

# Web Scraping

## Atividade 1

### Pacotes no R

Algumas das funções que vamos utilizar nesta atividade não estão na biblioteca básica do R. Temos, dessa forma, que começar instalando uma biblioteca chamada "rvest". A biblioteca já está instalada nos computadores que vamos utilizar hoje. Porém, vamos refazer o processo de instalação para aprender um pouco mais. Execute o comando abaixo:

```
install.packages("rvest")
install.packages("dplyr")
install.packages("stringr")
```

Uma vez instalada a biblioteca, as funções não estão automaticamente disponíveis. Para torná-las disponíveis é preciso "chamar" a biblioteca. Vamos fazer isso com a biblioteca "rvest", "dplyr" e "stringr". Execute o comando abaixo:

```
library(rvest) # Web Scraping
library(dplyr) # Manipular os dados
library(stringr) # Substituir padroes nos textos
```

Excelente! Já temos as funções que precisamos disponíveis na nossa sessão. Vamos utilizá-las logo mais.

### Atividade inicial - Pesquisa de Proposições na ALESP

*For loop e links com numeração de página*

Vamos começar visitando o site da ALESP e entrar na ferramenta de pesquisa de proposições: <http://www.al.sp.gov.br/alesp/pesquisa-proposicoes/> No site, vamos elaborar uma pesquisa qualquer que retorne uma quantidade de respostas que manualmente seria no mínimo ineficiente coletarmos. Por exemplo, podemos pesquisar por todas as proposições relacionados à palavra "merenda".

O resultado da pesquisa é dividido em diversas páginas com 10 observações em cada uma. Há 4 informações sobre as proposições: data, título (e número do projeto), autor e etapa.

Podemos prosseguir, clicando nos botões de navegação ao final da página, para as demais páginas da pesquisa. Por exemplo, podemos ir para a página 2 clicando uma vez na seta indicando à direita.

OBS: Há uma razão importante para começarmos nosso teste com a segunda página da busca. Em diversos servidores web, como este da ALESP, o link (endereço url) da primeira página é "diferente" dos demais. Em geral, os links são semelhantes da segunda página em diante.

Nossa primeira tarefa consiste em capturar estas informações. Vamos, no decorrer da atividade aprender bastante sobre R, objetos, estruturas de dados, loops e captura de tabelas em HTML.

Vamos armazenar a URL em um objeto ("url\_base", mas você pode dar qualquer nome que quiser).

```
url_base <- "http://www.al.sp.gov.br/alesp/pesquisa-proposicoes/?direction=acima&lastPage=75&currentPage=1&act=det"
```

Há muitas informações nesse link, basicamente todos os campos que poderiam ter sido especificados na busca são apresentados no

endereço. Como preenchemos apenas com o termo (“merenda”), esse será o único parametro definido na URL (“text=merenda”).

Também podemos observar que a pesquisa retornou 75 páginas (“lastPage=75”) e que a página atual é a de número 1 (“currentPage=1”) – página 2 da pesquisa é numerada como 0 nesta ferramenta de busca, mas o padrão muda de página para página.

Podemos ver que há muitas páginas de resultado para a palavra-chave que utilizamos. Nossa desafio é conseguir “passar” de forma eficiente por elas, ou seja, acessar os 75 links e “raspar” o seu conteúdo. Para isso, usaremos uma função essencial na programação, o “for loop”.

Loops são processos iterativos e são extremamente úteis para instruir o computador a repetir uma tarefa por um número finito de vezes. Por exemplo, vamos começar “imprimindo” na tela os números de 1 a 9:

```
for (i in 1:9) {  
  print(i)  
}
```

Simples, não? Vamos ler esta instrução da seguinte maneira: “para cada número i no conjunto que vai de 1 até 9 (essa é a parte no parênteses) imprimir o número i (instrução entre chaves)”. E se quisermos imprimir o número i multiplicado por 7 (o que nos dá a tabuada do 7!!!), como devemos fazer?

```
for (i in 1:9) {  
  print(i * 7)  
}
```

Tente agora construir um exemplo de loop que imprima na tela os números de 3 a 15 multiplicados por 10 como exercício.

*Substituição com `stringr::str_replace`*

Ótimo! Já temos alguma intuição sobre como loops funcionam. Podemos agora fazer “passar” pelas páginas que contêm a informação que nos interessa. Temos que escrever uma pequena instrução que indique ao programa que queremos passar pelas páginas de 1 a 75, substituindo apenas o número da página atual – “currentPage” – no endereço URL que guardamos no objeto `url_base`.

Nos falta, porém, uma função que nos permita substituir no texto básico do URL (“`url_base`”) os números das páginas. Há mais de uma opção para realizar essa tarefa, mas aqui usaremos a função `str_replace` do pacote `stringr`. Não se preocupem com caso não entenderem muito agora, veremos mais utilidade destes pacotes na Aula 10.

Com a função “`str_replace`” podemos substituir um pedaço de um objeto de texto por outro, de acordo com o critério especificado. Os argumentos (o que vai entre os parenteses) da função são, em ordem, objeto no qual a substituição ocorrerá, o termo a ser substituído e o termo a ser colocado no lugar.

Na prática, ela funciona da seguinte forma:

```
o_que_procuro_para_substituir <- "palavra"  
o_que_quero_substituir_por <- "batata"  
meu_texto <- "quero substituir essa palavra"  
  
texto_final <- stringr::str_replace( meu_texto, o_que_procuro_para_substituir, o_que_quero_substituir_por)  
  
print(texto_final)
```

Agora que sabemos substituir partes de textos e fazer loops, podemos mudar o número da página do nosso endereço de pesquisa.

Descobrimos que na URL, o que varia ao clicar na próxima página é o “`currentPage=1`” que vai para “`currentPage=2`”. Nesse caso é o “1” como

segunda página e o “2” como terceira e assim por diante, mas isso não é uma regra.

Vamos agora substituir na URL da página 2 da nossa busca o número por algo que “guarde o lugar” do número de página. Esse algo é um “placeholder” e pode ser qualquer texto. No caso, usaremos “NUMPAG”. Veja abaixo onde “NUMPAG” foi introduzido no endereço URL.

Obs: Lembremos que ao colocar na URL, não devemos usar as aspas mas elas se mantêm ao escrever a função, pois queremos dizer que procuramos a palavra “NUM\_PAG” e não o objeto chamado NUMPAG.

```
url_base <- "http://www.al.sp.gov.br/alesp/pesquisa-proposicoes/?direction=acima&lastPage=75&currentPage=NUMPAG&ac
```

Por exemplo, se quisermos gerar o link da página 6, podemos escrever:

```
url_base <- "http://www.al.sp.gov.br/alesp/pesquisa-proposicoes/?direction=acima&lastPage=75&currentPage=NUMPAG&ac
url <- stringr::str_replace(url_base, "NUMPAG", "6")
```

Ou, em vez de usar um número diretamente na substituição, podemos usar uma variável que represente um número – por exemplo a variável i, que já usamos no loop anteriormente.

```
i = "6"
url <- stringr::str_replace(url_base, "NUMPAG", i)
print(url)
```

Agora que temos o código substituindo funcionando, vamos implementar o loop para que as URLs das páginas sejam geradas automaticamente. Por exemplo, se quisermos “imprimir” na tela as páginas 0 a 5, podemos usar o seguinte código:

```
url_base <- "http://www.al.sp.gov.br/alesp/pesquisa-proposicoes/?direction=acima&lastPage=75&currentPage=NUMPAG&ac
for(i in 0:5){
  url <- stringr::str_replace(url_base, "NUMPAG", as.character(i))
  print(url)
}
```

[xml2::read\\_html\(\)](#) e [rvest::html\\_table\(\)](#)

Muito mais simples do que parece, não? Mas veja bem, até agora tudo que fizemos foi produzir um texto que, propositalmente, é igual ao endereço das páginas cujo conteúdo nos interessa. Porém, ainda não acessamos o seu conteúdo. Precisamos, agora, de funções que façam algo semelhante a um navegador de internet, ou seja, que se comuniquem com o servidor da página e receba o seu conteúdo.

Por enquanto, vamos usar apenas as funções [xml2::read\\_html\(\)](#) e [rvest::html\\_table\(\)](#), contida na biblioteca [rvest](#) (lembra que chamamos esta biblioteca lá no começo da atividade?). Algo que é legal destacarmos é que a função [read\\_html\(\)](#) não é do [rvest](#), porém ela é instalada junto com o pacote. Estas funções servem bem ao nosso caso: ela recebe uma URL como argumento e captura todas as tabelas da url, escritas em HTML, e retorna uma lista contendo as tabelas.

Vamos ver como ela funciona para a página 2 (ou seja, currentPage=1) contendo tabelas com os resultados em que aparecem o nosso termo pesquisado:

```
url_base <- "http://www.al.sp.gov.br/alesp/pesquisa-proposicoes/?direction=acima&lastPage=75&currentPage=NUMPAG&ac
i <- "1"
url <- stringr::str_replace(url_base, "NUMPAG", i)
tabela <- xml2::read_html(url) %>% # read_html ira ler a url e retornar uma estrutura html
  rvest::html_table(header = T) # html_table ira procurar as tabelas na estrutura html e retornas cada uma delas e
```

```
print(tabela)
```

Simples não? Geramos um endereço de URL e, após obtermos o endereço, capturamos o conteúdo da página (no caso, só as tabelas) usando o link como argumento das funções `read_html` e `html_table`. Antes de avançar, vamos falar um pouco sobre listas no R (pois o resultado deste código é uma lista).

### *Listas*

Um detalhe fundamental do resultado da função `html_table` é que o resultado dela é uma lista. Por que uma lista? Porque pode haver mais de uma tabela na página e cada tabela ocupará uma posição na lista. Para o R, uma lista pode combinar objetos de diversas classes: vetores, data frames, matrizes, etc. No site da ALESP, já nessa pesquisa nos deparamos que a função `html_table` retorna várias tabelas e não apenas a dos resultados das proposições.

Como acessar objetos em uma lista? Podemos utilizar colchetes. Porém, se utilizarmos apenas um colchete, estamos obtendo uma sublistas. Por exemplo, vamos criar diferentes objetos e combiná-los em uma lista:

```
# Objetos variados
matriz <- matrix(c(1:6), nrow=2)
vetor.inteiros <- c(42:1)
vetor.texto <- c("a", "b", "c", "d", "e")
vetor.logico <- c(T, F, T, T, T, T, T, T, F)
texto <- "meu querido texto"
resposta <- 42

# Lista
minha.lista <- list(matriz, vetor.inteiros, vetor.texto, vetor.logico, texto, resposta)
print(minha.lista)
```

Para produzirmos uma sublistas, usamos um colchete (mesmo que a lista só tenha um elemento!):

```
print(minha.lista[1:3])
class(minha.lista[1:3])
print(minha.lista[4])
class(minha.lista[4])
```

Se quisermos usar o objeto de uma lista, ou seja, extraí-lo da lista, devemos usar dois colchetes:

```
print(minha.lista[[4]])
class(minha.lista[[4]])
```

Ao obter as tabelas de uma página como uma lista de tabelas (nem sempre vai parecer que são tabelas, exceto se você entender um pouco de HTML), devemos, portanto, utilizar dois colchetes para extrair a tabela que queremos (para poder combiná-las com as tabelas das demais páginas, algo que faremos ao final). Exemplo (no nosso caso já sabemos que a tabela que queremos ocupa a posição 1 da lista, mas é necessário examinar sempre):

```
url_base <- "http://www.al.sp.gov.br/alesp/pesquisa-proposicoes/?direction=acima&lastPage=75&currentPage=NUMPAG&aci<- "1"
url <- stringr::str_replace(url_base, "NUMPAG", i)
lista.tabelas <- xml2::read_html(url) %>% rvest::html_table(header = T)
tabela <- lista.tabelas[[1]]
print(tabela)
```

```
class(tabela)
```

### Captura das tabelas

Podemos juntar tudo que vimos até agora: loop com a função `for`, substituição com `str_replace`, captura de tabelas em HTML, listas e seus elementos.

Vamos tentar capturar as cinco primeiras páginas do resultado da pesquisa de proposições por meio da palavra-chave “merenda”. Para podermos saber que estamos capturando, vamos usar a função `head`, que retorna as 6 primeiras linhas de um data frame, e a função `print`.

Avance devagar neste ponto. Leia o código abaixo com calma e veja se entendeu o que acontece em cada linha. Já temos um primeiro script de captura de dados quase pronto e é importante estarmos seguros para avançar.

```
url_base <- "http://www.al.sp.gov.br/alesp/pesquisa-proposicoes/?direction=acima&lastPage=75&currentPage=NUMPAG&ac
for (i in 0:4) {
  url <- stringr::str_replace(url_base, "NUMPAG", as.character(i))
  lista.tabelas <- xml2::read_html(url) %>% rvest::html_table(header = T)
  tabela <- lista.tabelas[[1]]
  print(head(tabela))
}
```

Vamos traduzir o que estamos fazendo: “para cada i de 0 a 4, vamos criar um link que é a combinação da URL base (“url\_base”) com i, vamos usar esta combinação (“url”) como argumento das funções para ler e extraír as tabelas e vamos imprimir as 6 primeiras linhas de cada tabela”.

### Data Frames

Excelente, não? Mas e aí? Cadê os dados? O problema é que até agora ainda não fizemos nada com os dados, ou seja, ainda não guardamos eles em novos objetos para depois podermos utilizá-los na análise.

Neste último passo, vamos fazer o seguinte: precisamos de uma estrutura que armazene as informações, então criamos um data frame vazio (chamado “dados”) e, para cada iteração no nosso loop (ou seja, para cada “i”), vamos inserir a tabela da página i como novas linhas no nosso data frame. A função nova que precisamos se chama `dplyr::bind_rows()`. Ela serve para unir diferentes data frames (ou vetores ou matrizes), colocando suas linhas uma debaixo da outra. Vejamos um exemplo antes de avançar:

```
# Criando 2 data frames separados
meus.dados1 <- data.frame("id" = 1:10, "Experimento" = rep(c("Tratamento"), 10))
print(meus.dados1)
meus.dados2 <- data.frame("id" = 11:20, "Experimento" = rep(c("Controle"), 10))
print(meus.dados2)

# Combinando os dois data.frames
meus.dados.completos <- dplyr::bind_rows(meus.dados1, meus.dados2)
print(meus.dados.completos)
```

### Captura das tabelas com armazenamento em data frames

Pronto. Podemos agora criar um data frame vazio (“dados”) e preenchê-lo com os dados capturados em cada iteração. O resultado final será um objeto com todas as tabelas de todas as páginas capturadas, que é o nosso objetivo central.

Novamente vamos trabalhar apenas com as cinco primeiras páginas, mas bastaria alterar um único número para que o processo funcionasse para todas as páginas de resultados - desde que sua conexão de internet e a memória RAM do seu computador sejam boas! (Obs: vamos inserir um “contador” das páginas capturadas com `print(i)`. Isso será muito útil quando quisermos capturar um número grande de páginas,

pois o contador nos dirá em qual iteração (sic, é sem “n” mesmo) do loop estamos. Além disso, incluímos o argumento “stringsAsFactors = FALSE” na função `readHTMLTable` para garantir que as variáveis de texto sejam lidas como “character” e não como “factors” – este é um assunto para você pesquisar após a oficina).

```
url_base <- "http://www.al.sp.gov.br/alesp/pesquisa-proposicoes/?direction=acima&lastPage=75&currentPage=NUMPAG&ac
dados <- data.frame()
for (i in 0:4) {
  print(i)
  url <- stringr::str_replace(url_base, "NUMPAG", as.character(i))
  lista.tabelas <- xml2::read_html(url) %>% rvest::html_table(header = T)
  tabela <- lista.tabelas[[1]]
  dados <- dplyr::bind_rows(dados, tabela)
}
```

Vamos observar o resultado utilizando a função `dplyr::glimpse()`, que retorna a estrutura do data frame, e `tail`, que é como a função `head`, mas retorna as 6 últimas em vez das 6 primeiras observações.

São 50 observações (5 páginas com 10 resultados) e 4 variáveis (data, título, autor, etapa), exatamente como esperávamos. As 4 variáveis são do tipo “character” contêm as informações corretas. As 6 observações apresentam o resultado adequado, o que nos dá uma boa dica que que tudo ocorreu bem até a última página capturada.

```
# Estrutura do data frame
dplyr::glimpse(dados)

# 6 primeiras observações
head(dados)

# 6 últimas observações
tail(dados)
```

Pronto! Conseguimos fazer nossa primeira captura de dados. Se quiser, você pode repetir o procedimento para pegar o resto dos resultados ou reproduzir o mesmo processo para capturar outras informações do site.

## Atividade 2

Na primeira atividades utilizamos o `rvest` para capturar uma tabela de um site. Quando a informação de interesse já se encontra dentro de uma tabela, o trabalho é relativamente fácil e com algumas linhas de código podemos criar um `data.frame`.

O que fazer, entretanto, com páginas que não tem tabelas? Como obter apenas as informações que nos interessam quando o conteúdo está “espalhado” pela página? Utilizaremos, como veremos abaixo, a estrutura do código HTML da própria página para selecionar apenas o que desejamos e construir data frames.

Nosso objetivo nessa atividade será capturar uma única página usando a estrutura do código HTML da página. Já sabemos que, uma vez resolvida a captura de uma página, podemos usar “loop” para capturar quantas quisermos, desde que tenha uma estrutura semelhante.

Antes disso, porém, precisamos falar um pouco sobre XML e HTML.

### Atividade 2.1

#### XML e HTML

XML significa “Extensive Markup Language”. Ou seja, é uma linguagem – e, portanto, tem sintaxe – e é uma linguagem com marcação. Marcação, neste caso, significa que todo o conteúdo de um documento XML está dentro de “marcas”, também conhecidas como “tags”. É

uma linguagem extremamente útil para transporte de dados – por exemplo, a Câmara dos Deputados utiliza XML em seu Web Service para disponibilizar dados abertos (mas você não precisa saber disso se quiser usar o pacote de R bRasilLegis que nós desenvolvemos ; ) – disponível aqui: <https://github.com/leobarone/bRasilLegis>.

Por exemplo, se quisermos organizar a informação sobre um indivíduo que assumiu diversos postos públicos, poderíamos organizar a informação da seguinte maneira:

```
<politicos>
  <politico>
    <id> 33333 </id>
    <nome> Fulano Deputado da Silva </nome>
    <data_nascimento> 3/3/66 </data_nascimento>
    <sexo> Masculino </sexo>
    <cargos>
      <cargo>
        <cargo> prefeito </cargo>
        <partido> PAN </partido>
        <ano_ini> 2005 </ano_ini>
        <ano_fim> 2008 </ano_fim>
      </cargo>
      <cargo>
        <cargo> deputado federal </cargo>
        <partido> PAN </partido>
        <ano_ini> 2003 </ano_ini>
        <ano_fim> 2004 </ano_fim>
      </cargo>
      <cargo>
        <cargo> deputado estadual </cargo>
        <partido> PAN </partido>
        <ano_ini> 1998 </ano_ini>
        <ano_fim> 2002 </ano_fim>
      </cargo>
    </cargos>
  </politico>
</politicos>
```

Exercício (difícil): se tivessemos que representar estes dados em um banco de dados (data frame), como seria? Quantas linhas teria? Quantas colunas teria?

Veja no link abaixo um exemplo de arquivo XML proveniente do Web Service da Câmara dos Deputados:

<http://www.camara.gov.br/SitCamaraWS/Deputados.asmx/ObterDetalhesDeputado?ideCadastro=141428&numLegislatura=>

Abra agora a página inicial da ALESP. Posicione o mouse em qualquer elemento da página e, com o botão direito, selecione “Inspecionar” (varia de navegador para navegador). Você verá o código HTML da página.

Sem precisar observar muito, é fácil identificar que o código HTML da ALESP se assemelha ao nosso breve exemplo de arquivo XML. Não por acaso: HTML é um tipo de XML. Em outras palavras, toda página da internet está em um formato de dados conhecido e, como veremos a seguir, pode ser re-organizado facilmente.

### Tags, nodes, atributos, valores e conteúdo na linguagem XML

Todas as informações em um documento XML estão dispostas em “tags” (id, nome, etc são as tags do nosso exemplo). Um documento XML é um conjunto de “tags” que contém hierarquia. Um conjunto de “tags” hierarquicamente organizado é chamado de “node”. Por exemplo, no

arquivo XML da Câmara dos Deputados apresentado acima, cada tag político contém diversos outras “tags” e formam “nodes”, ou seja, pedaços do arquivo XML.

Em geral, as “tags” vem em pares: uma de abertura e outra de fechamento. O que as diferencia é a barra invertida presente na tag de fechamento. Entre as “tags” de abertura e fechamento vemos o conteúdo da tag, que pode, inclusive, ser outras “tags”. Veja os exemplos abaixo:

```
<minha_tag> Este é o conteúdo da tag </minha_tag>

<tag_pai>
  <tag_filha>
  </tag_filha>
</tag_pai>

<tag_pai> Conteúdo da tag Pai
  <tag_filha> Conteúdo da tag Filha
  </tag_filha>
</tag_pai>
```

Identação (espaços) nos ajudam a ver a hierarquia entre as tags, mas não é obrigatória. Também as quebras de linha são opcionais.

Além do conteúdo e do nome da tag, é extremamente comum encontrarmos “atributos” nas tags em bancos de dados e, sobretudo, em códigos HTML. Atributos ajudam a especificar a tag, ou seja, identificam qual é o seu uso ou carregam quaisquer outras informações referentes. Voltando ao exemplo fictício acima, poderíamos transformar a informação do cargo, que hoje é uma tag cargo dentro de outra tag cargo (horrible, não?) em atributo.

Em vez de:

```
<politicos>
  <politico>
    <id> 33333 </id>
    <nome> Fulano Deputado da Silva </nome>
    <data_nascimento> 3/3/66 </data_nascimento>
    <sexo> Masculino </sexo>
    <cargos>
      <cargo>
        <cargo> prefeito </cargo>
        <partido> PAN </partido>
        <ano_ini> 2005 </ano_ini>
        <ano_fim> 2008 </ano_fim>
      </cargo>
      <cargo>
        <cargo> deputado federal </cargo>
        <partido> PAN </partido>
        <ano_ini> 2003 </ano_ini>
        <ano_fim> 2004 </ano_fim>
      </cargo>
      <cargo>
        <cargo> deputado estadual </cargo>
        <partido> PRONA </partido>
        <ano_ini> 1998 </ano_ini>
        <ano_fim> 2002 </ano_fim>
      </cargo>
    </cargos>
```

```
</politicos>  
</politicos>
```

Teríamos:

```
<politicos>  
  <politico>  
    <id> 33333 </id>  
    <nome> Fulano Deputado da Silva </nome>  
    <data_nascimento> 3/3/66 </data_nascimento>  
    <sexo> Masculino </sexo>  
    <cargos>  
      <cargo tipo = 'prefeito'>  
        <partido> PAN </partido>  
        <ano_ini> 2005 </ano_ini>  
        <ano_fim> 2008 </ano_fim>  
      </cargo>  
      <cargo tipo = 'deputado federal'>  
        <partido> PAN </partido>  
        <ano_ini> 2003 </ano_ini>  
        <ano_fim> 2004 </ano_fim>  
      </cargo>  
      <cargo tipo = 'deputado estadual'>  
        <partido> PRONA </partido>  
        <ano_ini> 1998 </ano_ini>  
        <ano_fim> 2002 </ano_fim>  
      </cargo>  
    </cargos>  
  </politico>  
</politicos>  
</politicos>
```

Veja que agora a tag “cargo” tem um atributo – “tipo” – cujos valores são “prefeito”, “deputado federal” ou “deputado estadual”. Estranho, não? Para bancos de dados em formato XML, faz menos sentido o uso de atributos. Mas para páginas de internet, atributos são essenciais. Por exemplo, sempre que encontrarmos um hyperlink em uma página, contido sempre nas tags de nome “a”, veremos apenas o “texto clicável” (conteúdo), pois o hyperlink estará, na verdade, no atributo “href”. Veja o exemplo

```
<a href = 'http://www.al.sp.gov.br/'> Vá o site da ALESP </a>
```

Tente, no próprio site da ALESP, clicar com o botão direito em um hyperlink qualquer (por exemplo, “TV Assembléia SP”) para observar algo semelhante ao exemplo acima. Adiante vamos ver como atributos são extremamente úteis ao nosso propósito.

### Caminhos no XML e no HTML

O fato de haver hierarquia nos códigos XML e HTML nos permite construir “caminhos”, como se fossem caminhos de pastas em um computador, dentro do código.

Por exemplo, o caminho das “tags” que contém a informação “nome” em nosso exemplo fictício é:

“/politicos/politico/nome”.

O caminho das “tags” que contém a informação “partido” em nosso exemplo fictício, por sua vez, é:

"/politicos/politico/cargos/cargo/partido".

Seguindo tal caminho chegamos às três "tags" que contém a informação desejada.

Simples, não? Mas há um problema: o que fazer quando chegamos a 3 informações diferentes (o indivíuo em nosso exemplo foi eleito duas vezes pelo PAN e uma pelo PRONA)? Há duas alternativas: a primeira, ficamos com as 3 informações armazenadas em um vetor, pois as 3 informações interessam. Isso ocorrerá com frequência.

Mas se quisermos apenas uma das informações, por exemplo, a de quando o indivíuo foi eleito deputado estadual? Podemos usar os atributos e os valores dos atributos das tag para construir o caminho. Neste caso, teríamos como caminho:

"/politicos/politico/cargos/cargo[@tipo = 'deputado estadual']/partido"

Guarde bem este exemplo: ele será nosso modelo quando tentarmos capturar páginas.

Vamos supor que queremos poupar nosso trabalho e sabemos que as únicas "tags" com nome "partido" no nosso documento são aquelas que nos interessam (isso nunca é verdade em um documento HTML). Podemos simplicar nosso caminho de forma a identificar "todas as 'tags' ", não importa em qual nível hierárquico do documento". Neste caso, basta usar duas barras:

"//partido"

Ou "todas as tags 'partido' que sejam descendentes de 'politico', não importa em qual nível hierárquico do documento":

"/politicos/politico//partido"

Ou ainda "todas as tags 'partido' que sejam descendentes de quaisquer tag 'politico', não importa em qual nível hierárquico do documento para qualquer uma das duas":

"//politico//partido"

Ou "todas as 'tags' filhas de qualquer 'tag' 'cargo'" (usa-se um asterisco para indicar 'todas'):

"//cargo/\*"

Observe o potencial dos "caminhos" para a captura de dados: podemos localizar em qualquer documento XML ou HTML uma informação usando a própria estrutura do documento. Não precisamos organizar o documento todo, basta extraí-lo cirurgicamente o que queremos – o que é a regra na raspagem de páginas de internet.

### Links na página de busca da ALESP

Vamos entrar na página principal da ALESP e fazer uma pesquisa simples na caixa de busca – exatamente como faríamos em um jornal ou qualquer outro portal de internet (o processo seria o mesmo em buscadores como Google ou DuckDuckGo). Por exemplo, vamos pesquisar a palavra "merenda". A ferramenta de busca nos retorna uma lista de links que nos encaminharia para diversos documentos ou páginas da ALESP relacionadas ao termo buscado.

Nosso objetivo é construir um vetor com os links dos resultados. Em qualquer página de internet, links estão dentro da tag "a". Uma maneira equivocada de encontrar todos os links da página usando "caminhos em XML" seria:

"//a"

Entretanto, há diversos outros elementos "clicáveis" na página além dos links que nos interessam – por exemplo, as barras laterais, o banner com os logotipos, os links do mapa do portal, etc. Precisamos, portanto, especificar bem o "caminho" para que obtivéssemos apenas os links que nos interessam.

Infelizmente não temos tempo para aprender aprofundadamente HTML, mas podemos usar lógica e intuição para obter caminhos únicos. Neste exemplo, todos as "tags" "a" que nos interessam são filhas de alguma tag "li" (que é abreviação de "list" e indica um único elemento de uma lista). Podemos melhorar nosso caminho:

```
//li/a"
```

A tag li, por sua vez, é filha da tag "ul" ("unordered list"), ou seja, é a tag que dá início à lista não ordenada formada pelos elementos "li".

Novamente, melhoramos nosso caminho:

```
//ul/li/a"
```

E se houver mais de uma "unordered list" na página? Observe que essa tag "ul" tem atributos: class="lista\_navegacao". Algumas tem função para o usuário da página – por exemplo, as tags "a" contém o atributo "href", que é o link do elemento "clicável". Mas, em geral, em uma página de internet os atributos não fazem nada além de identificar as tags para @ programador@. Diversos programas para construção de páginas criam atributos automaticamente. Por exemplo, se você fizer um blog em uma ferramenta qualquer de construção de blogs, sua página terá tags com atributos que você sequer escolheu.

As tags mais comum em páginas HTML são: head, body, div, p, a, table, tbody, td, ul, ol, li, etc. Os atributos mais comuns são: class, id, href (para links), src (para imagens), etc. Em qualquer tutorial básico de HTML você aprenderá sobre elas. Novamente, não precisamos aprender nada sobre HTML e suas tags. Apenas precisamos compreender sua estrutura e saber navegar nela.

Voltando ao nosso exemplo, se usarmos o atributo para especificar o "caminho" para os links teremos:

```
//ul[@class='lista_navegacao']/li/a"
```

Poderíamos subir um nível hierárquico para melhorar o "caminho":

```
//div[@id='lista_resultado']/ul[@class='lista_navegacao']/li/a"
```

Pegou o espírito? Vamos agora dar o nome correto a este "caminho": xPath.

Vamos agora voltar ao R e aprender a usar os caminhos para criar objetos no R.

### Capturar uma página e examinar sua estrutura

O primeiro passo na captura de uma página de internet é criar um objeto que contenha o código HTML da página. Para tanto, usamos a função "readLines" do pacote *rvest* – que é a mesma que utilizámos para qualquer documento de texto. Vamos usar a segunda página da busca pela palavra "merenda" na ALESP como exemplo:

```
rm(list = ls())
library(rvest)
url <- "http://www.al.sp.gov.br/alesp/busca/?q=merenda&page=2"
pagina <- readLines(url)
```

Em primeiro lugar, observe que a estrutura do endereço URL é bastante simples: "q" é o parâmetro que informa o texto buscado e "page" o número da página. Nesta atividade vamos capturar apenas a página 2, mas você já aprendeu na atividade anterior como capturar todas as páginas usando loops.

Note também que essa função apenas captura o texto de uma página e o salva. No objeto criado, não há referências à estrutura de um HTML. Logo se utilizar a função "class" ou "typeof" para identificar a natureza do vetor, recebemos como resposta um "character".

```
class(pagina)
```

Após usar a função “readLines” o R sabe apenas que um texto foi capturado, mas é incapaz de identificar a estrutura do documento – tags, atributos, valores e conteúdos. Precisamos, pois, informar ao R que se trata de um documento XML. O verbo em inglês para esta tarefa se chama “parse”, cuja definição é “to resolve (as a sentence) into component parts of speech and describe them grammatically”. Em outras palavras, “parsear” é identificar a estrutura sintática de um objeto e dividí-lo em seus componentes.

Para páginas em HTML, usaremos a função “read\_html”. Essa função recebe um URL como argumento e retorna um objeto “xml\_document”:

```
pagina <- read_html(url)  
class(pagina)
```

A partir de agora, sempre que capturarmos uma página, seja em HTML ou em XML (como são os RSSs), vamos fazer um “parse”. Com a página HTML salva em um objeto “xml\_document” podemos começar a trabalhar sobre as “tags” e os “nodes” a fim de coletar as informações de nosso interesse.

### Extraindo os conteúdos de uma página com a biblioteca “rvest”

Vamos agora aprender a “navegar” um objeto XML dentro do R e extrair dele apenas o conteúdo que nos interessa.

Basicamente, trabalharemos nas atividades com um conjunto limitado funções: “xml\_find\_all”, que extrai um pedaço (nodes) de um documento XML; “xml\_text”, que extrai de um node o seu conteúdo; e “xml\_attr”, que extrai os valores dos atributos especificados.

Vamos trabalhar com a ferramenta de busca da página inicial da ALESP, em particular com a página 2 de uma busca qualquer.

Em primeiro lugar, criamos um objeto com o endereço da página:

```
url <- "http://www.al.sp.gov.br/alesp/busca/?q=merenda&page=2"
```

Dessa vez, podemos usar o “read\_html” diretamente

```
pagina <- read_html(url)
```

E, finalmente, usamos uma função nova chamada “xml\_root” para eliminar quaisquer conteúdos (normalmente “sujeira”) que estejam foram da tag de maior nível hierárquico no documento (o que é necessário com frequência em páginas de internet e é mais raro em documentos XML):

```
pagina <- xml_root(pagina)
```

Pronto. Temos o conteúdo da página como documento XML pronto para ser “raspado”.

Vamos começar usando a função “xml\_find\_all”. Essa função, como o próprio nome deixa a entender, captura os “nodes” de um documento XML a partir de um caminho especificado por nós. Esse caminho, conhecido como xPath, garante que rasparemos apenas a informação desejada.

Nesse exemplo, o xPath é “//ul[@class='lista\_navegacao']/li/a”. Antes de rodar a função, você consegue entender esse xPath?

```
nodes_link <- xml_find_all(pagina, "//ul[@class='lista_navegacao']/li/a")
```

```
print(nodes_link)
```

Note que o resultado é uma lista que contém em cada posição o primeiro "node set". Ele poderia, inclusive, ter tags filhas, além de seu conteúdo e atributos. Vamos examinar melhor o primeiro node. Precisamos usar dois colchetes para indicar a posição, pois se trata de uma lista e não de um vetor.

```
print(nodes_link[[1]])
```

Trata-se de uma tag "a", com dois atributos, "class" e "href", sendo que o valor deste último é o link para o qual seríamos direcionados ao clicar no conteúdo apresentado na página. Nesse caso somos direcionados para a notícia "Newton Brandão critica administração da merenda escolar em Santo André".

O que nos interessa é extrair diretamente o valor dos atributos (se tiverem alguma informação valiosa) e o conteúdo. Vamos treinar com a primeira posição e extrair o primeiro o conteúdo:

```
conteudo_1 <- xml_text(nodes_link[[1]])  
print(conteudo_1)
```

Pronto, conseguimos extrair o texto. Depois, podemos extrair o valor do atributo "href":

```
atributo_1 <- xml_attr(nodes_link[[1]], attr = "href")  
print(atributo_1)
```

Excelente, não? Se quisessemos apenas a informação do primeiro link resultante da busca, teríamos terminado nossa tarefa. Mas queremos os 10 links. Vamos ver duas maneiras inefficientes de construir um data frame que contenha uma variável com os conteúdos e a outra com os links. Tente decifrá-las:

Sem usar o "for loop":

```
conteudo_1 <- xml_text(nodes_link[[1]])  
conteudo_2 <- xml_text(nodes_link[[2]])  
conteudo_3 <- xml_text(nodes_link[[3]])  
conteudo_4 <- xml_text(nodes_link[[4]])  
conteudo_5 <- xml_text(nodes_link[[5]])  
conteudo_6 <- xml_text(nodes_link[[6]])  
conteudo_7 <- xml_text(nodes_link[[7]])  
conteudo_8 <- xml_text(nodes_link[[8]])  
conteudo_9 <- xml_text(nodes_link[[9]])  
conteudo_10 <- xml_text(nodes_link[[10]])  
  
conteudos <- c(conteudo_1, conteudo_2, conteudo_3, conteudo_4, conteudo_5, conteudo_6, conteudo_7, conteudo_8, cor  
  
atributo_1 <- xml_attr(nodes_link[[1]], attr = "href")  
atributo_2 <- xml_attr(nodes_link[[2]], attr = "href")  
atributo_3 <- xml_attr(nodes_link[[3]], attr = "href")  
atributo_4 <- xml_attr(nodes_link[[4]], attr = "href")  
atributo_5 <- xml_attr(nodes_link[[5]], attr = "href")  
atributo_6 <- xml_attr(nodes_link[[6]], attr = "href")  
atributo_7 <- xml_attr(nodes_link[[7]], attr = "href")  
atributo_8 <- xml_attr(nodes_link[[8]], attr = "href")  
atributo_9 <- xml_attr(nodes_link[[9]], attr = "href")
```

```

atributo_10 <- xml_attr(nodes_link[[10]], attr = "href")

atributos <- c(atributo_1, atributo_2, atributo_3, atributo_4, atributo_5, atributo_6, atributo_7, atributo_8, atr

dados <- data.frame(conteudos, atributos)

head(dados)

```

Usando for loops:

```

conteudos <- c()
atributos <- c()
for (i in 1:10){

  conteudo_i <- xml_text(nodes_link[[1]])
  conteudos <- c(conteudos, conteudo_i)

  atributo_i <- xml_attr(nodes_link[[1]], attr = "href")
  atributos <- c(atributos, atributo_i)
}

dados <- data.frame(conteudos, atributos)
head(dados)

```

Excelente! Temos um data frame após a raspagem.

Há, porém, uma forma mais rápida de resolver o problema: usando a função “map”! Esse será o seu primeiro passo dentro do que chamamos de programação funcional. Vamos imaginar, antes de utilizá-la o seguinte: por que criamos funções? A resposta mais breve seria “porque não queremos repetir determinadas tarefas muitas vezes!” Imagine se, para calcular a média de um vetor, tivesse que somar todos os valores e dividir pelo número de observações. Me parece uma coisa muito chata e por isso eu e você utilizamos a função “mean”.

A ideia do map segue na mesma linha, porém com uma pequena diferença. Dessa vez nós não queremos repetir o uso de uma função muitas vezes. Logo podemos repetir o uso de qualquer função sobre um conjunto de valores sem escrever várias vezes a função ou utilizar loops.

Veja como funciona e note que a função “xml\_text” entra como argumento da função “map”:

```

library(tidyverse)
conteudos <- map(nodes_link, xml_text)

print(conteudos)

```

Incrível, não? Veja como economizamos na “quantidade” de código para a tarefa de capturar os conteúdos. Porém, a resposta da função “map” é uma lista, o que pode ser um pouco assustador já que estávamos trabalhando com vetores. Para resolver esse problema, basta utilizar a função “map\_chr”. Ela devolve um vetor de “characters”.

```

conteudos <- map_chr(nodes_link, xml_text)

print(conteudos)

```

Vamos repetir o procedimento para os valores dos atributos, com o cuidado de nota que o argumento “attr” necessário para a função “xml\_attr” deve vir como argumento também quando usamos a função “map” após os demais. Vamos economizar tempo e pedir, dessa vez, “map\_chr” diretamente:

```
atributos <- map_chr(nodes_link, xml_attr, attr = "href")
print(atributos)
```

Encerramos juntando ambos vetores em um data frame:

```
dados <- data.frame(conteudos, atributos)
head(dados)
```

O desafio a partir de agora é conectar o que aprendemos da raspagem de dados de várias páginas com “for loop”, mas usando a função “map” ou “map\_chr”. Existem outras funções da família “map” que irão variar conforme o tipo de vetor que você deseja como resposta. Para saber mais, recomendo a leitura do capítulo 21 do livro “R 4 Data Science”.

## Atividade 2.2

Agora que sabemos coletar de uma página da busca o link por meio da extração de atributos de um node e o título do resultado com a função “xml\_text”, precisamos repetir o procedimento para todas as páginas de resultado.

Fizemos algo semelhante na atividade 1, mas ainda usávamos a função “html\_table”. O objetivo é repetir o que fizemos lá, mas com as novas funções que vimos ao longo da atividade 2.1.

Para isso, vamos usar o “for loop” da atividade 1 para ir de uma página a outra.

O primeiro passo é, mais uma vez, ter o nosso link da pesquisa que queremos coletar armazenado em um objeto.

```
rm(list = ls()) #limpa a área de trabalho do R
library(tidyverse)
library(rvest)

urlbase <- "http://www.al.sp.gov.br/alesp/busca/?q=merenda&page="
```

### Função str\_c

Como é possível reparar, o número da página fica ao final do link, por isso podemos utilizar uma nova função chamada “str\_c” ou “colar” ao invés da função “str\_replace”.

O que faremos é colar nosso contador (o “i”, aquilo que vai mudar a cada vez que o loop realizar uma iteração) no final do link. Então o que queremos é uma combinação do nosso url com o contador sem nada separando os dois. A função “str\_c” é ideal para isso e funciona com os argumentos da seguinte maneira: primeiro texto a ser colado, segundo texto a ser colado, terceiro, ..., último, e, finalmente, a especificação de qual é o separador que você deseja entre os textos (pode ser vazio “”).

Na linguagem do R, escreveremos assim para o nosso caso:

```
i = 10

str_c(urlbase, i, sep = "")
```

A “URL” é o endereço da página de busca, o “i” é o contador numérico do loop e o último argumento refere-se ao separador que igualamos a um par de aspas sem nada dentro, deixando claro que para funcionar corretamente nada, nem uma barra de espaço, deve ficar entre o endereço e o contador.

### *Coletando o conteúdo e o atributo de todos os links*

A lógica de coleta do atributo e do conteúdo de um node continua o mesma. A única diferença é que precisamos aplicar isso para todas as páginas. Agora que temos a URL construída, podemos montar um “for loop” que fará isso para nós.

No momento que essa atividade foi feita, a pesquisa pelo termo “merenda” tinha 1068 páginas de resultado, o que implica que o nosso loop irá de 1, a primeira página, até a página 1068, a última. Colocamos um “print(i)” para mostrar que página o loop está durante a execução.

```
for (i in 1:1068){  
  print(i)  
}
```

O que precisamos agora é incluir o que foi discutido ao longo da atividade 2.1. Para extrair as informações de uma página da internet precisamos examinar o código HTML, ler no R e transformar em um objeto XML.

```
pagina <- read_html(urlbase)  
  
pagina <- xml_root(pagina)
```

Sabemos também coletar de forma eficiente todos os títulos e links por meio do “xml\_text” e “xml\_attr”, respectivamente.

```
nodes_link <- xml_find_all(pagina, "//ul[@class='lista_navegacao']/li/a")  
  
conteudos <- map_chr(nodes_link, xml_text)  
  
atributos <- map_chr(nodes_link, xml_attr, attr = "href")
```

Falta empilhar o que produziremos em cada iteração do loop de uma forma que facilite a visualização. Usaremos a função “bind\_row” com data frames, pois para cada página agora, teremos 10 resultados em uma tabela com duas variáveis. O que queremos é a junção dos 10 resultados em cada uma das 162 páginas.

```
dados <- rbind(dados, data.frame(conteudos, atributos))
```

Chegou o momento de colocar dentro loop tudo o que queremos que execute em cada uma das vezes que ele ocorrer. Ou seja, que imprima na tela a página que está executando, que a URL da página de resultados seja construída com a função “str\_c”, para todas elas o código HTML seja examinado, lido no R e transformado em objeto XML, cole todos os links e todos os títulos e que “empilhe”. Lembrando que não podemos esquecer de definir a URL que estamos usando e criar um data frame vazio para colocar todos os links e títulos coletados antes de iniciar o loop.

Por questão de tempo, não iremos criar um data.frame das 1000 páginas. Vamos nos concentrar nas 30 primeiras. Caso queira, sinta-se à vontade para colocar a quantidade que quiser no loop.

```
urlbase <- "http://www.al.sp.gov.br/alesp/busca/?q=merenda&page="  
  
dados <- tibble()  
  
for (i in 1:30){  
  print(i)
```

```
url <- paste(urlbase, i, sep = "")  
  
pagina <- read_html(url)  
pagina <- xml_root(pagina)  
  
nodes_link <- xml_find_all(pagina, "//ul[@class='lista_navegacao']/li/a")  
  
conteudos <- map_chr(nodes_link, xml_text)  
  
atributos <- map_chr(nodes_link, xml_attr, attr = "href")  
  
dados <- bind_rows(dados, data.frame(conteudos, atributos))  
  
}
```

Pronto! Temos agora todos os títulos e links de todos os resultados do site da ALESP para a palavra “merenda” reunidos.

## Atividade 3

Nas atividades 1 e 2 utilizamos as ferramentas básicas de captura de dados disponíveis na biblioteca `rvest`. Em primeiro, aprendemos a capturar várias páginas contendo tabelas em formato HTML de uma só vez. Depois, aprendemos como um documento XML está estruturado e que podemos extrair com precisão os conteúdos de tags e os valores dos atributos das tags de páginas escritas em HTML. Nesta última atividade vamos colocar tudo em prática e construir um banco de dados de notícias. O nosso exemplo será o conjunto de notícias (516 na data da construção deste tutorial) publicadas sobre eleições no site do instituto de pesquisa DataFolha. Ainda que o DataFolha não seja um portal, por estar vinculado ao jornal Folha de São Paulo e ao portal UOL, a busca do DataFolha se assemelha muito às ferramentas de busca destes últimos.

Entre no link abaixo e veja como está estruturada a busca do DataFolha sobre eleições:

[http://search.folha.uol.com.br/search?q=elei%E7%F5es&site=datafolha%2Feleicoes&sr=1&skin=datafolha&results\\_count=516&search\\_time=0.255&url=http%3A%2F%2Fsearch.folha.uol.com.br%2Fsearch%3Fq%3Delei%25E7%25F5es%26site%3Ddatafolha%252Feleicoes%26sr%3D26%26skin%3Ddatafolha](http://search.folha.uol.com.br/search?q=elei%E7%F5es&site=datafolha%2Feleicoes&sr=1&skin=datafolha&results_count=516&search_time=0.255&url=http%3A%2F%2Fsearch.folha.uol.com.br%2Fsearch%3Fq%3Delei%25E7%25F5es%26site%3Ddatafolha%252Feleicoes%26sr%3D26%26skin%3Ddatafolha)

### Raspando uma notícia no site DataFolha

Antes de começar, vamos chamar a biblioteca `rvest` para tornar suas funções disponíveis em nossa sessão do R:

```
library(rvest)  
library(dplyr)
```

Nossa primeira tarefa será escolher uma única notícia (a primeira da busca, por exemplo), e extrair dela 4 informações de interesse: o título da notícia; a data e hora da notícia; o link para a pesquisa completa em .pdf; e o texto da notícia.

O primeiro passo é criar um objeto com endereço URL da notícia e outro que contenha o código HTML da página:

```
url <- "http://datafolha.folha.uol.com.br/eleicoes/2016/02/1744581-49-nao-votariam-em-lula.shtml"  
  
pagina <- xml2::read_html(url)
```

Felizmente, a função `read_html` já estrutura os dados de forma que o R seja capaz de identificar as estruturas de um HTML, como por exemplo, tags, atributos, valores e conteúdo das tags. Para olharmos a estrutura do HTML que nós lemos, podemos utilizar a função `xml_structure(pagina)` do pacote `xml2`

```
xml2::xml_structure(pagina)
```

Com o objeto XML preparado e representando a página com a qual estamos trabalhando, vamos à caça das informações que queremos.

Volte para a página da notícia. Procure o título da notícia e examine-o, inspecionando o código clicando com o botão direito do mouse e selecionando “Inspecionar”. Note o que encontramos:

```
<h1 class="main_color main_title"><!--TITULO-->49% não votariam em Lula<!--/TITULO--></h1>
```

Tente sozinh@ e por aproximadamente 1~2 minutos construir um “xpath” (caminho em XML) que nos levaria a este elemento antes de avançar. (Tente sozinh@ antes de copiar a resposta abaixo!)

A resposta é: “//h1[@class = ‘main\_color main\_title’]”

Usando agora as funções `rvest::html_node()` e `rvest::html_text()`, como vimos no tutorial anterior, vamos capturar o título da notícia:

```
titulo <- rvest::html_node(pagina ,  
                           xpath = '//h1[@class = "main_color main_title"]') %>%  
rvest::html_text()  
print(titulo)
```

Simples, não? Repita agora o mesmo procedimento para data e hora (tente sozinh@ antes de copiar a resposta abaixo!):

```
datahora <- rvest::html_node(pagina ,  
                           xpath = '//time') %>%  
rvest::html_text()  
print(datahora)
```

E também para o link do .pdf disponibilizado pelo DataFolha com o conteúdo completo da pesquisa – dica: o link é o valor do atributo “href” da tag “a” que encontramos ao inspecionar o botão para download:

```
pesquisa <- rvest::html_node(pagina ,  
                           xpath = '//p[@class = "stamp download"]/a') %>%  
rvest::html_attr("href")  
print(pesquisa)
```

Note que para obtermos o atributo “href” mudamos da função `html_text()` para `html_attr()`.

Finalmente, peguemos o texto. Note que o texto está dividido em vários parágrafos cujo conteúdo está inseridos em tags “p”, todas filhas da tag “article”. Se escolhemos o xpath sem especificar a tag “p” ao final, como abaixo, capturamos um monte de “sujeira”, como os botões de twitter e facebook.

```
texto <- rvest::html_node(pagina ,  
                           xpath = '//article[@class = "news"]') %>%  
rvest::html_text()  
print(texto)
```

Por outro lado, se especificamos a tag “p” ao final do xpath, recebemos um vetor contendo cada um dos parágrafos do texto. Precisaríamos “juntar” (concatenar) todos os parágrafos para formar um texto único.

```
texto <- rvest::html_nodes(pagina ,  
                           xpath = '//article[@class = "news"]/p') %>%  
rvest::html_text()  
print(texto)
```

Note que neste caso tivemos que utilizar `rvest::html_nodes()`! Não percebeu a diferença? Esta função tem um "s" no final, significa que ele vai raspar todos os xpath com o caminho `//article[@class = "news"]/p`, ou seja, caso tivessemos utilizado `rvest::html_node()`, nosso resultado seria apenas o primeiro parágrafo do texto.

Por simplicidade, usaremos a primeira opção. Ao final, construímos um código ligeiramente mais complexo do que esperamos para a atividade que dá conta deste pequeno problema.

### Sua vez - tente raspar a notícia seguinte na busca do DataFolha

Tente agora raspar a notícia seguinte usando a mesma estratégia. É fundamental notar que variamos a notícia, mas as informações continuam tendo o mesmo caminho. Essa é a propriedade fundamental do portal raspado que nos permite obter todas as notícias sem nos preocuparmos em abrir uma por uma. O link para a próxima notícia está no objeto "url" abaixo:

```
url<- "http://datafolha.folha.uol.com.br/eleicoes/2015/11/1701573-russomanno-larga-na-frente-em-disputa-pela-prefeitura-de-sao-paulo"
```

### Download de arquivos

Por vezes, queremos fazer download de um arquivo cujo link encontramos na página raspada. Por exemplo, no datafolha seria interessante obter o relatório em .pdf da pesquisa (para extrair seu conteúdo no futuro, por exemplo). Vamos ver como fazer download de um arquivo online.

Em primeiro lugar, obtemos seu endereço URL, como acabamos de fazer com a notícia que capturamos na busca do DataFolha (tente ler o código e veja se o entende por completo):

```
library(rvest)  
library(dplyr)  
url <- "http://datafolha.folha.uol.com.br/eleicoes/2016/02/1744581-49-nao-votariam-em-lula.shtml"  
pagina <- read_html(url)  
pesquisa <- rvest::html_nodes(pagina ,  
                           xpath = '//p[@class = "stamp download"]/a') %>%  
rvest::html_attr("href")
```

O link está no objeto "pesquisa":

```
print(pesquisa)
```

Usando a função `download.file`, rapidamente salvamos o link no "working directory" (use `getwd()` para descobrir qual é o seu) e com o nome "pesquisa.pdf" (poderíamos salvar com o nome que quisesssemos):

```
getwd()  
download.file(pesquisa, "pesquisa.pdf")
```

Caso você tenha problemas com essa função, talvez você tenha que substituir as barras "\\" por "/". Para isso, utilizaremos o `gsub`.

```
pesquisa2 <- gsub("\\\\/", "/", pesquisa)
```

```
download.file(pesquisa2, "pesquisa.pdf")
```

Vá ao “working directory” e veja o arquivo!

Sempre que estiver em posse de um conjunto de links que contém arquivos, você pode colocar a função `download.file` em loop e capturar todos os objetos ao mesmo tempo (por exemplo, na Câmara dos Deputados – vamos deixar um exemplo no github “leobarone”). Há uma dificuldade boba: nomear sem repetir os nomes diversos arquivos. Uma dica é usar o final do endereço URL como nome, mas você pode salvar os arquivos com nomes que sejam uma sequência numérica ou que provenham de um vetor que contenha os nomes todos. Use a criatividade!

Vamos voltar agora às notícias do DataFolha em HTML e ignorar o download de arquivos com os relatórios das pesquisas.

### Um código, duas etapas: raspando todas as notícias de eleições do DataFolha

Vamos fazer um breve roteiro do que precisamos fazer para criar um banco de dados que contenha todos os títulos, data e hora e texto de todas as notícias sobre eleições do DataFolha (Obs: por enquanto vamos ignorar os links de pesquisa, pois nem todas as notícias contêm os links e isso causa interrupção do código. Ao final, apresentamos um código que resolve tal problema).

#### *Etapa 1*

- Passo 1: conhecer a página de busca (e compreender como podemos “passar” de uma página para outra)
- Passo 2: raspar (em loop!) as páginas de busca para obter todos os links de notícia

Esta é a primeira etapa da captura. Em primeiro lugar temos que buscar todos os URLs que contêm as notícias buscadas. Em outras palavras, começamos obtendo “em loop” os links das notícias e, só depois de termos os links, obtemos o conteúdo destes links. Nossos passos seguintes, portanto, são:

#### *Etapa 2*

- Passo 3: conhecer a página da notícia (e ser capaz de obter nela as informações desejadas). Já fizemos isso acima!
- Passo 4: raspar (em um novo loop!) o conteúdo dos links capturados no Passo 2.

Vamos construir o código da primeira etapa da captura e, uma vez resolvida a primeira etapa, faremos o código da segunda.

#### *Código da etapa 1*

Em primeiro lugar, vamos observar o URL da página de busca (poderíamos buscar termos chave, mas, neste caso, vamos pegar todas as notícias relacionadas a eleições). Na página 2 da busca vemos que o final é “sr=26”. Na página 3 o final é “sr=51”. Há um padrão: as buscas são realizadas de 25 em 25. De fato, a 21a. é última página da busca. Para “passarmos” de página em página, portanto, temos que ter um “loop” que conte não mais de 1 até 21, mas na seguinte sequência numérica: {1, 26, 51, 76, ..., 476, 501}.

Parece difícil, mas é extremamente simples. Veja o loop abaixo, que imprime a sequência desejada multiplicando  $(i - 1)$  por 25 e somando 1 ao final:

```
for (i in 1:21){  
  i <- (i - 1) * 25 + 1  
  print(i)  
}
```

Vamos, dessa forma, criar o objeto “url\_base” a partir do URL da página 2 e substituir o número 26 em “sr=26” por um “place holder”, “CONTADORLINK”, por exemplo:

```
url_base <- "http://search.folha.uol.com.br/search?q=elei%E7%F5es&site=datafolha%2Feleicoes&skin=datafolha&results
```

Capturar os links das notícias de uma única página é simples: examinamos o código HTML, lemos no R, transformamos em um objeto XML ("parse") procuramos o "xpath" que caracteriza os links e extraímos o valor do atributo "href". Este seria o Passo 1 descrito acima. Veja abaixo.

```
pagina <- read_html(url)

link <- rvest::html_nodes(pagina, xpath = '//h2[@class = "title"]/a') %>%
  rvest::html_attr("href")
```

Combinando o que vimos até agora, podemos executar o Passo 2. Faltaria apenas criar antes do loop um vetor vazio – por exemplo, o vetor "links\_datafolha" no código abaixo – que, a cada iteração do loop "guarda" os links raspados da página. Sua tarefa é gastar MUITOS minutos no código abaixo para entendê-lo na totalidade.

```
links_datafolha <- c()
for (i in 1:21){
  print(i)
  i <- (i - 1) * 25 + 1
  url <- stringr::str_replace(url_base, "CONTADORLINK", as.character(i))
  pagina <- read_html(url)
  link <- rvest::html_nodes(pagina,
    xpath = '//h2[@class = "title"]/a') %>%
    rvest::html_attr("href")
  links_datafolha <- c(links_datafolha, link)
}
```

Temos, ao final, o objeto `links_datafolha` que contém todos os links para as notícias sobre eleições no DataFolha. Encerramos com sucesso a Etapa 1 – caracterizada pelo primeiro loop. Esta etapa se assemelha bastante ao que fizemos nas atividades 1 e 2 e o código deve ser compreensível para você a essa altura do campeonato. Vamos agora iniciar a etapa 2.

#### *Código da etapa 2*

No começo da atividade resolvemos a captura do título, data e hora, link para o relatório de pesquisa completa e texto para uma única notícia no portal do instituto DataFolha. Nos resta agora capturar, em loop, o conteúdo de cada uma das páginas cujos links estão guardados no vetor "links\_datafolha".

Vamos rever o procedimento, para uma URL qualquer, da captura do título, data e hora e texto (vamos deixar o link para o relatório de pesquisa de lado por enquanto, posto que algumas notícias não contêm o link e esta pequena ausência interromperia o funcionamento do código).

```
pagina <- read_html("http://datafolha.folha.uol.com.br/eleicoes/2016/02/1744581-49-nao-votariam-em-lula.shtml")

titulo <- rvest::html_node(pagina ,
  xpath = '//h1[@class = "main_color main_title"]') %>%
  rvest::html_text()
datahora <- rvest::html_node(pagina , xpath = '//time') %>%
  rvest::html_text()

texto <- rvest::html_node(pagina , xpath = "//article[@class = 'news ']") %>%
  rvest::html_text()
```

Para fazermos a captura de todos os links em “loop” deve ter o seguinte aspecto, como se vê no código abaixo que imprime todos os 516 links cujo conteúdo queremos capturar. Note que a forma de utilizar o loop é ligeiramente diferente da que havíamos visto até então. No lugar de uma variável “i” que “percorre” um vetor numérico (1:21, por exemplo), temos uma variável “link” que recebe, a cada iteração, um endereço URL do vetor “links\_datafolha”, em ordem. Assim, na primeira iteração temos que “link” será igual “links\_datafolha[1]”, na segunda “links\_datafolha[2]” e assim por diante até a última posição do vetor “links\_datafolha” – no nosso caso a posição 516.

```
for (link in links_datafolha){
  print(link)
}
```

Combinando os dois código, e criando um data frame “dados” que é vazio antes do loop temos o código completo da captura. Tal como quando trabalhamos com tabelas, utilizando a função “rbind” para combinar o data frame que resultou da iteração anterior com a linha que combina o conteúdo armazenado em “titulo”, “datahora” e “texto”.

```
dados <- data.frame()

for (link in links_datafolha){
  print(link)
  pagina <- read_html(link)
  titulo <- rvest::html_node(pagina, xpath = '//h1[@class = "main_color main_title"]') %>%
    rvest::html_text()
  datahora <- rvest::html_node(pagina, xpath = '//time') %>%
    rvest::html_text()
  texto <- rvest::html_node(pagina, xpath = "//article[@class = 'news']") %>%
    rvest::html_text()
  dados <- rbind(dados, data.frame(titulo, datahora, texto))
}
```

O resultado do código é um data frame (“dados”) que contém 3 variáveis em suas colunas: “titulo”, “datahora” e “texto”. A partir de agora você poderia, por exemplo, usar as ferramentas presentes no pacote “tm” da linguagem R (“tm” é acronimo de “text mining”) para criar uma nuvem de palavras (“wordcloud”), fazer a contagem de termos, examinar a semelhança da linguagem usada pelo instituto DataFolha com a usada por outros institutos de opinião pública, fazer análise de sentimentos, etc.

Antes disso, sua tarefa é a seguinte: executar ambas as etapas do código e comentá-lo por completo (use # para inserir linhas de comentário). Comentar o código alheio é uma excelente maneira de ver se você conseguiu comprehendê-lo por completo e serve para você voltar ao código no futuro quando for usá-lo de modelo para seus próprios programas em R.

*EXTRA: versão do código com links para pesquisa e com texto “limpo”*

```
nullToNA <- function(x) {
  if (is.null(x)){
    return(NA)
  } else {
    return(x)
  }
}

texto_vetor <- rvest::html_nodes(pagina ,
  xpath = '//article[@class = "news"]/p') %>%
  rvest::html_text()
texto <- c()
for (paragrafo in texto_vetor){
  texto <- paste(texto, paragrafo)
```

```
}
```

```
texto <- nullToNA(texto)
```

© 2019 Released under the MIT license – Documentation built with [Hugo](#) using the [Material](#) them...