Lista 04 - Aprendizado de Máquinas

10 março, 2025

Discussão Preliminar

Antes de iniciar a lista, gostaria de apresentar algumas ferramentas da biblioteca `tidymodels` que serão muito úteis para:

1. pré-processamento dos dados para treinamento e testes - através de receitas;
2. a escolha de "hiper-parâmetros" que são usados nos algoritmos por um processo de validação cruzada.

1. Preparando Receitas para seus Dados

Primeiro, usamos "receitas" para preparamos os dados (do pacote `recipes`). Nestas receitas, dizemos ao `R` como o conjunto que vamos usar deve ser processado antes do treinamento e do teste.

No exemplo abaixo, o conjunto de dados é dividido em treinamento e teste para ilustrar este processo:

```
library(tidymodels)
df <- as_tibble(mtcars)

init.split <- initial_split(df, prop = 0.8)
train <- training(init.split)
test  <- testing(init.split)
```

Vamos criar agora nossa receita de processamento dos dados. O banco de dados `df` é passado aqui somente como um *template* para informar a função `recipe` que tipo de colunas temos nos nossos dados.

```
receita <- recipe( mpg ~ ., data = df ) %>%
  step_center(all_predictors()) %>%  # Centra os dados pela média
  step_scale(all_predictors())       # Escalona os preditores com o desvio padrão
```

Note que criamos uma receita onde iremos centrar os dados pela média (com a função `step_center`) e iremos dividir pelo desvio padrão (com o passo `step_scale`). A função `all_predictors` seleciona todos os preditores que serão usados no nosso modelo (nesse caso todas colunas exceto `mpg`). As funções `step_` são passos para processar nossos dados. Existem várias opções de passos para sua receita de preparo dos dados, veja o link abaixo para maiores detalhes:

https://recipes.tidymodels.org/reference/index.html

Ao declarar a receita como acima, ainda não calculamos nada nem fizemos alterações nos dados. Para fazer o cálculo da receita, precisamos chamar a função `prep` e, para aplicarmos a receita no conjunto de treinamento, precisamos chamar a função `juice`. Vamos preparar a receita usando um **conjunto de treinamento**. Ao preparar a receita, calculamos médias e desvios padrões do conjunto de treinamento e não queremos expor nossos dados de testes nessa etapa, para que nossos modelos não tenham nenhum acesso à eles.

Abaixo chamamos a função de `prep` para preparar a receita usando os dados do conjunto de treinamento e depois usamos `bake` para gerar os novos dados de treinamento corretamente preparados:

```
receita_prep <- prep(receita, training = train)
                                  # ^ Prepara a receita sobre os
                                  # dados de treinamento
train_prep   <- juice(receita_prep) # < Altera o conjunto de dados de treinamento
```

```
                                    #  de acordo com nossa receita de preparação
                                    #  dos dados. Você também pode usar
                                    # bake(receita_prep,new_data = train)
```

Note que `train_prep` foi escalonada para ter preditores com média 0 e desvio padrão 1 em relação às colunas:

```
map_dbl(train_prep,mean)
```

```
##           cyl          disp            hp          drat            wt
## -1.842797e-16 -6.495238e-17 -4.828820e-17 -4.407499e-16  1.249001e-16
##          qsec            vs            am          gear          carb
## -1.157126e-16  1.774622e-17  1.774622e-17 -1.776292e-16  3.939557e-17
##           mpg
##  1.965600e+01
```

```
map_dbl(train_prep,sd)
```

```
##     cyl    disp      hp    drat      wt    qsec      vs      am
## 1.00000 1.00000 1.00000 1.00000 1.00000 1.00000 1.00000 1.00000
##    gear    carb     mpg
## 1.00000 1.00000 6.21799
```

A função `map_dbl(a,f)` aplica a função `f` nos elementos de `a`. Como `a` é um *dataframe*, `a` é interpretada como uma lista cujos elementos são as colunas da tabela `a`. O resultado é concatenando em um vetor de reais (*doubles*, por isso o pós-fixo `_dbl`).

Para rodar a receita preparada sobre os dados de teste, usamos a função `bake`, que altera a base de treinamento com os passos da receita. Isso é ilustrado no código abaixo:

```
test_prep <- bake(receita_prep,new_data = test) # < Altera o conjunto de testes
                                                # de acordo com nossa receita de
                                                # preparação dos dados.
```

O banco de testes é centralizada **com a média e desvio padrão do conjunto de treinamento**, como pode ser visto com os comando abaixo (o resultado foi omitido para reduzir espaço - execute-os em sua máquina):

```
map_dbl(test_prep,mean)
map_dbl(test_prep,sd)
```

2. Escolhendo hiper-parâmetros

Para escolher hiper-parâmetros por um processo de validação cruzada, precisamos aplicar o seguinte algoritmo:

a. Para cada hiper-parâmetro θ de interesse fazer:

   i. Para cada *fold* do nosso conjunto de validação cruzada, fazer:

      1. preparar a receita para o conjunto de treinamento (ou de análise `analysis`) do *fold* atual;
      2. treinar o modelo com os dados preparados pela receita;
      3. aplicar o modelo resultante no conjunto de testes (ou de `assessment`) do *fold* atual, devidamente "*baked*" com nossa receita;
      4. calcular a medida de interesse, como o `rmse`.

   ii. Coletar medidas de interesse, como o `rmse` e tirar a média dos valores dentre os diferentes *folds*.

b. Escolher parâmetro θ com o melhor valor de medida.

Esse processo acima é o mesmo que usamos na Lista02 e Lista03, exceto que agora preparamos os dados antes de treinar e testar com a receita. A biblioteca tidymodels possui uma função que implementa o algoritmo

acima. Para rodar este algoritmo, você precisa passar o modelo com parâmetros marcados para ajuste com a função `tune()`, uma receita - sem preparo - para preprocessar os dados, e uma malha que será usada para buscar os parâmetros, como ilustrado abaixo:

```
## Define um modelo de regressão linear regularizada,
##  queremos encontrar o melhor parâmetro de penalização
##  \lambda (= penalty)
lin.model <- linear_reg( penalty = tune(), # marcamos a penalização para ajuste com tune()
                         mixture = 1) %>% # 0 = Ridge, 1 = Lasso, (0,1) = Elastic-Net
             set_engine("glmnet")

## Malha para buscar o melhor valor para o parâmetro penalty.
##  A função penalty() indica que queremos uma malha para este
##  parâmetro e indica limiares recomendados para o parâmetro.
##  Rode 'penalty()' no console do R para ver o que faz.
lm.grid <- grid_regular(# Define intervalo e escala recomendada para o
                        #  parâmetro penalty. Neste caso, alterei para
                        #  o intervalo de busca para [0.0001,2] (chamei
                        #  a função log10 porque os parâmetros são escalonados
                        #  em uma escala de log na base 10).
                        penalty(range = log10(c(0.0001,2))),
                        levels = 5)# Defini o número de pontos que desejamos gerar
                                   #  para cada parâmetro testado
```

Depois de definir nosso modelo e a malha onde queremos buscar os parâmetros, podemos calcular o erro de validação cruzada usando a função `tune_grid`, como ilustrado abaixo:

```
## Gerando 10-folds para validação cruzada
vfolds <- vfold_cv(df, v = 10)

## Calculando parâmetros
tune.res <- tune_grid(
  lin.model,              # Nosso modelo de aprendizado de máquina
  receita,                # Nossa receita de preparo de dados
  resamples = vfolds,     # O nosso conjunto de validação cruzada
  grid = lm.grid          # Malha de parâmetros para busca
)
```

Por fim, podemos coletar as medidas e fazer um gráfico para escolher o melhor parâmetro. Os resultados abaixo (incluindo gráfico e tabela) foram omitidos - gere-os em sua máquina.

```
tune.res %>%
  collect_metrics() %>%
  filter(.metric == "rmse") %>%
  ggplot(aes(x = penalty, y = mean)) +
  geom_line() +
  geom_point()

#Podemos ver os melhores com o comando:
show_best(tune.res,metric = "rmse")
```

Exercício 01

Vamos usar o banco de dados da FIFA novamente da aula passada. Vamos fazer o mesmo pré-processamento (exceto que usaremos um conjunto menor de preditores para agilizar a análise - você pode usar todos da Lista 03 se desejar):

```
library(tidyverse)
file_url <- "https://drive.google.com/uc?export=download&id=1jiWcGsl_tbqK5F0ryUTq48kcDTKWTTuk"
df <- file_url %>%
    read.csv %>%
    as_tibble %>%
    select(Age, Overall, Potential, Wage, Special,
           Acceleration, Aggression, Agility, Balance, Ball.control,
           Composure, Crossing, Curve, Dribbling, Finishing,Positioning,Vision,Stamina,
           Strength) %>%
    mutate( Wage = as.integer(str_extract(Wage,"[0-9]+")) ) %>%
    mutate_if(is.character,as.integer) %>%
    na.omit()
```

Rode o método Lasso e escolha, por um processo de validação cruzada, o melhor valor para λ, ou seja, a melhor escolha de penalização para a norma ℓ_1 dos coeficientes. Após a escolha do melhor parâmetro, ajuste o modelo sobre todos os dados e veja quais variáveis escolhidas e faça um gráfico do decaimento dos coeficientes a medida que aumentamos a penalização, como ilustrado abaixo:

```
# Com o processo de validação cruzada,
#  suponha que a penalização que faz o rmse
#  ser mínimo (dentre os valores do grid) é
#  quando \lambda = 0.5. Vamos ver os
#  coeficientes \beta_i para o modelo com
#  esse valor de penalização:
library(tidymodels)
lin.fit <- linear_reg( penalty = 0.5, #<--- menor penalização
                       mixture = 1) %>%
             set_engine("glmnet") %>%
             fit( Wage ~ . , data = df )

# Extraindo os coeficientes \beta_j do ajuste
betas <- lin.fit %>%
    pluck("fit") %>% # Extrai o objeto do modelo ajustado
    coef(s = 0.5)    # Calcula os coeficientes para penalização 0.5

# Os valores com um `.` valem exatamente zero.
betas

# Fazendo o gráfico do decaimento
plot( lin.fit %>% pluck("fit"), xvar = "lambda")
```

Exercício 02

Repita o exercício anterior usando o método de regressão Ridge. Note que podemos fazer isso alterando o parâmetro `mixture` para 0 na função `linear_reg`. Compare e discuta os resultados do decaimento dos parâmetros β_j a medida que a penalização aumenta em relação ao resultado do exercício anterior. Compare e justifique o número de valores β_j anulados usando o Lasso e a regressão Ridge.

Exercício 03

Neste exercício, vamos ajustar um modelo de regressão Elastic-Net aos dados `Salaries` da biblioteca `car`. Você pode carregar este banco rodando o seguinte código:

```
library(car)
df <- as_tibble(Salaries)
```

Este banco de dados possui salários de professores em uma faculdade. Note que o banco contém várias variáveis categóricas (ou qualitativas nominais), representadas por fatores. Podemos adicionar variáveis *dummy* para fazer regressão em nossa receita! Com a função `step_dummy`, você pode passar `all_nominal()` para ter todas as colunas correspondentes à variáveis qualitativas nominais.

Para rodar o Elastic-Net, escolha um valor para o parâmetro `mixture` entre $(0, 1)$. Use a função `tune()` para ajustar tanto o parâmetro `penalty` quanto o `mixture`. Use o seguinte grid para a busca de cross-validação:

```
lm.grid <- grid_regular( penalty(),
                         mixture(range = c(0.1,0.9)),
                         levels = 4)
```

Exercício 04

Vamos gerar um banco de dados artificial para esse exercício usando o código abaixo:

```
df <- tibble( x = runif(100,-1,1),
              y = 2*x^3 + x + 10 + rnorm(100,0,0.3))
```

Note que é um polinômio de grau 3. Suponha que não soubéssemos que estes dados foram produzidos por um polinômio de grau 3 e desejamos escolher o melhor grau para a nossa regressão polinomial. Vamos fazer isso variando o grau do polinômio de 1 até 5. Podemos fazer isso, marcando o grau do polinômio na função `set_poly` com a função `tune()`, como ilustrado abaixo:

```
receits <- recipe(y ~ . ,df) %>%
    step_poly(x,degree = tune())
```

Para fazer uma regressão simples, sem penalização, use `set_engine("lm")` na definição do modelo.

Depois de fazer o ajuste, guarde o resultado da função `tune_grid` na variável `tune.res` e faça o gráfico resultante usando o código abaixo:

```
tune.res %>%
  collect_metrics() %>%
  filter(.metric == "rmse") %>%
  ggplot(aes(x = degree, y = mean)) +
  geom_line()
```

Exercício 05 (opcional)

Repita o exercício acima usando uma regressão spline. Faça um ajuste para os graus de liberdade da spline e no final, faça um gráfico do RMSE a medida que os graus de liberdade variam. Use a função `step_ns()` para introduzir as novas colunas relacionadas às splines. O parâmetro `deg_free` pode ser usado para definir o grau de liberdade.

Exercício 06 (opcional)

Fazer exercício 1 do livro ISL na página 297.