Regular Expressions (RegEx)

Stat 133, Fall 2024

Lectures 17 - 19, 10/18/2024

1. Introduction

Regular expressions (regex) are patterns that allow for powerful search and manipulation of text strings. They are essential tools in data science for tasks such as data cleaning, parsing, and extracting information from unstructured data. In R, the primary functions for working with regex include:

- `grep()` - searches for patterns within text and returns the indices of matches
- `grepl()` - returns a logical vector indicating if a pattern was found
- `gsub()` - performs search and replace operations
- `regexpr()` and `gregexpr()` - return positions of matches and can capture multiple occurrences

2. Basic Regex Syntax

In regex, characters are divided into two main categories:

- **Literal characters** - characters that match themselves, such as `a`, `b`, `1`, etc.
- **Metacharacters** - symbols with special meanings in regex, including `.`, `*`, `?`, and \.

2.1. Example of Basic Pattern Matching

The following code checks if the word "data" is present in a text string:

```
text <- "statistics and data science"
grepl("data", text)  # Returns TRUE if "data" is found
```

Output:

```
[1] TRUE
```

Here, `grepl()` returns `TRUE` since "data" is a substring within the text.

3. Character Sets

Character sets are defined using square brackets `[]`, allowing us to specify groups of characters to match. For instance, `[abc]` matches either "a", "b", or "c".

3.1. Using Ranges in Character Sets

Ranges simplify the specification of a range of characters, e.g., `[a-z]` for lowercase letters.

```
text <- "abc123XYZ"
grepl("[a-z]", text)  # Checks for any lowercase letter
grepl("[0-9]", text)  # Checks for any digit
```

Output:

```
[1] TRUE
[1] TRUE
```

3.2. Negating Character Sets

Using `^` inside square brackets negates the set, matching any character not in the set.

```
text <- "Hello World!"
grepl("[^a-zA-Z]", text)  # Checks for non-alphabet characters
```

Output:

```
[1] TRUE
```

3.3. Combining Character Sets & Ranges

Character sets can include ranges, specific characters, and even metacharacters for flexibility:

```
text <- "The price is $100."
grepl("[0-9$]", text)  # Checks for digits or the "$" symbol
```

This flexibility allows complex searches within strings.

4. Anchors

Anchors define specific positions within text, like the beginning or end of a line.

4.1. Start of Line Anchor (^)

The ^ symbol asserts the start of a line or string, useful for finding patterns at the beginning.

```
text <- "data science is powerful"
grepl("^data", text)  # Returns TRUE if "data" is at the start
```

4.2. End of Line Anchor ($)

The $ symbol asserts the end of a line or string.

```
text <- "powerful data"
grepl("data$", text)  # Returns TRUE if "data" is at the end
```

These anchors are essential for precise matches in text analysis.

5. Quantifiers

Quantifiers control the number of repetitions allowed for characters or groups in a pattern:

- * - zero or more occurrences
- + - one or more occurrences
- ? - zero or one occurrence
- {n} - exactly n occurrences
- {n,m} - between n and m occurrences

5.1. Example with * Quantifier

Matches "a" followed by zero or more "b"s, and then "c".

```
text <- "abbc"
grepl("ab*c", text)  # Returns TRUE as "b" can appear zero or more times
```

Output:

```
[1] TRUE
```

5.2. Example with + Quantifier

Requires one or more occurrences of ”b”.

```
text <- "abc"
grepl("ab+c", text)  # Returns FALSE as no "b" follows "a"
```

6. Boundaries

Boundaries define the context for matches in terms of word positions.

6.1. Word Boundary (\b)

The assertion `\b` specifies a position at the start or end of a word.

```
text <- "data analysis is essential"
grepl("\\bdata\\b", text)  # Matches "data" as a whole word
```

6.2. Non-Word Boundary (\B)

The assertion `\B` matches positions within words.

```
text <- "looploop"
grepl("oo\\B", text)  # Matches "oo" not at a boundary
```

These boundaries are key when isolating specific word patterns in complex text.

7. Applications

7.1. Pattern Replacement with `gsub()`

Replacing words or patterns within text is common in data cleaning.

```
text <- "Clean the data and analyze the data."
result <- gsub("data", "information", text)
print(result)
```

Output:

```
[1] "Clean the information and analyze the information."
```

7.2. Extracting Positions with `regexpr()`

The `regexpr()` function is used to locate the starting position of the first match.

```
text <- "R is popular in data science"
match_pos <- regexpr("data", text)
print(match_pos)
```

Output:

```
[1] 15
```

This example illustrates `regexpr()`'s utility in locating specific phrases.

8. Combining Regex Patterns

Regex patterns can be combined to form complex searches. For instance, matching a capitalized word at the start of a sentence:

```
text <- "Data analysis is important. Science evolves."
grepl("^[A-Z][a-z]+", text)  # Checks for capitalized words at the start
```

Output:

```
[1] TRUE
```

8.1. Example of Complex Pattern Matching

Finding words that contain either "data" or "science" using alternation:

```
text <- "data science is evolving"
grepl("data|science", text)  # Checks for either "data" or "science"
```

Output:

```
[1] TRUE
```

9. Summary

Using regex effectively requires practice and familiarity with syntax and functions. Key takeaways include:

- Regex functions such as `grep()`, `gsub()`, and `regexpr()` are essential for text processing.
- Anchors, quantifiers, and boundaries enable precise control over matching patterns.
- Character sets allow for flexibility, while negations and ranges simplify pattern specification.

Regex is integral to efficient data processing and manipulation, enabling automation and accuracy in handling text data in R.