- Tutorial
- Exercícios
- Apostila

Edição de Texto

Em muitos casos e aplicações as informações que queremos manipular ou analisar não são numéricas. Elas são compostas por caracteres de texto. Essas informações vão desde variáveis categóricas (e.g. “sim” ou “não”) até campos de texto livre, contendo descrições associadas a uma dada observação. Em ecologia e ciências florestais, um caso muito comum desse tipo de informação é o nome da espécie do qual foi tomada uma certa medida ou realizada uma certa contagem.

Se o nosso conjunto de dados é pequeno, editar manualmente o texto pode ser a solução mais direta ou simples. Mas à medida que o conjunto de dados cresce e a complexidade das edições aumenta, aprender técnicas básicas de manipulação de texto podem poupar muito tempo de trabalho.

Assim como outras linguagens de programação, o R está preparado e possui várias funções disponíveis para manipular texto. Mas, ele também possui as suas particularidades e conhecê-las é um importante passo para atingir a proficiência nessa linguagem. Assim, neste tutorial iremos apresentar conceitos básicos e alguns exemplos práticos de manipulação e edição de texto no R:

Vetores de texto no R

Informações de texto no R são armazenadas em objetos que são vetores da classe `character`, que podem ser vazios ou ter comprimento 1 ou mais.

O R possui uma função para gerar vetores nulos ou vários de texto, a função `character()` que possui apenas um argumento `length` que define o tamanho do vetor. Por exemplo:

```
vetor <- character(length = 5)
vetor

class(vetor)
is.character(vetor)
```

Frequentemente, o vetor de texto é obtido a partir de um arquivo contendo o conjunto de dados. Para esse tutorial, iremos gerar um pequeno vetor de texto com nomes fictícios de espécies para mostrar

como vetores de texto podem ser operados usando algumas funcionalidades básicas do R para vetores com outras classes de elementos (e.g. numéricos ou lógicos)

```
texto <- c("esp A", "esp C", "esp B", "esp B", "esp B1", "Esp D10")
sort(texto)
sample(x = texto, size = 2)
duplicated(texto)
```

Veja que a função `sort` organiza tanto vetores númericos quanto de texto, este último que são organizados em ordem alfabética. Da mesma maneira as funções `sample` e `duplicated` (e várias outras) também funcionam para vetores de texto. Obviamente, nem todas as funções irão funcionar, em especial aquelas relacionadas a operações exclusivas para números (e.g. matemáticas).

```
sum(texto)
## Erro...
```

Funções básicas de manipulação texto

Assim como existem funções específicas para manipular vetores numéricos, existem funções para manipular vetores de texto.

Número de elementos e caracteres

As funções `length` e `nchar` funcionam para vetores numéricos e para vetores de texto. A primeira retorna o comprimento do vetor, enquanto que a segunda retorna o número de caracteres de cada elemento do vetor.

```
length(texto)
## 6
nchar(texto)
## 5 5 5 5 6 7
```

Capitalização

As funcionalidades básicas relacionadas a tornar letras de vetores em maiúsculas ou minúsculas são: `toupper`, `tolower` e `casefold`. Esta última função é mais flexível e pode transformar as letras tanto em maiúsculas quanto minúsculas

```
tolower(texto) # o mesmo que:
casefold(texto, upper = FALSE)

toupper(texto) # o mesmo que:
casefold(texto, upper = TRUE)
```

Colagem e concatenação

Duas funções muito úteis ao trabalho com vetores de texto são as funções `paste` e `cat`. A primeira concatena elementos um ou mais vetores e o resultado dela pode ser salvo em um objeto ou imprimir o resultado da operação no console do R.

A segunda é mais usada apenas para concatenar e imprimir o resultado no console do R, sendo muito usado dentro de funções e dos ciclos de iteração (i.e. loops) que vimos anteriormente.

Uma variante da função `paste` é a função `paste0`, que funciona de maneira parecida porém não possui opções sobre como separar os elementos colados.

```
paste(texto, "- area 1")
paste0(texto, "-area", 1:length(texto))

cat("As espécies encontradas foram:", texto)
```

Localização e substituição

Duas funções muito úteis são as funções de localização `grep` e `grepl`. A primeira retorna apenas a posição dos elementos que contém a busca desejada (no nosso exemplo a palavra maiúscula 'C'). Já a função `grepl` retorno o vetor lógico de mesmo tamanho com 'TRUE' nas posições que contém a busca desejada.

```
grep("C", texto)
grepl("C", texto)
```

Além de localizar é possível substituir caracteres de texto usando as funções `chartr` ou `sub` e `gsub`. A função `chartr` faz substituições caractere a caractere. Já as funções `sub` e `gsub` (de substituição e substituição global) são mais versáteis.

```
chartr("C", "E", texto)
chartr("C", "E1", texto) # não era bem isso que queríamos

sub("C", "E1", texto)
gsub("C", "E1", texto)
```

Divisão, corte e abreviação

É possível ainda dividir ou “quebrar” vetores de texto baseado em letras, espaços, pontuação ou qualquer outro caractere desejado. Para usamos a função `strsplit`.

No exemplo abaixo, fazemos a separação por espaço, com o intuito de separar as duas partes dos nossos nomes fictícios de espécie. Note que a função não retorna um vetor e sim uma lista de mesmo

comprimento com o resultado da operação.

```
strsplit(texto, split = " ")
```

É possível extrair caracteres dos vetores de texto. Isso pode ser feito extraindo um número fixo do início de cada elemento usando a função `strtrim` ou usando comprimentos variáveis e mais flexíveis tanto no início quanto no fim de cada elemento usando as funções `substr` ou `substring`.

```
strtrim(texto, width = 2)
substr(texto, start = 1, stop = 2)
substring(texto, first = 1, last = 2)
```

Por último é possível abreviar vetores de texto usando a função `abbreviate` e seu argumento `minlength` para escolher o número ideal de caracteres que cada elemento do resultado deve ter.

```
abbreviate(texto, minlength = 4)
```

Expressões regulares

Os exemplos e funcionalidades que vimos acima dependiam de buscas exatas. Ou seja, a função só identificava, localizava, substituía ou dividia o texto baseado em caracteres de busca únicos e explicitamente definidos (e.g. `sub("C", "E1", texto)`). Nesse formato, cada operação é específica e, de certa forma, limitada.

Para aproveitar todo o potencial da manipulação de texto devemos usar as chamadas Expressões Regulares (ou 'RegExp' do inglês). Usadas em quase todas as linguagens de programação (e.g. Java, Python, Ruby, etc.) essas expressões permitem identificar caracteres (ou cadeias de caracteres) de interesse de maneira mais flexível, concisa e generalizável.

'RegExp' é praticamente uma linguagem à parte e nem sempre é fácil de criar ou entender ou traduzir uma expressão regular. Alguns sites podem ajudar nessa tarefa (e.g. www.regextranslator.com). Mas entender o básico já é muito útil para tornar a manipulação do texto ainda mais poderosa.

Por exemplo, podemos usar 'RegExp' para identificar qualquer elemento em um vetor de texto que termina com letras maiúsculas. Para isso, podemos usar a seguinte expressão regular (ou padrão de busca):

```
grepl('[A-Z]$', texto)
```

Em bom português, a expressão `[A-Z]$` que dizer: 'indentifique o elemento contendo qualquer letra maiúscula de A a Z (o componente `[A-Z]`) no final da cadeia de caracteres (componente `$`)'.

Padrões individuais e âncoras

Na expressão acima introduzimos dois elementos básicos: as sequências ou intervalos de caracteres

de busca (declarados usando os colchetes `[]`) e as âncoras, que podem ser usadas para identificar o fim da cadeia de caracteres (declarado usando $) ou no ínicio (declarado usando ^).

O resultado da busca é bem distinto se usarmos a âncora de início, pois nenhum dos elementos começam com uma letra maiúscula. O resultado também depende da sequência de letras declaradas. Por exemplo, poderíamos buscar apenas vogais maiúsculas ao invés de todas as letras maiúsculas:

```
grepl('^[A-Z]', texto)
grepl('[AEIOU]$', texto)
```

Existem muitos outros padrões e âncoras para expressões regulares, mas os mais usuais são:

```
* ''\\d'' ou ''[:digit:]'': qualquer digito/número (oposto ''\\D'')
* ''\\s'' ou ''[:space:]'': qualquer espaço (oposto ''\\S'')
* ''\\b'': início e fim (limites) de palavras (oposto ''\\B'')
* ''\\w'' ou ''[:alnum:]'': qualquer letra ou dígito (oposto
''\\W'')
* ''[:alpha:]'': qualquer letra
* ''[:lower:]'': qualquer letra minúscula
* ''[:upper:]'': qualquer letra maiúscula
* ''[:punct:]'': qualquer pontuação
* ''[:ascii:]'': qualquer caracteres ASCII
```

Alternância

E se eu quiser identificar qualquer elemento de texto que termina ou começa com letras maiúsculas? Você pode combinar duas expressões com o sinal de 'ou' que no R é representado pelo operador `|` (barra vertical).

```
grepl('[A-Z]$', texto) | grepl('^[A-Z]', texto)
```

Ou você pode usar alternância dentro da própria expressão usando o mesmo operador `|`, que permite buscar dois ou mais padrões de busca de uma vez só.

```
grepl('[A-Z]$|^[A-Z]', texto)
```

Agrupamento

Além de identificar podemos também substituir padrões usando RegExp. Por exemplo, vamos substituir todo elemento de texto que termine com letras maiúsculas seguidas de um espaço, que pode ser traduzido para RegExp usando o padrão `' [A-Z]$'`. Iremos substituir esse padrão por um

elemento vazio, o que equivale a um apagador.

```
gsub(' [A-Z]$', '', texto)
```

Note que, como o padrão não é identificado nos últimos elementos do vetor `texto`, a substituição não é feita.

É possível ainda fazer substituições mais arrojadas. Por exemplo, substituir espaços por traços e letras finais maiúsculas por minúsculas, usando agrupamento. Nesse caso específico, precisamos definir o argumento `perl` como TRUE, para ativarmos a compatibilidade com expressões regulares em linguagem Perl (mas não se preocupem com isso agora).

```
gsub('( )([A-Z]$)', '-\\L\\2', texto, perl = TRUE)
```

O agrupamento é feito usando '()' (parênteses) e o seu uso permite isolar os elementos que queremos identificar (i.e. o escopo). No exemplo acima, em português, estamos identificando dois grupos: um espaço `'( )'` seguido de uma letra final maiúscula de A a Z `'([A-Z]$)'`.

Note que ao usar agrupamento, podemos fazer substituições para cada grupo. No exemplo acima estamos substituindo o primeiro grupo (o espaço) por um traço (`-`) e transformamos o segundo grupo (`[A-Z]$`, identificado por `\\2` na substituição) em letras minúsculas, usando o `\\L` (L representa a plavra em inglês 'lower').

Quantificadores

Usando 'RegExp' é possível também especificar a quantidade de vezes que queremos identificar ou operar caracteres de texto, usando os chamados 'quantificadores'. Eles vem logo após o padrão em si e eles especificam buscas de um padrão pelo menos uma vez, todas as vezes ou o número de vezes que você quiser!

Os quantificadores em R e seus significados são:

```
* ''?'' (interrogação): identifica o padrão no máximo 1 vez (i.e. zero
ou 1)
* ''*'' (aterisco): identifica o padrão zero ou mais vezes
* ''+'' (sinal de adição): identifica o padrão pelo menos 1 vez
* ''{n}'' : identifica o padrão exatamente n vezes
* ''{n,}'' : identifica o padrão pelo menos n vezes
* ''{n,m}'' : identifica o padrão pelo menos n e no máximo m vezes
```

Vamos aos exemplos para deixar os quantificadores mais palpáveis. Para isso, vamos usar um vetor de nomes de pessoas como exemplo (para facilitar).

Primeiro, vamos identificar os elementos que contém zero ou 1 letra 'm'

```
nomes <- c("renato", "maria", "mohammed", "luís 4", "luís 15")
grep(pattern= "m+", nomes, value = TRUE) # pelo menos 1 'm'
grep(pattern= "m{2}", nomes, value = TRUE) # pelo menos 1 'mm'
grep(pattern= "m+.h", nomes, value = TRUE) # pelo menos 1 'm' e 'h'
grep(pattern= "\\d", nomes, value = TRUE) # nomes com digitos
grep(pattern= "\\d{2}", nomes, value = TRUE) # pelo menos 2 digitos
```

Funções específicas para RegEx no R

Se você quiser extrair o resultado das buscas, ao invés de apenas listar quais elementos contém a busca, você pode usar as funções do R que são específicas para o uso de expressões regulares. Por exemplo, as funções `regexpr()` e `regexec()` fazem a busca e a função `regmatches()` uso o resultado dessas funções para fazer a extração. A diferença entre as duas funções de busca é na maneira que elas retornam a busca (vetor ou lista).

```
regmatches(nomes, m = regexpr('m{2}', nomes))
regmatches(nomes, m = regexec('m{2}', nomes))
```

Para saber mais

Pronto, você já sabe o básico para poder manipular vetores de texto no R. Veja abaixo algumas sugestões para você se aprofundar no assunto.

- veja o help do próprio R usando `?regexp`
- site do wikipedia em português (https://pt.wikipedia.org/wiki/Express%C3%A3o_regular)
- “Handling and Processing Strings in R” (https://gotellilab.github.io/Bio381/Scripts/Feb07/HandlingAndProcessingStringsInR.pdf)
- um tutorial do datacamp (https://www.datacamp.com/pt/tutorial/regex-r-regular-expressions-guide) mais focado em RegExp usando o pacote contribuído `stringr`