Insper

Revisão e tidymodels

Aula 1

Magno Severino
PADS - Aprendizagem Estatística de Máquina II

Programa

Alguns dos tópicos que serão discutidos nesta disciplina:

- utilizar `tidymodels` para acelerar o processo de modelagem;
- mineração de texto (text mining);
- redução de dimensionalidade, análise de componentes principais;
- k-médias;
- análise de agrupamento hierárquico;
- escalonamento multidimensional;
- redes neurais e aplicações;
- inteligência artificial generativa (genAI).

Referências bibliográficas do curso

- An Introduction to Statistical Learning: with Applications in R. James, G. and Witten, D. and Hastie, T. and Tibshirani, R. 2021.
- The Elements of Statistical Learning. Hastie, T. and Tibshirani, R. and Friedman, J. 2017.
- R for Data Science Wickham, H. and Grolemund, G. 2017.
- Data Science, Marketing & Business. Fernandez, P., Marques, P. 2019.
- Aprendizado de Máquina, uma abordagem estatística. Izbicki, R. and Santos, T. 2020.
- Tidy modeling with R Kuhn, M. and Silge, Julia. 2021.

Critérios de avaliação

- Atividades práticas: 30%.
- Projeto de análise de dados:
  - Entrega 1: 30%,
  - Entrega 2: 60%.

Objetivos de aprendizagem da aula de hoje

Ao final dessa aula você deverá ser capaz de

- relembrar todo o processo de definição de modelagem de dados,
- compreender o que são dados *tidy*,
- realizar todo o processo de modelagem utilizando o pacote `tidymodels`.

CLASSICAL MACHINE LEARNING

Data is pre-categorized or numerical

SUPERVISED

Predict a category

CLASSIFICATION

«Divide the socks by color»

Predict a number

REGRESSION

«Divide the ties by length»

Data is not labeled in any way

UNSUPERVISED

Divide by similarity

CLUSTERING

«Split up similar clothing into stacks»

Find hidden dependencies

DIMENSION REDUCTION (generalization)

«Make the best outfits from the given clothes»

Identify sequences

ASSOCIATION

«Find what clothes I often wear together»

Tidy data

- É possível representar um mesmo conjunto de dados de diferentes maneiras.
- Veja a seguir.

```
library(tidyverse)
table1
```

country	year	cases	population
Afghanistan	1999	745	19987071
Afghanistan	2000	2666	20595360
Brazil	1999	37737	172006362
Brazil	2000	80488	174504898
China	1999	212258	1272915272
China	2000	213766	1280428583

Tidy data

```
table2
```

country	year	type	count
Afghanistan	1999	cases	745
Afghanistan	1999	population	19987071
Afghanistan	2000	cases	2666
Afghanistan	2000	population	20595360
Brazil	1999	cases	37737
Brazil	1999	population	172006362
Brazil	2000	cases	80488
Brazil	2000	population	174504898
China	1999	cases	212258
China	1999	population	1272915272
China	2000	cases	213766
China	2000	population	1280428583

Tidy data

```
table3
```

country	year	rate
Afghanistan	1999	745/19987071
Afghanistan	2000	2666/20595360
Brazil	1999	37737/172006362
Brazil	2000	80488/174504898
China	1999	212258/1272915272
China	2000	213766/1280428583

Tidy data

```
table4a # cases
```

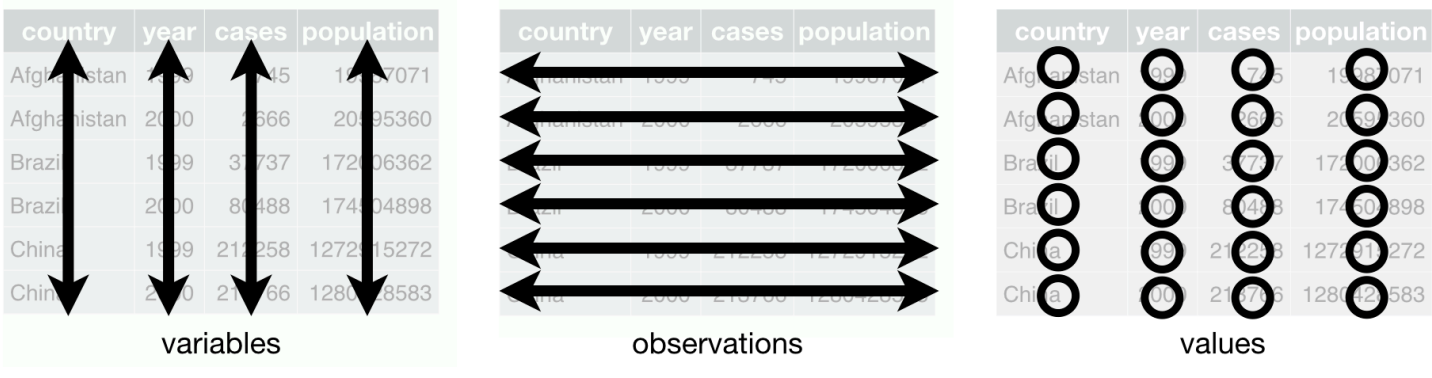
country	1999	2000
Afghanistan	745	2666
Brazil	37737	80488
China	212258	213766

```
table4b # population
```

country	1999	2000
Afghanistan	19987071	20595360
Brazil	172006362	174504898
China	1272915272	1280428583

Tidy data

- Todas as tabelas anteriores representam o mesmo dado, mas nem todas podem ser usadas de maneira fácil.
- Dados organizados em formato *tidy* são facilmente usados no contexto do `tidyverse`.
- Regras que definem um conjunto de dados *tidy*:
  - Cada variável deve ter sua própria coluna;
  - Cada observação deve ter sua própria linha;
  - Cada valor deve ter a sua própria célula.



- Qual/quais das tabelas apresentadas anteriormente está/estão no formato *tidy*?

[1] Figura do capítulo Tidy data do livro R for Data Science Wickham, H. and Grolemund, G. 2017.

Tidy data

- Os dados em `table1` são os únicos que estão no formato tidy.
- Veja como é simples calcular a taxa de casos por população:

```
table1 %>%
  mutate(rate = cases / population * 10000)
```

country	year	cases	population	rate
Afghanistan	1999	745	19987071	0.372741
Afghanistan	2000	2666	20595360	1.294466
Brazil	1999	37737	172006362	2.193931
Brazil	2000	80488	174504898	4.612363
China	1999	212258	1272915272	1.667495
China	2000	213766	1280428583	1.669488

Desafio: obter as taxas para as tabelas 2, 3 e 4.

Pivotando linhas e colunas

- Grande parte dos dados **não** estão no formato tidy.
- Você deve identificar o que são variáveis e o que são observações.
- Em seguida, resolver os seguintes possíveis problemas:

    1. Uma variável está separada em múltiplas colunas;
    2. Uma observação está em múltiplas linhas.

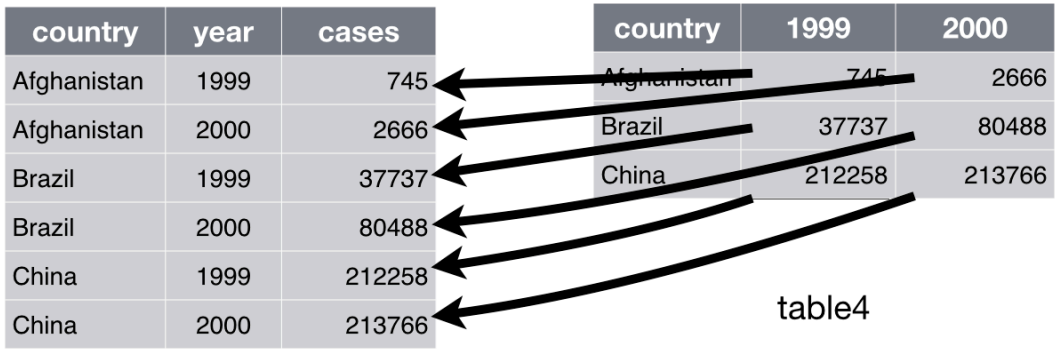
- As funções `pivot_longer()` e `pivot_wider()` te ajudam a resolver estes problemas.

Pivot longer

Relembre a `table4a`. Os nomes das colunas `1999` e `2000` representam *valores* da variável **ano**. Os valores nas colunas `1999` e `2000` representam *valores* da variável **total de casos**.

country	1999	2000
Afghanistan	745	2666
Brazil	37737	80488
China	212258	213766

O que devemos fazer para deixar essa tabela no formato tidy?



[1] Figura do capítulo Tidy data do livro R for Data Science Wickham, H. and Grolemund, G. 2017.

Pivot longer

```
table4a %>%
  pivot_longer(cols = -country,
               names_to = "year",
               values_to = "cases")
```

country	year	cases
Afghanistan	1999	745
Afghanistan	2000	2666
Brazil	1999	37737
Brazil	2000	80488
China	1999	212258
China	2000	213766

Pivot wider

O `pivot_wider()` faz o oposto de `pivot_longer()`. Na `table2`, uma observação é o par país-ano, mas cada observação está dividida em duas linhas.

country	year	type	count
Afghanistan	1999	cases	745
Afghanistan	1999	population	19987071
Afghanistan	2000	cases	2666
Afghanistan	2000	population	20595360
Brazil	1999	cases	37737
Brazil	1999	population	172006362
Brazil	2000	cases	80488
Brazil	2000	population	174504898
China	1999	cases	212258
China	1999	population	1272915272
China	2000	cases	213766
China	2000	population	1280428583

Pivot wider

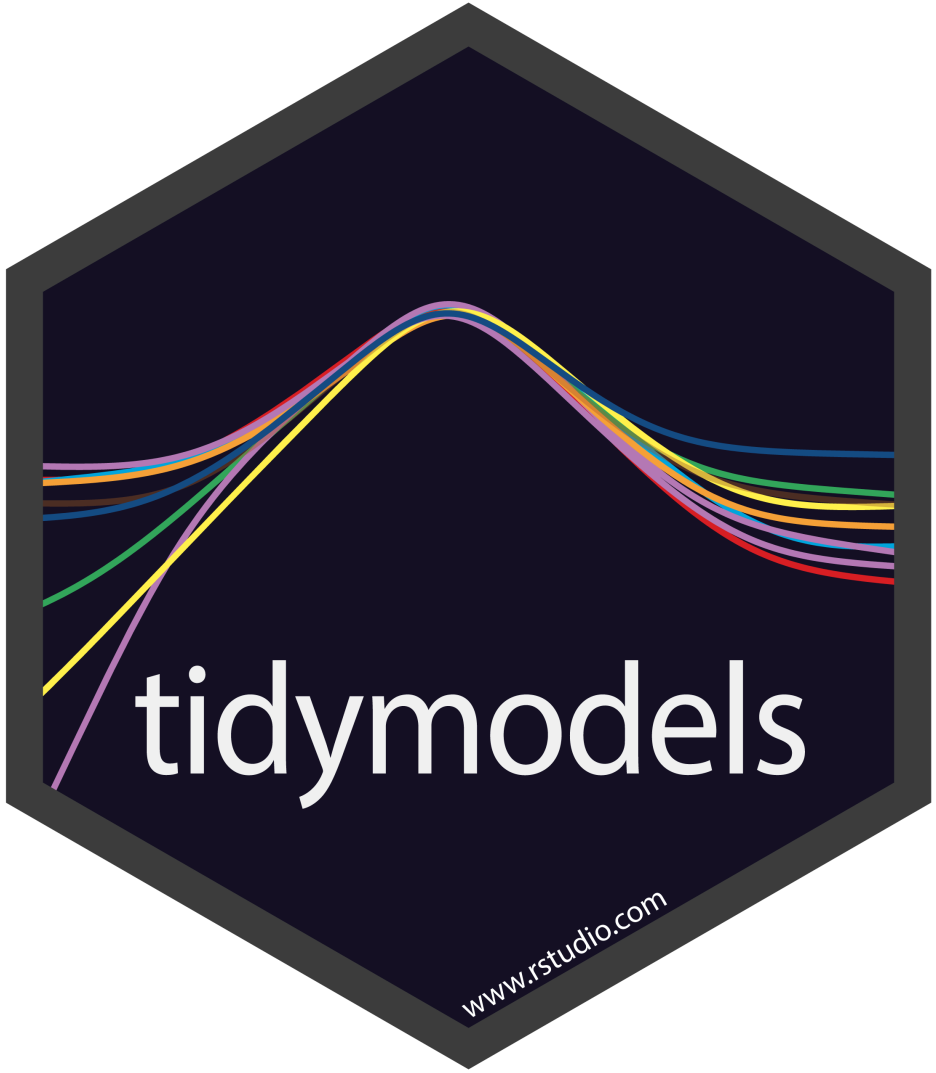
```
table2 %>%
    pivot_wider(names_from = type,
                values_from = count)
```

country	year	cases	population
Afghanistan	1999	745	19987071
Afghanistan	2000	2666	20595360
Brazil	1999	37737	172006362
Brazil	2000	80488	174504898
China	1999	212258	1272915272
China	2000	213766	1280428583

Resumindo...

- Dados em formato *tidy* facilitam a sua manipulação.
- Definição de dados em formato *tidy*:
  - Cada variável deve ter sua própria coluna;
  - Cada observação deve ter sua própria linha;
  - Cada valor deve ter a sua própria célula.
- As funções `pivot_longer()` e `pivot_wider()` podem de ajudar a deixar dados em formato *tidy*.

Tidymodels



- É uma coleção de pacotes para processamento de dados, definição de modelos, ajustes de hiperparâmetros e avaliação de desempenho de modelos.
- Fonte de informação: tidymodels.org.
- Referência: livro Tidy modeling with R.

Pacotes que fazem parte do tidymodels

- **rsample**: métodos de reamostragem e validação cruzada;
- **recipes**: processamento dos dados;
- **parsnip**: definição dos modelos;
- **tune**: ajuste de hiperparâmetros;
- **yardstick**: cálculo de métricas de desempenho.

Dados Credit

Como exemplo, vamos usar os dados `Credit`.

```
library(ISLR)
library(tidyverse)
library(tidymodels)
Credit
```

ID	Income	Limit	Rating	Cards	Age	Education	Gender	Student	Married	Ethnicity	B
1	14.891	3606	283	2	34	11	Male	No	Yes	Caucasian	
2	106.025	6645	483	3	82	15	Female	Yes	Yes	Asian	
3	104.593	7075	514	4	71	11	Male	No	No	Asian	
4	148.924	9504	681	3	36	11	Female	No	No	Asian	
5	55.882	4897	357	2	68	16	Male	No	Yes	Caucasian	
6	80.180	8047	569	4	77	10	Male	No	No	Caucasian	
7	20.996	3388	259	2	37	12	Female	No	No	African American	
8	71.408	7114	512	2	87	9	Male	No	No	Asian	
9	15.125	3300	266	5	66	13	Female	No	No	Caucasian	
10	71.061	6819	491	3	41	19	Female	Yes	Yes	African American	

rsample

- Contém um conjunto de funções que podem aplicar diferentes métodos de reamostragem.
- Esses métodos podem ser utilizados por diferentes pacotes para:
  - técnicas de reamostragem tradicionais para estimar a distribuição amostral de uma estatística;
  - estimar o desempenho de um modelo utilizando um conjunto *holdout*.
- Vantagem: não ocupa memória de forma desnecessária!

> *Note that resampled data sets created by rsample are directly accessible in a resampling object but do not contain much overhead in memory. Since the original data is not modified, R does not make an automatic copy. For example, creating 50 bootstraps of a data set does not create an object that is 50-fold larger in memory.*

- Mais detalhes em https://tidymodels.github.io/rsample/.

Treinamento e teste com `rsample`

Para os dados `Credit`, podemos separá-los em conjuntos de treinamento e teste da seguinte maneira.

```
set.seed(15)

split <- initial_split(Credit, prop = 0.8)
split

treinamento <- training(split) # treinamento
teste <- testing(split) # teste

treinamento %>% slice(1:3)
```

ID	Income	Limit	Rating	Cards	Age	Education	Gender	Student	Married	Ethnicity	E
37	62.413	6457	455	2	71	11	Female	No	Yes	Caucasian	
362	53.217	4943	362	2	46	16	Female	No	Yes	Asian	
162	31.353	1705	160	3	81	14	Male	No	Yes	Caucasian	

Validação cruzada com `rsample`

Objetivo: estimar o erro de validação cruzada de um modelo linear para a previsão do `Balance` considerando dois modelos:

- Modelo 1: utiliza somente as variáveis `Income` e `Limite`;
- Modelo 1: utiliza somente as variáveis `Rating` e `Age`.

Para isso, vamos considerar validação cruzada em 10 lotes usando o pacote `rsample`.

```
# definição de validação cruzada em 10 lotes
vfold_cv(Credit, v = 10)
```

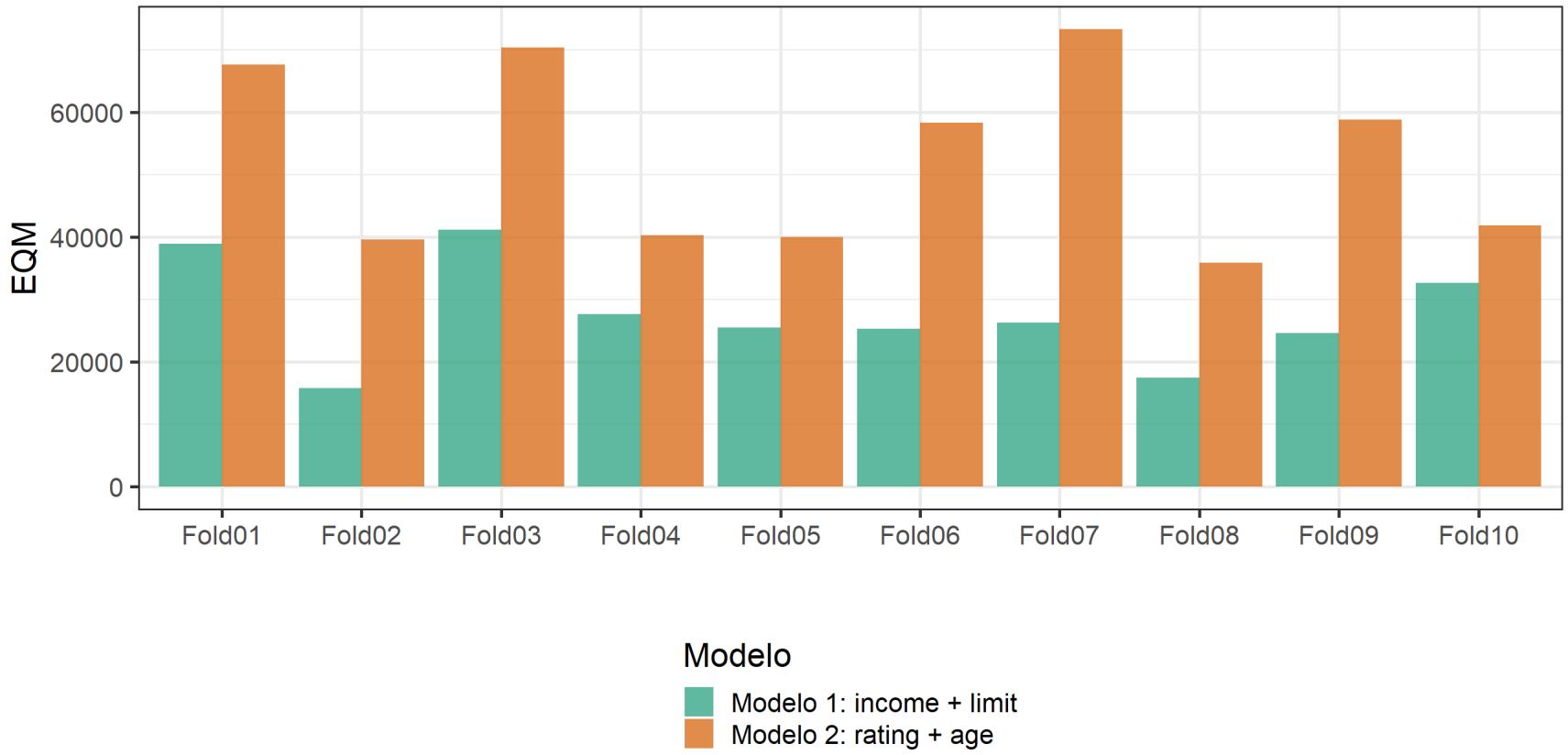
```
## #  10-fold cross-validation
## # A tibble: 10 × 2
##    splits           id
##    <list>           <chr>
##  1 <split [360/40]> Fold01
##  2 <split [360/40]> Fold02
##  3 <split [360/40]> Fold03
##  4 <split [360/40]> Fold04
##  5 <split [360/40]> Fold05
##  6 <split [360/40]> Fold06
##  7 <split [360/40]> Fold07
##  8 <split [360/40]> Fold08
##  9 <split [360/40]> Fold09
## 10 <split [360/40]> Fold10
```

Validação cruzada com `rsample`

```
erro <- function(split){
  tr <- training(split)
  tst <- testing(split)

  fit_1 <- lm(Balance ~ Income + Limit, data = tr)
  fit_2 <- lm(Balance ~ Rating + Age, data = tr)

  tibble(eqm1 = Metrics::mse(tst$Balance,
                             predict(fit_1, tst)),
         eqm2 = Metrics::mse(tst$Balance,
                             predict(fit_2, tst)))
}

set.seed(123)
vfold_cv(Credit, v = 10) %>%
  mutate(erro = map(splits, erro)) %>%
  unnest(erro) %>%
  summarise_if(is.numeric, mean)
```

```
## # A tibble: 1 × 2
##     eqm1   eqm2
##    <dbl>  <dbl>
## 1 27562. 52664.
```

Validação cruzada com `rsample`



Processamento com `recipes`

- Com o `recipes` é possível definir sequências como as do `dplyr` com o *pipe* (`%>%`) para realizar passos de engenharia de *features* para processar os dados antes da modelagem e visualização.
- A ideia desse pacote é definir uma receita que possa ser utilizada para definir sequencialmente as codificações e pré processamentos dos dados.
- O processo é realizado da seguinte forma
  - `recipe()`: especifica o pré-processamento. Recebe informação dos dados, mas não recebe os dados.
  - `prep()`: estima os parâmetros que poderão ser utilizados para aplicar o processamento em outro conjunto de dados futuramente.
  - `bake()`: aplica a receita a um determinado banco de dados.
- Em https://tidymodels.github.io/recipes/ há mais detalhes e exemplos.

Processamento com `recipes`

Para criar uma receita, utilizamos a função `recipe()`

```
receita <- recipe(Balance ~ ., data = treinamento)

receita
```

```
## Recipe
##
## Inputs:
##
##       role #variables
##    outcome           1
##  predictor          11
```

Processamento com `recipes`

O processamento é feito através de funções do tipo **step_...()**, por exemplo:

- **step_dummy**: converte variáveis nominais ou categóricas em variáveis indicadoras;
- **step_log**: aplica logaritmo;
- **step_meaninput**, **step_medianinput**, **step_modeinput**: imputa os dados faltantes por uma das medidas (média, mediana e moda) do conjunto de treinamento;
- **step_normalize**: padroniza as variáveis para ter média zero e desvio padrão igual a um;
- **step_rm**: remove variáveis;
- **step_zv**: remove variavels que contém apenas um único valor;
- **step_BoxCox**: aplica transformação de Box-Cox;
- **step_discretize**: converte os dados númericos em fatores com níveis de tamanhos amostrais aproximadamente iguais.

A lista completa de funções disponíveis estão aqui.

Processamento com `recipes`

Para normalizar a variável `Income`, por exemplo, basta utilizar a função `step_normalize`.

```
receita <- recipe(Balance ~ ., data = treinamento) %>%
  step_rm(ID) %>%
  step_normalize(Income)

receita
```

```
## Recipe
##
## Inputs:
##
##       role #variables
##    outcome           1
##  predictor          11
##
## Operations:
##
## Variables removed ID
## Centering and scaling for Income
```

Observe que neste ponto a receita foi **definida**, mas não **aplicada**.

Processamento com `recipes`

A função `prep` prepara a receita.

```
receita <- recipe(Balance ~ ., data = treinamento) %>%
  step_rm(ID) %>%
  step_normalize(Income) %>%
  prep()

receita
```

```
## Recipe
##
## Inputs:
##
##       role #variables
##    outcome           1
##  predictor          11
##
## Training data contained 320 data points and no missing data.
##
## Operations:
##
## Variables removed ID [trained]
## Centering and scaling for Income [trained]
```

Processamento com `recipes`

- Não é necessário sempre listar todas as variáveis à qual uma função **step_...** deve ser aplicada.
- As funções abaixo permitem selecionar variáveis que tem certos papeis/características:
  - `all_predictors()`: todas as variáveis preditoras;
  - `all_outcomes()`: todas as variáveis resposta;
  - `all_numeric()`: todas as variáveis numéricas;
  - `all_nominal()`: todas as variáveis nominais.
- Podemos utilizar junto com o sinal de subtração também: `-all_numeric()`.
- Qual receita o código abaixo define?

```
receita <- recipe(Balance ~ ., data = treinamento) %>%
    step_rm(ID) %>%
    step_normalize(all_numeric(), -all_outcomes()) %>%
    step_other(Ethnicity, threshold = .3, other = "outros") %>%
    step_dummy(all_nominal(), -all_outcomes())

receita_prep <- prep(receita)
```

Processamento com `recipes`

A função `bake` aplica a receita em um novo conjunto de dados. Para obter os dados de treinamento, utilize o argumento `new_data=NULL`.

```
tr_proc <- bake(receita_prep, new_data = NULL)

tst_proc <- bake(receita_prep, new_data = teste)

tr_proc
```

Income	Limit	Rating	Cards	Age	Education	Balance	Gender_Female	S
0.5026300	0.7564359	0.6596327	-0.6989599	0.9099906	-0.8085101	762	1	
0.2432040	0.1140014	0.0715360	-0.6989599	-0.5270258	0.8085101	382	1	
-0.3735957	-1.2599769	-1.2058355	0.0202271	1.4847972	0.1617020	0	0	
2.7103745	2.7690299	2.9487835	2.8969754	-0.5270258	-1.4553181	1677	0	
-0.5796187	-0.8772319	-0.8200946	0.7394142	-1.3317549	1.4553181	0	0	
0.6584662	0.2162647	0.2043320	0.7394142	0.4501454	-0.4851060	345	0	
-0.8214976	-0.8114608	-0.8580363	0.0202271	0.2777034	1.4553181	52	0	
0.6660548	0.8951916	0.9378720	1.4586012	-0.8719097	0.8085101	1411	1	
-0.5200376	0.1810454	0.1980084	0.0202271	0.3926647	1.7787221	710	1	
-0.7157356	-1.3753945	-1.4461331	0.0202271	-0.9868710	0.1617020	0	1	

Processamento com `recipes`

Veja o antes e depois do processamento da variável `Ethnicity`.

```
treinamento %>%
  count(Ethnicity) %>%
  mutate(Porcentagem = 100*n/sum(n))
```

Ethnicity	n	Porcentagem
African American	74	23.12
Asian	91	28.44
Caucasian	155	48.44

```
tr_proc %>%
  count(Ethnicity_outros) %>%
  mutate(Porcentagem = 100*n/sum(n))
```

Ethnicity_outros	n	Porcentagem
0	155	48.44
1	165	51.56

Modelagem

Para ajustar um modelo linear utilizamos a função `lm` e um **dataframe** como argumento.

```
fit_lm <- lm(resposta ~ ., dados)
```

Para ajustar um LASSO, ridge ou elastic-net, utilizamos a função `glmnet` e uma matriz com as preditoras e um vetor com a variável resposta.

```
X <- model.matrix(resposta ~ ., dados)
y <- dados$resposta
fit_lasso <- glmnet(X, y, alpha = 1) # alpha = 1: lasso
```

As funções podem ter diferentes interfaces e argumentos.

Modelagem com `parsnip`

O objetivo do `parsnip` é

- Separar a definição de um modelo de sua avaliação;
- Dissociar a especificação do modelo de sua implementação. Por exemplo, podemos utilizar `rand_forest` no lugar de `ranger` ou `randomForest`.
- Uniformizar o nome dos argumentos (por exemplo, `n.trees`, `ntrees`, `trees`) para facilitar a utilização.
- Mais detalhes em https://tidymodels.github.io/parsnip/.

Modelagem com `parsnip` - modelo linear

O código abaixo define o modelo de regressão linear utilizando `lm` como *engine*.

```
lm <- linear_reg() %>%
  set_engine("lm")
lm
```

```
## Linear Regression Model Specification (regression)
##
## Computational engine: lm
```

O modelo ainda **não** foi ajustado! Fizemos apenas uma especificação.

Modelagem com `parsnip` - modelo linear

Para ajustar um modelo, temos que utilizar a função `fit`.

```
lm_fit <- linear_reg() %>%
  set_engine("lm") %>%
  fit(Balance ~ ., tr_proc)
lm_fit
```

```
## parsnip model object
##
##
## Call:
## stats::lm(formula = Balance ~ ., data = data)
##
## Coefficients:
##      (Intercept)            Income             Limit            Rating             Cards
##          476.867          -266.684           382.502           236.530            23.760
##              Age         Education     Gender_Female       Student_Yes       Married_Yes
##          -11.591            -5.607            -5.603           409.687            -7.785
## Ethnicity_outros
##            1.520
```

Modelagem com `parsnip` - modelo linear

Uma forma prática de obter informações sobre o modelo é utilizando a função `tidy` do pacote `broom`.

```
tidy(lm_fit)
```

```
## # A tibble: 11 × 5
##    term              estimate std.error statistic   p.value
##    <chr>                <dbl>     <dbl>     <dbl>     <dbl>
##  1 (Intercept)        477.       12.4      38.3   1.96e-119
##  2 Income            -267.        9.67    -27.6   2.71e- 85
##  3 Limit              383.       88.1       4.34  1.93e-  5
##  4 Rating             237.       88.4       2.68  7.85e-  3
##  5 Cards               23.8       6.79      3.50  5.40e-  4
##  6 Age                -11.6       5.80     -2.00  4.67e-  2
##  7 Education           -5.61      5.71     -0.982 3.27e-  1
##  8 Gender_Female       -5.60     11.4      -0.493 6.22e-  1
##  9 Student_Yes        410.       20.1      20.4   3.92e- 59
## 10 Married_Yes         -7.78     11.8      -0.657 5.12e-  1
## 11 Ethnicity_outros     1.52     11.4       0.134 8.94e-  1
```

Modelagem com `parsnip` - modelo linear

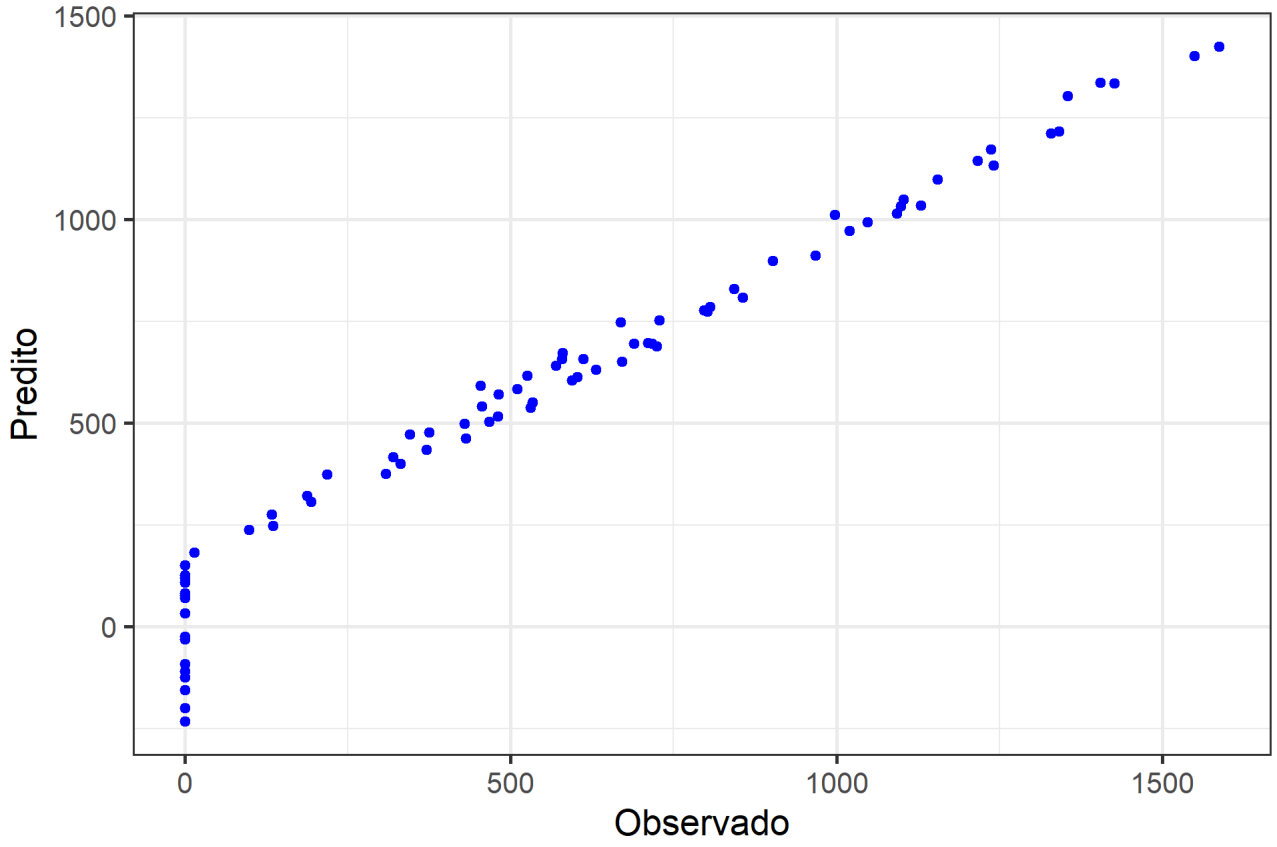
Uma vez que temos o modelo ajustado, podemos utilizá-lo para fazer previsões.

```
fitted_lm <- lm_fit %>%
  predict(new_data = tst_proc) %>%
  mutate(observado = tst_proc$Balance,
         modelo = "lm")

head(fitted_lm)
```

```
## # A tibble: 6 × 3
##   .pred observado modelo
##   <dbl>     <int> <chr>
## 1 672.        580 lm
## 2 400.        331 lm
## 3  71.7         0 lm
## 4 127.          0 lm
## 5 912.        968 lm
## 6 120.          0 lm
```

Modelagem com `parsnip` - modelo linear

```
fitted_lm %>%
  ggplot(aes(observado, .pred)) +
  geom_point(size = 2, col = "blue") +
  labs(y = "Predito", x = "Observado")
```



Modelagem com `parsnip`

engine	mode	model
C5.0	classification	boost_tree()
C5.0	classification	decision_tree()
glm	classification	logistic_reg()
glmnet	classification	logistic_reg()
glmnet	classification	multinom_reg()
glmnet	regression	linear_reg()
kknn	regression	nearest_neighbor()
kknn	classification	nearest_neighbor()
lm	regression	linear_reg()
randomForest	classification	rand_forest()
randomForest	regression	rand_forest()
ranger	classification	rand_forest()
ranger	regression	rand_forest()
rpart	classification	decision_tree()
rpart	regression	decision_tree()
xgboost	classification	boost_tree()
xgboost	regression	boost_tree()

A lista completa está neste link.

Modelagem com `parsnip` - floresta aleatória

Agora vamos ajustar uma floresta aleatória com o pacote `ranger`. O primeiro passo é definir o modelo.

```
rf <- rand_forest() %>%
  set_engine("ranger", importance = "permutation") %>%
  set_mode("regression")

rf
```

```
## Random Forest Model Specification (regression)
##
## Engine-Specific Arguments:
##   importance = permutation
##
## Computational engine: ranger
```

Modelagem com `parsnip` - floresta aleatória

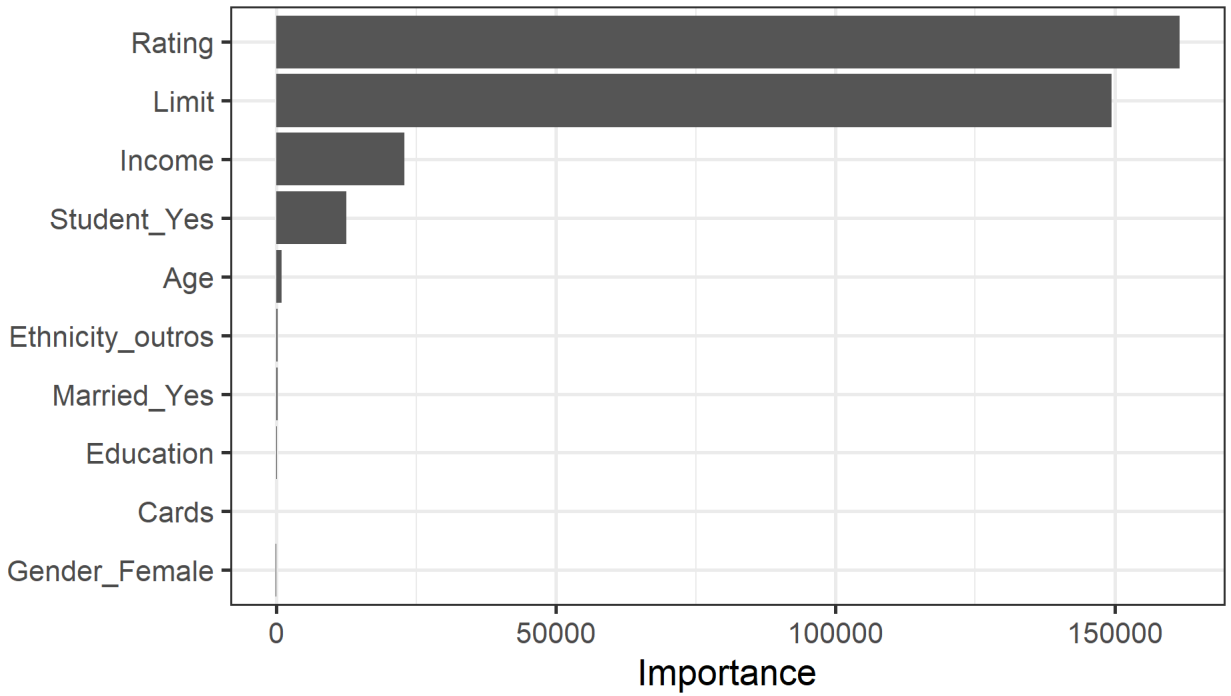
Com o modelo definido, podemos utilizar os dados para treiná-lo.

```
## parsnip model object
##
## Ranger result
##
## Call:
##  ranger::ranger(x = maybe_data_frame(x), y = y, importance = ~"permutation",      num.threads = 1, v
##
## Type:                             Regression
## Number of trees:                  500
## Sample size:                      320
## Number of independent variables:  10
## Mtry:                             3
## Target node size:                 5
## Variable importance mode:         permutation
## Splitrule:                        variance
## OOB prediction error (MSE):       20209.55
## R squared (OOB):                  0.9044498
```

Modelagem com `parsnip` - floresta aleatória

Podemos verificar a importância das variaveis com o pacote `vip`.

```
library(vip)

vip(rf_fit)
```



Modelagem com `parsnip` - floresta aleatória

Uma vez que temos o modelo ajustado, podemos utilizá-lo para fazer previsões.

```
fitted_rf <- rf_fit %>%
  predict(new_data = tst_proc) %>%
  mutate(observado = tst_proc$Balance,
         modelo = "random forest")

fitted <- fitted_lm %>%
  bind_rows(fitted_rf)

tail(fitted)
```

```
## # A tibble: 6 × 3
##    .pred observado modelo
##    <dbl>     <int> <chr>
## 1 341.         371 random forest
## 2 905.        1129 random forest
## 3 891.         806 random forest
## 4   3.08         0 random forest
## 5 411.         480 random forest
## 6  13.8          0 random forest
```

Avaliando desempenho com `yardstick`

- O pacote `yardstick` é utilizado para avaliar o desempenho dos modelos utilizando princípios de dados `tidy`.
- Esse pacote pode ser utilizado para calcular diversas métricas para modelos de regressão (rmse, rsq, mae, etc) e classificação (acurácia, roc_auc, precision, etc).
- Mais informações em https://tidymodels.github.io/yardstick/.

Metricas para regressão

- RMSE (root mean squared erro) - está na mesma unidade de medida dos dados

$$\text{rmse} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}.$$

- RSQ (R squared) é dado pela correlação entre o valor observado e o predito

$$\text{rsq} = \text{corr}^2(y, \hat{y}).$$

- MAE (mean absolute error) é dado pelo erro absoluto médio - está na mesma unidade de medida dos dados

$$\text{mae} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|.$$

Metricas para classificação

	Observado	
Classificado	**No**	**Yes**
No	a	b
Yes	c	d

Erro de classificação total: $\frac{b+c}{n} = 1 - \frac{a+d}{n}$;

Verdadeiro positivo (sensibilidade ou recall): $\frac{d}{b+d}$;

Verdadeiro negativo (especificidade): $\frac{a}{a+c}$;

Valor preditivo positivo (precision): $\frac{d}{c+d}$;

Valor preditivo negativo: $\frac{a}{a+b}$;

F-score: $2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}}$.

Avaliando desempenho com `yardstick`

Podemos obter diferentes métricas a partir do objeto `fitted` que construimos com as predições de cada modelo para os dados de teste.

```
fitted %>%
  group_by(modelo) %>%
  metrics(truth = observado, estimate = .pred)
```

modelo	.metric	.estimator	.estimate
lm	rmse	standard	92.1504246
random forest	rmse	standard	174.5574308
lm	rsq	standard	0.9639332
random forest	rsq	standard	0.8644793
lm	mae	standard	77.5163740
random forest	mae	standard	129.7557634

Note que o modelo linear se saiu melhor do que a floresta aleatória nesse caso.

Podemos melhorar o desempenho da floresta aleatória?

Ajuste de hiperparâmetros com `tune`

- O objetivo desse pacote é facilitar o ajuste de hiperparâmetros com a utilização dos pacotes do tidymodels.
- Depende principalmente dos pacotes `recipes`, `parsnip` e `dials`.
- Mais detalhes em https://tidymodels.github.io/tune/.

Este pacote facilita muito o processo de ajuste de hiperparâmetros!

Ajuste de hiperparâmetros com `tune`

A função `tune()` define quais serão os hiperparâmetros que deverão ser selecionados.

```
rf2 <- rand_forest(mtry = tune(), trees = tune(),
                   min_n = tune()) %>%
  set_engine("ranger") %>%

  set_mode("regression")
rf2
```

```
## Random Forest Model Specification (regression)
##
## Main Arguments:
##   mtry = tune()
##   trees = tune()
##   min_n = tune()
##
## Computational engine: ranger
```

Ajuste de hiperparâmetros com `tune`

Para selecionar os valores dos hiperparâmetros, vamos utilizar validação cruzada em 10 lotes.

```
cv_split <- vfold_cv(treinamento, v = 10)
```

Em seguida, definimos um grid para avaliar diferentes valores desses hiperparâmetros.

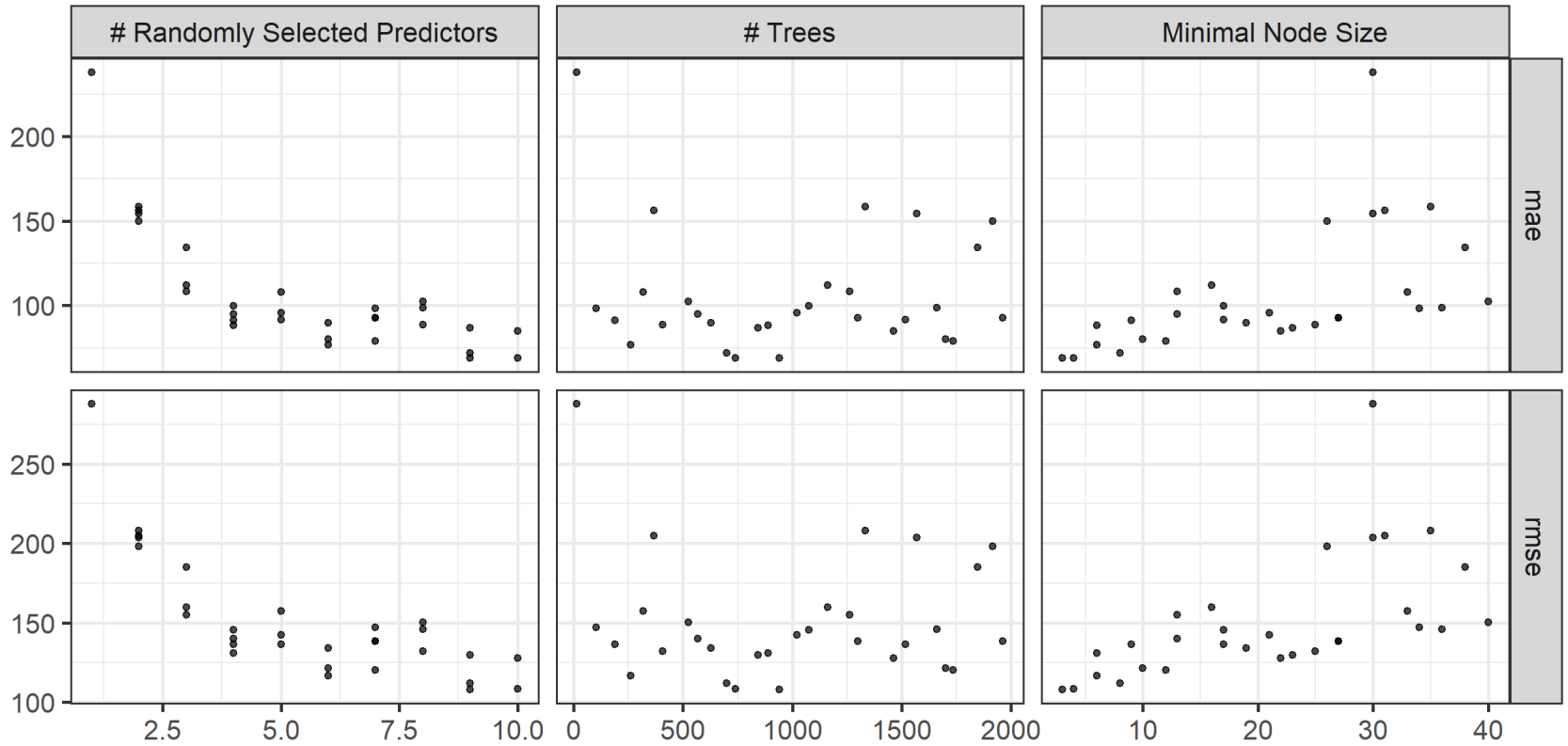
```
doParallel::registerDoParallel()
rf_grid <- tune_grid(rf2,
                     receita,
                     resamples = cv_split,
                     grid = 30,
                     metrics = metric_set(rmse, mae))
rf_grid
```

```
## # Tuning results
## # 10-fold cross-validation
## # A tibble: 10 × 4
##    splits           id     .metrics          .notes
##    <list>           <chr>  <list>            <list>
##  1 <split [288/32]> Fold01 <tibble [60 × 7]> <tibble [0 × 3]>
##  2 <split [288/32]> Fold02 <tibble [60 × 7]> <tibble [0 × 3]>
##  3 <split [288/32]> Fold03 <tibble [60 × 7]> <tibble [0 × 3]>
##  4 <split [288/32]> Fold04 <tibble [60 × 7]> <tibble [0 × 3]>
##  5 <split [288/32]> Fold05 <tibble [60 × 7]> <tibble [0 × 3]>
##  6 <split [288/32]> Fold06 <tibble [60 × 7]> <tibble [0 × 3]>
##  7 <split [288/32]> Fold07 <tibble [60 × 7]> <tibble [0 × 3]>
```

Ajuste de hiperparâmetros com `tune`

A função `autoplot` apresenta um resumo do desempenho de acordo com os níveis do grid.

```
autoplot(rf_grid)
```



Ajuste de hiperparâmetros com `tune`

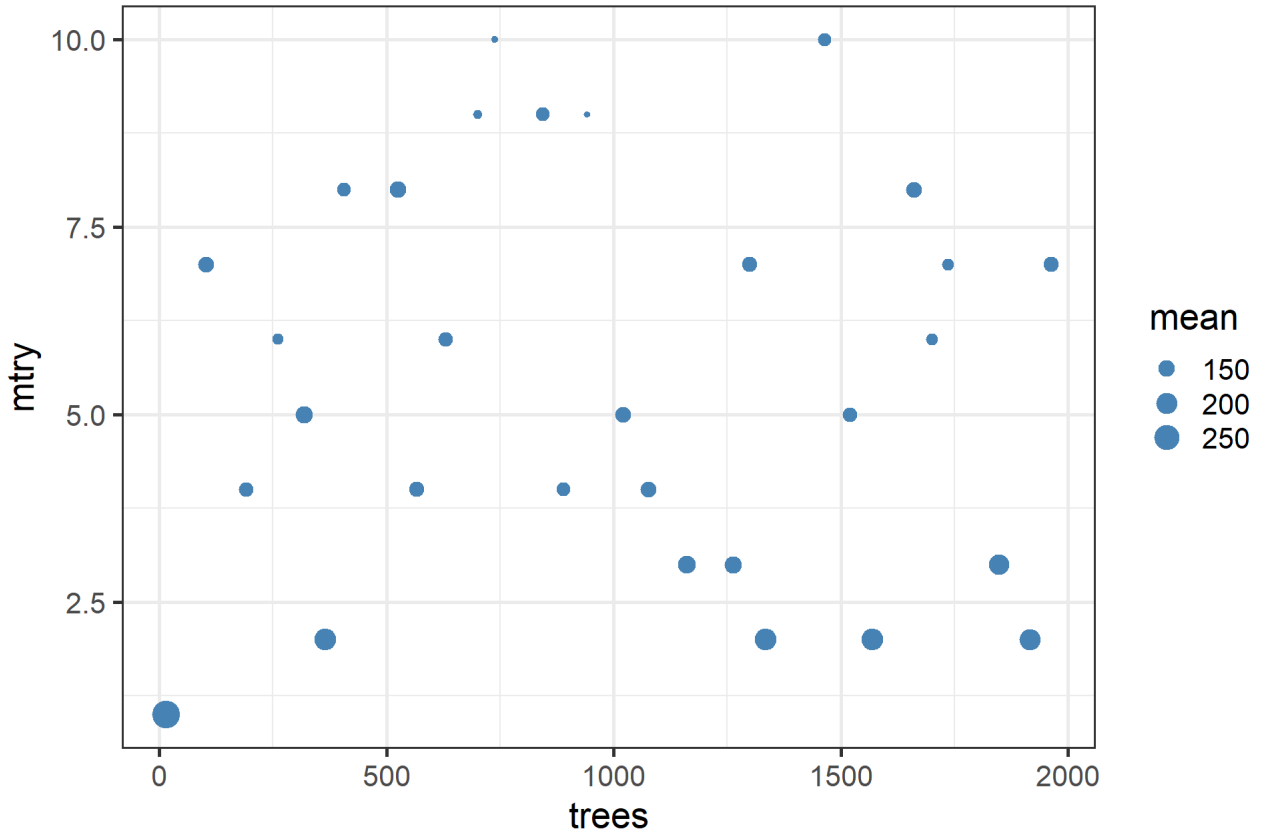
A função `collect_metrics` obtem as métricas de acordo com o grid de hiperparâmetros.

```
rf_grid %>%
  collect_metrics() %>%
  head()
```

mtry	trees	min_n	.metric	.estimator	mean	n	std_err	.config
10	739	4	mae	standard	68.84330	10	4.462490	Preprocessor1_Model01
10	739	4	rmse	standard	108.39240	10	6.590486	Preprocessor1_Model01
8	406	25	mae	standard	88.77308	10	4.691901	Preprocessor1_Model02
8	406	25	rmse	standard	132.23147	10	7.175703	Preprocessor1_Model02
4	890	6	mae	standard	88.31788	10	5.085021	Preprocessor1_Model03
4	890	6	rmse	standard	130.98993	10	8.451637	Preprocessor1_Model03

Ajuste de hiperparâmetros com `tune`

O gráfico abaixo mostra o desempenho do RMSE de acordo com a combinação de valores de `mtrye` `trees`.



Ajuste de hiperparâmetros com `tune`

A função `select_best` seleciona diretamente a melhor combinação de valores para os hiperparâmetros de acordo com uma métrica.

```
rf_grid %>% select_best("rmse")
```

```
## # A tibble: 1 × 4
##    mtry trees min_n .config
##   <int> <int> <int> <chr>
## 1     9   941     3 Preprocessor1_Model05
```

Definido os valores dos hiperparâmetros, podemos finalizar o modelo.

```
best <- rf_grid %>%
         select_best("rmse")

# finaliza modelo
rf_fit2 <- finalize_model(rf2, parameters = best) %>%
            fit(Balance ~ ., tr_proc)
```

Ajuste de hiperparâmetros com `tune`

Finalmente, vamos fazer previsões e adicioná-las ao `tibble` com as informações dos resultados dos outros modelos (lm e floresta aleatória sem ajuste de hiperparâmetros).

```
fitted_rf2 <- rf_fit2 %>%
  predict(new_data = tst_proc) %>%
  mutate(observado = tst_proc$Balance,
         modelo = "random forest - tune")

fitted <- fitted %>%
            bind_rows(fitted_rf2)

tail(fitted)
```

```
## # A tibble: 6 × 3
##     .pred observado modelo
##     <dbl>     <int> <chr>
## 1 334.          371 random forest - tune
## 2 980.         1129 random forest - tune
## 3 857.          806 random forest - tune
## 4   0             0 random forest - tune
## 5 468.          480 random forest - tune
## 6   0.445         0 random forest - tune
```

Comparação entre os modelos ajustados

```
fitted %>%
  group_by(modelo) %>%
  metrics(truth = observado, estimate = .pred)
```

modelo	.metric	.estimator	.estimate
lm	rmse	standard	92.1504246
random forest	rmse	standard	174.5574308
random forest - tune	rmse	standard	130.4013129
lm	rsq	standard	0.9639332
random forest	rsq	standard	0.8644793
random forest - tune	rsq	standard	0.9221225
lm	mae	standard	77.5163740
random forest	mae	standard	129.7557634
random forest - tune	mae	standard	91.7009188

Ajuste do boosting

Vamos adicionar o modelo boosting na comparação. Iniciamos definindo o modelo, os hiperparâmetros, os 10 lotes para a validação cruzada, em seguida ajustamos e selecionamos o melhor modelo.

```
boost <- boost_tree(trees = tune(), min_n = tune(),
                    tree_depth = tune()) %>%
  set_engine("xgboost") %>%
  set_mode("regression")

cv_split <- vfold_cv(treinamento, v = 10)

doParallel::registerDoParallel()

boost_grid <- tune_grid(boost,
                        receita,
                        resamples = cv_split,
                        grid = 30,
                        metrics = metric_set(rmse, mae))

best <- boost_grid %>%
  select_best("rmse")
```

Ajuste do boosting

Finaliza do modelo e adiciona as previsões para o conjunto de teste ao tibble `fitted`.

```
# finaliza modelo
boost_fit <- finalize_model(boost, parameters = best) %>%
  fit(Balance ~ ., tr_proc)

fitted_bst <- boost_fit %>%
  predict(new_data = tst_proc) %>%
  mutate(observado = tst_proc$Balance,
         modelo = "boosting - tune")

fitted <- bind_rows(fitted, fitted_bst)
```

Comparação entre os modelos ajustados

```
fitted %>%
  group_by(modelo) %>%
  metrics(truth = observado, estimate = .pred)
```

modelo	.metric	.estimator	.estimate
boosting - tune	rmse	standard	108.8449788
lm	rmse	standard	92.1504246
random forest	rmse	standard	174.5574308
random forest - tune	rmse	standard	130.4013129
boosting - tune	rsq	standard	0.9439267
lm	rsq	standard	0.9639332
random forest	rsq	standard	0.8644793
random forest - tune	rsq	standard	0.9221225
boosting - tune	mae	standard	70.8996211
lm	mae	standard	77.5163740
random forest	mae	standard	129.7557634
random forest - tune	mae	standard	91.7009188

Resumindo

- Dados em formato tidy seguem um padrão em que cada variável corresponde a uma coluna e cada observação corresponde a uma linha.
- Esse formato permite uma manipulação mais fácil e eficiente dos dados (é compatível com diversos pacotes).
- `tidymodels` fornece um conjunto completo de ferramentas para realizar análise de dados e modelagem preditiva, seguindo os princípios do tidy data.
- `rsample`: Fornece ferramentas para criação e validação de conjuntos de dados de treinamento e teste.
- `recipes`: Fornece ferramentas para pré-processamento de dados, incluindo transformações de variáveis, seleção de recursos e amostragem de dados.
- `parsnip`: Fornece uma interface unificada para criar, avaliar e ajustar modelos.
- `tune`: Fornece ferramentas para ajustar modelos usando validação cruzada e outras técnicas de ajuste de hiperparâmetros.
- `yardstick`: Fornece métricas para avaliar a qualidade de modelos e visualizar os resultados.

Obrigado!

magnotfs@insper.edu.br