Programação em R

Julho, 2016

O que veremos

Vamos dar um foco adicional na manipulação de data frames, pois esta é uma atividade que será bastante recorrente. Em geral, você gasta a maior parte do seu tempo arrumando, explorando e colocando os dados em uma forma adequada para análise. Veremos:

- Uma breve revisão e algumas novas funções (adicionar, remover colunas, subset etc);
- A estratégia Dividir, Aplicar e Combinar com funções nativas do R e `dplyr`;
- Como colocar seus dados no formato ideal com o pacotes `reshape2` e `tidyr`

Manipulação de data.frames: revisão e algumas funções

Carregando a base de dados

Iremos trabalhar com dados de oferta online de imóveis no plano piloto no mês de agosto de 2014.

```
arquivo <- url("https://dl.dropboxusercontent.com/u/44201187/dados.rds")
con <- gzcon(arquivo)
imoveis <- readRDS(con)
close(con)
```

Carregando a base de dados

Vendo a estrutura da base:

```
str(imoveis, vec.len = 1)
## 'data.frame':    366666 obs. of  13 variables:
##  $ bairro   : chr  "Park Sul" ...
##  $ location : chr  "SMAS APARTAMENTO SOMENTE PARA DIÁRIA/TEMPORADA !!!" ...
##  $ preco    : num  400 600 ...
##  $ quartos  : num  3 1 ...
##  $ m2       : num  95 30 ...
##  $ corretora: chr  "21900" ...
##  $ data     : Date, format: "2014-08-01" ...
##  $ link     : chr  "/imovel/aluguel-apartamento-brasilia-df-3-quartos-smas-1223980" ...
##  $ estado   : chr  "df" ...
##  $ cidade   : chr  "brasilia" ...
##  $ imovel   : chr  "apartamento" ...
##  $ tipo     : chr  "aluguel" ...
##  $ coleta   : Date, format: "2014-08-01" ...
```

Renomeando Colunas

Note que a segunda coluna, que contém o endereço do apartamento, está com o nome `location`. Vamos renomeá-la para `endereco`.

```
names(imoveis)[2]
## [1] "location"
names(imoveis)[2] <- "endereco"
names(imoveis)[2]
## [1] "endereco"
```

Adicionando colunas

Note que não há uma coluna com preço por metro quadrado. Vamos adicionar esta coluna:

```
imoveis$pm2 <- imoveis$preco/imoveis$m2
```

Uma forma conveniente de realizar isto é com a função `with`:

```
imoveis$pm2 <- with(imoveis, preco/m2)
str(imoveis$pm2)
##  num [1:366666] 4.21 20 20 21.67 28.38 ...
```

Head e Tail

A base de dados é relativamente grande (mais de 300 mil obs) então é preciso tomar cuidado para não ficar imprimindo os valores na tela. Para "espionar" os dados, duas funções convenientes são `head` e `tail`. Os resultados são omitidos, pois são muito grandes para caber no slide.

```
# head - primeiras observações
head(imoveis) # mostra 6 primeiras linhas
imoveis[1:6,] # equivalente
head(imoveis, 10) # mostra 10 primeiras linhas
imoveis[1:10,] # equivalente

# tail - últimas observações
tail(imoveis) # mostra as 6 últimas linhas
imoveis[(nrow(imoveis)-5):nrow(imoveis),] # equivalente
tail(imoveis, 10) # mostra as 10 últimas linhas
imoveis[(nrow(imoveis)-10):nrow(imoveis),] # equivalente
```

Subset

Vamos revisar brevemente algumas formas de selecionar linhas e colunas do data.frame:

```
# com números
imoveis[1,] #primeira linha, todas as colunas
imoveis[,1] #primeira coluna, todas as linhas
imoveis[1,1] #primeira linha e primeira coluna
imoveis[c(2,4,6), c(1,3:5)] #linhas 2, 4 e 6, e colunas 1,3,4,5.
imoveis[1:10,-(1:5)] #linhas 1:10, exclui colunas 1 a 5
head(imoveis[-c(1,2,5),]) # exclui linhas 1, 2 e 5

# com nomes
imoveis[imoveis$bairro == "Asa Sul", c("data", "bairro", "preco")]
subset(imoveis, bairro == "Asa Sul", select=c(data, bairro, preco))
```

Subset

E comparar as diferentes formas de selecionar uma coluna:

```
precos1 <- imoveis$preco # vetor
precos2 <- imoveis[ ,"preco"]  # vetor
precos3 <- imoveis[["preco"]]  # vetor
precos4 <- imoveis["preco"]  # data.frame
precos5 <- subset(imoveis, select=preco)  # data.frame
precos6 <- imoveis[,"preco", drop=FALSE] # data.frame

str(precos1)
##  num [1:366666] 400 600 600 650 650 680 700 700 700 700 ...
str(precos4)
## 'data.frame':    366666 obs. of  1 variable:
##  $ preco: num  400 600 600 650 650 680 700 700 700 700 ...
rm(list=ls(pattern="precos"))
```

Filtrando com índices lógicos

Vamos passar alguns filtros no data.frame. Como selecionar os imóveis que estavam à venda no dia 31 de Agosto que foram anunciados a mais de 1 milhão de reais?

```
indice <- with(imoveis, coleta=="2014-08-31" &
                        tipo=="venda" &
                        preco > 1e6)
sum(indice) # quantos imóveis?
## [1] 3826
dia31_venda_maior1m <- imoveis[indice,] # filtrando
```

Função subset

A mesma operação com `subset`:

```
dia31_venda_maior1m <- subset(imoveis,
                              coleta == "2014-08-31" &
                              tipo == "venda" &
                              preco > 1e6)
```

Criando subgrupos com cut

Note que a variável m2 é uma variável numérica contínua. Mas podemos querer analisar esta variável em categorias discretas, como até 50m2, de 50m2 a 100m2. A função `cut` serve para isso.

```
imoveis$m2_cat <-
  cut(imoveis$m2,
      breaks= c(0, 50, 150, 200, 250, Inf),
      labels=c("de 0 a 50 m2", "de 50 a 150 m2",
               "de 150 a 200 m2","de 200 a 250 m2",
               "mais de 250 m2") )

str(imoveis$m2_cat)
##  Factor w/ 5 levels "de 0 a 50 m2",..: 2 1 1 1 1 1 1 1 1 1 ...
```

Unique

A função unique retorna as observações únicas de um vetor. Vejamos com mais detalhes quais são os valores de algumas colunas de nossa base de dados.

```
unique(imoveis$tipo)
## [1] "aluguel" "venda"
unique(imoveis$imovel)
## [1] "apartamento"    "casa"           "kitinete"       "loja"
## [5] "sala-comercial"
unique(imoveis$bairro)
##  [1] "Park Sul"                          "Asa Norte"
##  [3] "Lago Norte"                        "Lago Sul"
##  [5] "Vila Planalto"                     "Sudoeste"
##  [7] "Asa Sul"                           "Granja do Torto"
##  [9] "Noroeste"                          "Octogonal"
## [11] "Setor Habitacional Jardim Botânico" "Vila Telebrasília"
## [13] "Brasília"                          "Park Way"
## [15] "Taquari"
```

Unique

Quantas obserções únicas de imóveis temos neste 1 mês de coleta? Considere o **link** do anúncio como seu identificador único.

```
unique(imoveis$coleta) # quais os dias de coleta únicos?
##  [1] "2014-08-01" "2014-08-02" "2014-08-03" "2014-08-04" "2014-08-05"
##  [6] "2014-08-06" "2014-08-07" "2014-08-08" "2014-08-09" "2014-08-12"
## [11] "2014-08-13" "2014-08-14" "2014-08-15" "2014-08-16" "2014-08-17"
## [16] "2014-08-18" "2014-08-19" "2014-08-20" "2014-08-21" "2014-08-22"
## [21] "2014-08-23" "2014-08-24" "2014-08-25" "2014-08-26" "2014-08-27"
## [26] "2014-08-28" "2014-08-29" "2014-08-30" "2014-08-31"
length(unique(imoveis$link)) # quantas observações únicas?
## [1] 16635
```

duplicated

Vamos eliminar os duplicados para trabalhar com as observações únicas do mês. A função `duplicated` retorna um vetor lógico com TRUE para uma observação que apareceu anteriormente.

```
duplicados <- duplicated(imoveis$link)
unicos <- imoveis[!duplicados, ]
```

Eliminando NA's

Uma função de conveniência para eliminar NA de um data.frame é a `na.omit`. Mas, **cuidado**, ela irá eliminar todas as observações que cotenham ao menos um NA em alguma coluna.

```
unicos <- na.omit(unicos)
```

Exercícios

Sua vez.
Para cada uma das bases (`imoveis` e `unicos`), quantos registros únicos temos para cada coluna? Para cada uma das bases (`imoveis` e `unicos`), quantos NA's temos para cada coluna?

Soluções

Lembra que o `sapply` aplica uma função a todos os elementos do objeto?

```
sapply(imoveis, function(x) length(unique(x)))
sapply(unicos, function(x) length(unique(x)))
sapply(imoveis, function(x) sum(is.na(x)))
sapply(unicos, function(x) sum(is.na(x)))
```

Dividir, Aplicar e Combinar

Um padrão recorrente

Nossa base de dados contém preços tanto de aluguel quanto de venda de apartamentos. Suponha que queiramos tirar a médias dos preços. Não faz muito sentido tirar a média dos dois tipos (aluguel e venda) juntos, certo? Como poderíamos fazer isso então?
Uma possível solução seria fazer o seguinte:
Primeiramente, dividimos nossa base em duas outras para cada grupo: aluguel e venda.

```
# 1) separar a base em duas bases diferentes:
#    - aluguel; e,
#    - venda
aluguel <- unicos[unicos$tipo == "aluguel", ]
venda <- unicos[unicos$tipo == "venda", ]
```

Um padrão recorrente

Em seguida nós calculamos a média para cada uma das bases que criamos.

```
# 2) calcular a média para cada uma das bases
media_aluguel <- mean(aluguel$preco)
media_venda   <- mean(venda$preco)
```

Por fim, nós combinamos os resultados em um único vetor:

```
# 3) combinar os resultados em um único vetor
medias = c(aluguel = media_aluguel, venda = media_venda)
medias
## aluguel   venda
##   29672 1440468
```

Um padrão recorrente

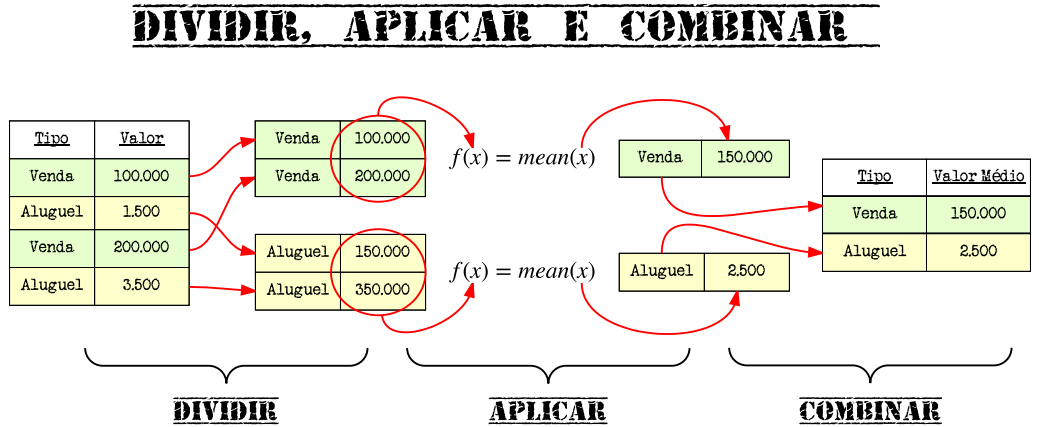
Nós gastamos cerca de 5 linhas para chegar ao resultado. E se eu te dissesse que podemos fazer isso (ou coisas mais complexas) em apenas uma ou duas linhas? Para você saber onde vamos chegar, seguem alguns exemplos:

```
# com tapply
tapply(X = unicos$preco, INDEX = unicos$tipo, FUN = mean)

# com aggregate
aggregate(x = list(media = unicos$preco), by = list(tipo = unicos$tipo),
          FUN = mean)

# com dplyr
library(dplyr)
unicos %>% group_by(tipo) %>% summarise(media = mean(preco))
```

Dividir, Aplicar e Combinar (Split, Apply and Combine)

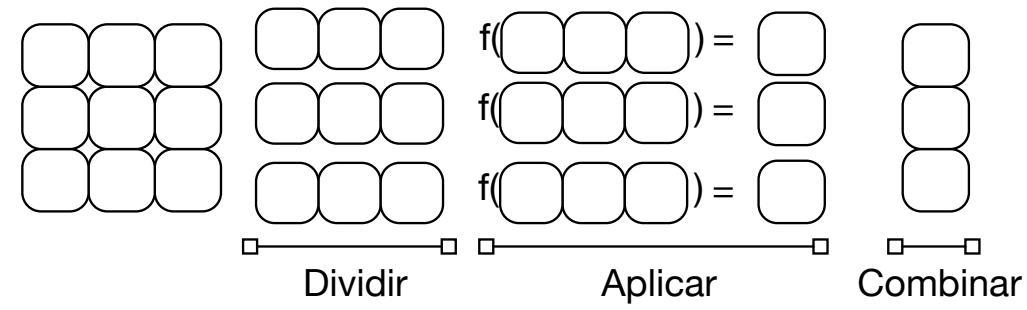
O padrão de análise que fizemos anteriormente é bastante recorrente em análise de dados. Dentro da comunidade do R é conhecido como "Dividir, Aplicar e Combinar" (DAC) ou, em inglês, "Split, Apply and Combine" (SAC). Neste caso específico, nós pegamos um vetor (o vetor de preços), dividimos segundo algum critério (por tipo), aplicamos um função em cada um dos grupos separadamente (no nosso caso, a média) e depois combinamos os resultados novamente. Para quem conhece SQL, muitas dessas operações são similares ao `group by`, ou, para quem usa Excel, similar a uma tabela dinâmica (mas não equivalentes, o conceito aqui é mais flexível).

Dividir, Aplicar e Combinar (Split, Apply and Combine)



Dividir, Aplicar e Combinar (Split, Apply and Combine)

Nós já vimos este padrão diversas vezes com as funções do tipo apply. Por exemplo, ao aplicar uma função por linhas, você está *dividindo* a matriz por uma das dimensões, *aplicando* funções a cada uma das partes e *combinando* os resultados em um vetor:



Vejamos agora algumas peças para construir essa estratégia de análise de dados usando as funções base.

Dividir: a função split

```
str(split)
## function (x, f, drop = FALSE, ...)
```

Dividir: a função split

```
str(split)
## function (x, f, drop = FALSE, ...)
```

- **x**: é o vetor ou data.frame que será divido;
- **f**: são os fatores que irão definir os grupos de divisão.

Dividir: a função split

```
str(split)
## function (x, f, drop = FALSE, ...)
```

- **x**: é o vetor ou data.frame que será divido;
- **f**: são os fatores que irão definir os grupos de divisão.

O resultado da função é uma lista com os vetores ou data.frames dos grupos.

Dividir: a função split

No nosso exemplo:

```
# 1) Dividir o data.frame segundo uma lista de fatores
alug_venda <- split(unicos$preco, unicos$tipo)

# Note o resultado
# temos na lista alug_venda dois vetores
# um para aluguel
# e outro para venda
str(alug_venda, max.level = 1)
## List of 2
##  $ aluguel: num [1:5246] 400 600 600 650 650 680 700 700 700 700 ...
##  $ venda  : num [1:11383] 750 845 159000 170000 175000 180000 180000 18
```

Aplicar e combinar - voltando à família apply

Como visto, o resultado de split é uma lista para cada categoria. Agora queremos aplicar uma função a cada um dos elementos dessa lista. Já vimos uma função que faz isso: `lapply()`:

```
medias <- lapply(alug_venda, mean)
medias
## $aluguel
## [1] 29672
##
## $venda
## [1] 1440468
```

Aplicar e combinar - voltando à família apply

Mas o lapply nos dá uma lista e queremos um vetor. Então agora queremos simplificar o resultado. Uma das formas seria utilizar uma função que vocês já aprenderam: `unlist()`

```
unlist(medias)
## aluguel   venda
##   29672 1440468
```

Aplicar e combinar - voltando à família apply

Existe uma função de conveniência que, em conjunto com `rbind()` e `cbind()` pode ser bastante útil para simplificar resultados: a função `do.call()`. Como ela funciona?

```
do.call("alguma_funcao", lista_de_argumentos) =
  alguma_funcao(lista_de_argumentos[1],
                lista_de_argumentos[2],
                  ...,
                lista_de_argumentos[n])
```

Aplicar e combinar - voltando à família apply

No nosso caso temos apenas duas médias, então não seria complicado elencar um por um os elementos no `rbind()` ou no `cbind()`.
Todavia, imagine que tívessemos centenas de médias. Neste casso a função `do.call()` é bastante conveniente.

```
do.call("rbind", medias)
##            [,1]
## aluguel   29672
## venda   1440468

do.call("cbind", medias)
##      aluguel   venda
## [1,]   29672 1440468
```

Aplicando e simplificando ao mesmo tempo

Agora podemos encaixar conceitualmente outra função que já tínhamos aprendido, o `sapply()`. Essa função tenta fazer os dois passos ao mesmo tempo: aplicar e combinar (que neste caso é *simplificar*):

```
sapply(alug_venda, mean)
## aluguel   venda
##   29672 1440468
```

Aplicando e simplificando ao mesmo tempo

Como vimos, existe, ainda, uma versão mais restrita do `sapply()`: o `vapply()`. Lembrando que a principal diferença entre eles é que o `vapply()` exige que você especifique o formato do resultado esperado da operação. Enquanto o o primeiro é mais prático para uso interativo, o segundo é mais seguro para programar suas próprias funções, pois se o resultado não vier conforme esperado ele irá acusar o erro.

```
vapply(alug_venda, mean, numeric(1))
## aluguel   venda
##   29672 1440468

vapply(alug_venda, mean, character(1))
## Error in vapply(alug_venda, mean, character(1)): valores devem ser do tipo 'character',
##  mas o resultado de FUN(X[[1]]) é de tipo 'double'
```

Antes de prosseguir... outliers

Aplicando a função `summary()` para cada elemento de alug_venda:

```
# Aplicando a função summary para cada elemento de alug_venda
lapply(alug_venda, summary)
## $aluguel
##      Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
##         0      1100      2000     29700      3700 120000000
##
## $venda
##      Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
##         0    450000    850000   1440000   1450000 995000000
```

Antes de prosseguir... outliers

Note a clara presença de **outliers**, possivelmente dados errados. Então, antes de prosseguir, façamos o seguinte: vamos "limpar" o data.frame `unicos` retirando os valores muito discrepantes de preço por metro quadrado (Para venda, valores abaixo de 3000 ou acima de 20000. Para aluguel, valores abaixo de 25 ou acima de 60). Depois vamos salvar o resultado em um data.frame chamado `limpos`.

Antes de prosseguir... outliers

```
# filtro para venda
ok.venda <- with(unicos, tipo == "venda" &
                         pm2 > 3000 &
                         pm2 < 20000)

# filtro para aluguel
ok.aluguel <- with(unicos, tipo == "aluguel" &
                           pm2 > 25 &
                           pm2 < 100)
# juntando os dois
ok <- (ok.venda | ok.aluguel)

# separando outliers e limpos
outliers <- unicos[!ok,]
limpos <- unicos[ok,]
```

Dividir, Aplicar e Combinar: tapply

A função `tapply()` tem a seguinte estrutura:

```
str(tapply)
## function (X, INDEX, FUN = NULL, ..., simplify = TRUE)
```

- **X**: o objeto que será agregado. *Ex: preços*;
- **INDEX**: uma lista de vetores que servirão de índice para argegar o objeto. *Ex: bairros*;
- **FUN**: a função que será aplicada a X para cada INDEX. *Ex: mediana*.
- **simplify**: tentará simplificar o resultado para uma estrutura mais simples?

Dividir, Aplicar e Combinar: tapply

Vejamos alguns exemplos: calcular a mediana do metro quadrado para aluguel e para venda:

```
tapply(limpos$pm2, limpos$tipo, median)
## aluguel   venda
##      36    8983
```

Agora para aluguel e venda, separado por bairro:

```
tapply(limpos$pm2, list(limpos$bairro, limpos$tipo), median)
##                                   aluguel venda
## Asa Norte                              35  9005
## Asa Sul                                39  9167
## Brasília                               NA 11844
## Granja do Torto                        NA  4138
## Lago Norte                             34  6826
## Lago Sul                               33  5500
## Noroeste                               38  9654
## Octogonal                              29  8652
## Park Sul                               42  9544
```

Dividir, Aplicar e Combinar: tapply

Outro exemplo: qual a mediana do preço por metro quadrado dos apartamentos, separados aluguel, venda e número de quartos?

```
aps <- limpos[limpos$imovel == "apartamento", ]

with(aps, tapply(pm2, list(quartos, tipo), median))
##    aluguel venda
## 1       37  9790
## 2       33  9048
## 3       30  9127
## 4       32 10194
## 5       32 10204
## 6       NA  5481
## 11      NA  3580
```

Dividir, Aplicar e Combinar: aggregate

O aggregate é similar ao tapply mas, ao invés de retornar um array, retorna um data.frame com uma coluna para cada índice e apenas uma coluna de valor.
A função aggregate tem duas sintaxes principais.
A primeira, similar ao tapply é:

```
aggregate(dados$valor, by = list(dados$indice1, dados$indice2), funcao)
```

Já a segunda sintaxe utiliza a *formula interface* do R e é do tipo:

```
aggregate(valor ~ indice1 + indice2, dados, funcao)
```

Dividir, Aplicar e Combinar: aggregate

Exemplo: calculando a mediana do preço por metro quadrado, separada por bairro, venda ou aluguel, e tipo de imóvel. Note a diferença do formato deste resultado para o formato do `tapply`.

```
pm2_bairro_tipo_imovel <- aggregate(formula = pm2 ~ bairro + tipo + imovel,
                                    data = limpos,
                                    FUN = median)
head(pm2_bairro_tipo_imovel)
##       bairro    tipo      imovel pm2
## 1  Asa Norte aluguel apartamento  31
## 2    Asa Sul aluguel apartamento  31
## 3 Lago Norte aluguel apartamento  34
## 4   Lago Sul aluguel apartamento  29
## 5   Noroeste aluguel apartamento  37
## 6  Octogonal aluguel apartamento  29
```

Dividir, Aplicar e Combinar: aggregate

Você também pode passar mais de uma variável a ser agregada. Vamos calcular a mediana do preço, metro quadrado e preço por metro quadrado dos valores de aluguel de apartamento. Note que tudo isso pode ser passado diretamente ao aggregate.

```
mediana_aluguel <- aggregate(cbind(preco, m2, pm2) ~ bairro,
                             data = limpos,
                             subset = (tipo == "aluguel" &
                                       imovel == "apartamento"),
                             FUN = median)

mediana_aluguel[order(mediana_aluguel$pm2, decreasing = TRUE), ]
##          bairro preco m2 pm2
## 7      Park Sul  2500 44  41
## 5      Noroeste  2675 75  37
## 3    Lago Norte  1800 55  34
## 8      Sudoeste  2700 80  33
## 1     Asa Norte  2300 70  31
## 2       Asa Sul  2700 76  31
```

dplyr: Eficiente e intuitivo

Com as funções da família `apply` e similares, você consegue fazer praticamente tudo o que você precisa para explorar os dados e deixá-los no(s) formato(s) necessário(s) para análise. E é importante você ser exposto a essas funções para se familiarizar com o ambiente R.
Entretanto, muitas vezes essas funções podem **deixar a desejar em performance** e existe um pacote **bastante rápido** para manipulação de data.frame e com **sintaxe muito intuitiva** chamado `dplyr`. É provável que para o grosso de suas necessidades o `dplyr` seja a solução mais rápida e eficiente.
Vejamos.

dplyr: Funções principais

- **filter**: filtra um data.frame com vetores lógicos. Por exemplo, filtrar valores de pm2 menores ou maiores do que determinado nível.
- **select**: seleciona uma ou mais colunas de um data.frame. Por exemplo, selecionar a coluna de preços.
- **mutate**: cria uma nova coluna. Por exemplo, criar a coluna pm2 como preco/m2.
- **arrange**: orderna o data.frame com base em uma coluna. Por exemplo, ordernar do maior ao menor pm2.
- **group_by**: agrupa um data.frame por índices. Por exemplo, agrupar os dados de imóveis por bairro e número de quartos.
- **summarise**: geralmente utilizado após o group_by. Calcula valores por grupo. Por exemplo, tirar a média ou mediana do preço por bairro.

dplyr: Conectando tudo com %>%

O `dplyr` vem também com o *pipe operator* `%>%` do pacote `magritrr`. Basicamente, o operador faz com que você possa escrever `x %>% f()` ao invés de `f(x)`. Na prática, isso tem uma grande utilidade: você vai poder escrever o código de manipulação dos dados da mesma forma que você pensa nas atividades.

Ex: pegue a base de dados `limpos`, filtre apenas os dados coletados de apartamento, selecione as colunas bairro e preco, crie uma coluna `pm2 = preco/m2`, ordene os dados de forma decrescente em pm2 e mostre apenas as 6 primeiras linhas (head).

```
library(dplyr)
limpos %>% filter(imovel == "apartamento") %>%
  select(bairro, preco, m2) %>% mutate(pm2 = preco/m2) %>%
  arrange(desc(pm2)) %>% head()
```

dplyr: Agrupando e sumarizando

Pegue a base de dados `limpos`, filtre apenas os dados de venda de apartamento. Agrupe os dados por bairro. Calcule as medianas do preço, m2 e pm2 e o número de observações. Filtre apenas os grupos com mais de 30 observações. Ordene de forma decrescente com base na mediana de pm2.

```
limpos %>%
  filter(imovel == "apartamento", tipo == "venda") %>%
  group_by(bairro) %>%
  summarise(Mediana_Preco = median(preco),
            Mediana_M2 = median(m2),
            Mediana_pm2 = median(pm2),
            Obs = length(pm2)) %>%
  filter(Obs > 30) %>%
  arrange(desc(Mediana_pm2))
```

dplyr: Voltando um pouco aos textos

Vamos usar a nossa base de dados para fazer uma busca de apartamentos.
Apartamento na Asa Sul ou Asa Norte, entre 101-404, para aluguel, com preço menor do que R$ 2.000,00.

```
consulta <-
  imoveis %>% filter(grepl("SQ(N|S) (4|3|2|1)0(1|2|3|4)", endereco),
                     tipo == "aluguel",
                     bairro %in% c("Asa Sul", "Asa Norte"),
                     preco < 2200,
                     coleta == "2014-08-31")
```

Melhor do que a busca do *wimoveis*.

Exercícios

Sua vez.
Considerando a base de dados `limpos`, Responda:

- Qual o bairro com o maior preço mediano de venda?
- Qual o bairro com o maior preço por m2 mediano de venda?
- Qual o bairro com o maior preço mediano de venda para apartamentos?
- Qual o bairro com o maior preço mediano de venda para lojas?

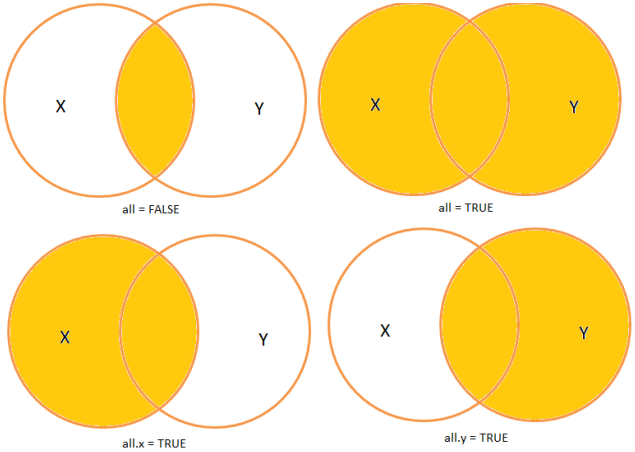
Colocando seus dados em forma

Merge

A função `merge()` serve para combinar data.frames. A função tenta identificar quais são as colunas identificadores em comum entre dois data frames para realizar a combinação. Para quem conhece SQL, a função `merge()` é equivalente ao `join`. Alguns argumentos da função são:

- **x**: um data.frame
- **y**: um data.frame
- **by**: a coluna em comum nos data.frames pela qual será feito o merge.
- **all**, **all.x**, **all.y**: especifica o tipo do merge. O default é FALSE e é equivalente ao "natural join" do SQL; "all" é equivalente ao "outer join"; "all.x" é equivalente ao "left outer join" e "all.y" é equivalente ao "right outer join".

Merge



Merge

Vejamos um exemplo. Vamos usar o aggregate para calcular a mediana do pm2 separadamente para aluguel e para venda. Depois vamos usar o merge para juntar os dados:

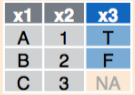
```
dados1 <- aggregate(formula = pm2 ~ bairro,
                    data = limpos,
                    subset = (limpos$tipo == "aluguel"),
                    FUN = median)

dados2 <- aggregate(formula = pm2 ~ bairro,
                    data = limpos,
                    subset = (limpos$tipo == "venda"),
                    FUN = median)

names(dados1)[2] <- "aluguel"
names(dados2)[2] <- "venda"
```

Merge

```
merge_all_false <- merge(dados1, dados2, all = FALSE)

merge_all_true <- merge(dados1, dados2, all = TRUE)

str(merge_all_false) # 11 linhas
## 'data.frame':    11 obs. of  3 variables:
##  $ bairro : chr  "Asa Norte" "Asa Sul" "Lago Norte" "Lago Sul" ...
##  $ aluguel: num  34.9 38.7 34.5 33.3 37.8 ...
##  $ venda  : num  9005 9167 6826 5500 9654 ...

str(merge_all_true) # 15 linhas
## 'data.frame':    15 obs. of  3 variables:
##  $ bairro : chr  "Asa Norte" "Asa Sul" "Brasília" "Granja do Torto" ...
##  $ aluguel: num  34.9 38.7 NA NA 34.5 ...
##  $ venda  : num  9005 9167 11844 4138 6826 ...
```

Joins do dplyr

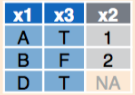
O dplyr também vem com funções de merge (join).



x1	x2	x3
A	1	T
B	2	F
C	3	NA

dplyr::**left_join(a, b, by = "x1")**

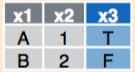
Join matching rows from b to a.



x1	x3	x2
A	T	1
B	F	2
D	T	NA

dplyr::**right_join(a, b, by = "x1")**

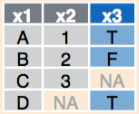
Join matching rows from a to b.



x1	x2	x3
A	1	T
B	2	F

dplyr::**inner_join(a, b, by = "x1")**

Join data. Retain only rows in both sets.



x1	x2	x3
A	1	T
B	2	F
C	3	NA
D	NA	T

dplyr::**full_join(a, b, by = "x1")**

Join data. Retain all values, all rows.

Joins do dplyr

Alguns exemplos:

```
# igual merge com all = FALSE
innerjoin <- inner_join(dados1, dados2)
## Joining by: "bairro"

# igual merge com all = TRUE
fulljoin <- full_join(dados1, dados2)
## Joining by: "bairro"
```

Wide x Long

A tabela do `tapply` seria um exemplo do que chamamos de tabela no formato *wide*: as colunas representam grupos.
Já o formato do `aggregate` ou do `dplyr` é o que chamamos de formato **long**: há várias colunas identificadoras e apenas uma coluna de valores.
É bastante comum usar dados no formato **long** em pacotes de visualização, como o `ggplot2` ou `lattice`. O formato **wide**, por sua vez, é bastante utilizado em display de tabelas.
Assim, muitas vezes queremos passar nossos dados de um formato para o outro.

reshape2: Melt

Transforma dados no formato **wide** para o formato **long**:

```
# dados no formato wide
ap_wide <- tapply(aps$preco, list(aps$bairro, aps$m2_cat), median)
ap_wide[1:2, 1:3]
##           de 0 a 50 m2 de 50 a 150 m2 de 150 a 200 m2
## Asa Norte       260000         669999          1730000
## Asa Sul         350000         680000          1450000

#carrega o pacote e transforma em long
library(reshape2)
ap_long <- melt(ap_wide)
head(ap_long, 2)
##        Var1         Var2  value
## 1 Asa Norte de 0 a 50 m2 260000
## 2   Asa Sul de 0 a 50 m2 350000
```

reshape2: Cast - dcast

Transforma dados no formato **long** para o formato **wide**. Existe duas funções de **cast**: **dcast** e **acast** sendo que a primeira retorna um data.frame e a segunda um array. Vejamos o **dcast**:

```
long <- aggregate(pm2 ~ bairro + tipo + imovel + quartos,
                  data = limpos,
                  median)
head(long, 2)
##      bairro    tipo   imovel quartos pm2
## 1 Asa Norte aluguel kitinete       0  32
## 2   Asa Sul aluguel kitinete       0  35

wide <- dcast(data = long,
              formula = imovel + quartos ~ tipo + bairro,
              value.var = "pm2", sum)
wide[1:2, 1:4]
##        imovel quartos aluguel_Asa Norte aluguel_Asa Sul
## 1 apartamento       1                33               44
## 2 apartamento       2                33               30
```

reshape2: Cast - acast

Para mais dimensões do que duas, é preciso usar um *array*:

```
# note que cada dimensão é separada por ~
# para um data.frame só podemos ter um ~
# para arrays podemos ter vários
cast2 <- acast(long,
               imovel~quartos~tipo~bairro,
               value.var="pm2", sum)

str(cast2, vec.len=2)
##  num [1:5, 1:13, 1:2, 1:15] 0 0 ...
##  - attr(*, "dimnames")=List of 4
##   ..$ : chr [1:5] "apartamento" "casa" ...
##   ..$ : chr [1:13] "0" "1" ...
##   ..$ : chr [1:2] "aluguel" "venda"
##   ..$ : chr [1:15] "Asa Norte" "Asa Sul" ...
```

tidyr: reshape novo para data.frames

O `tidyr` é um pacote novo para fazer diversas tarefas para arrumar seus dados, entre elas transformar do formato **wide** para o formato **long** e vice-versa. Entretanto, só serve para data.frames. Vejamos alguns exemplos:

```
library(tidyr)
library(dplyr)
tidy_wide <- long %>%
  spread(imovel, pm2)

head(tidy_wide)
##      bairro    tipo quartos apartamento casa kitinete loja sala-comercial
## 1 Asa Norte aluguel       0          NA   NA       32   48             47
## 2 Asa Norte aluguel       1          33   NA       33   35             34
## 3 Asa Norte aluguel       2          33   NA       37   36             NA
## 4 Asa Norte aluguel       3          29   29       NA   NA             NA
## 5 Asa Norte aluguel       4          32   NA       NA   NA             NA
## 6 Asa Norte aluguel       5          NA   NA       NA   NA             50
```

tidyr: reshape novo para data.frames

Agora fazendo a operação inversa:

```
tidy_long <- tidy_wide %>%
  gather(imovel, pm2, -bairro, - tipo, - quartos)

head(tidy_long)
##      bairro    tipo quartos      imovel pm2
## 1 Asa Norte aluguel       0 apartamento  NA
## 2 Asa Norte aluguel       1 apartamento  33
## 3 Asa Norte aluguel       2 apartamento  33
## 4 Asa Norte aluguel       3 apartamento  29
## 5 Asa Norte aluguel       4 apartamento  32
## 6 Asa Norte aluguel       5 apartamento  NA
```

Há outras funções interessantes no **tidyr** como `unite()`, `separate()` e `extract()`.

Exemplo

Vamos calcular sua média de buscas no Google por hora, dia da semana, mês...