", perl = TRUE)
  ```

  See p140 of Friedl.

- Look-arounds

  ```
  str = "(-0.791,-0.263].(-38,-1.24].(0.96,2.43]"
  strsplit(str, "\\.(?![0-9])", perl = TRUE)
  ```

- Avoiding capturing a group (?:pattern) and explicitly naming captured matches.

- Unicode (i.e. adding unicode to patterns)

- Working with different locales and languages.