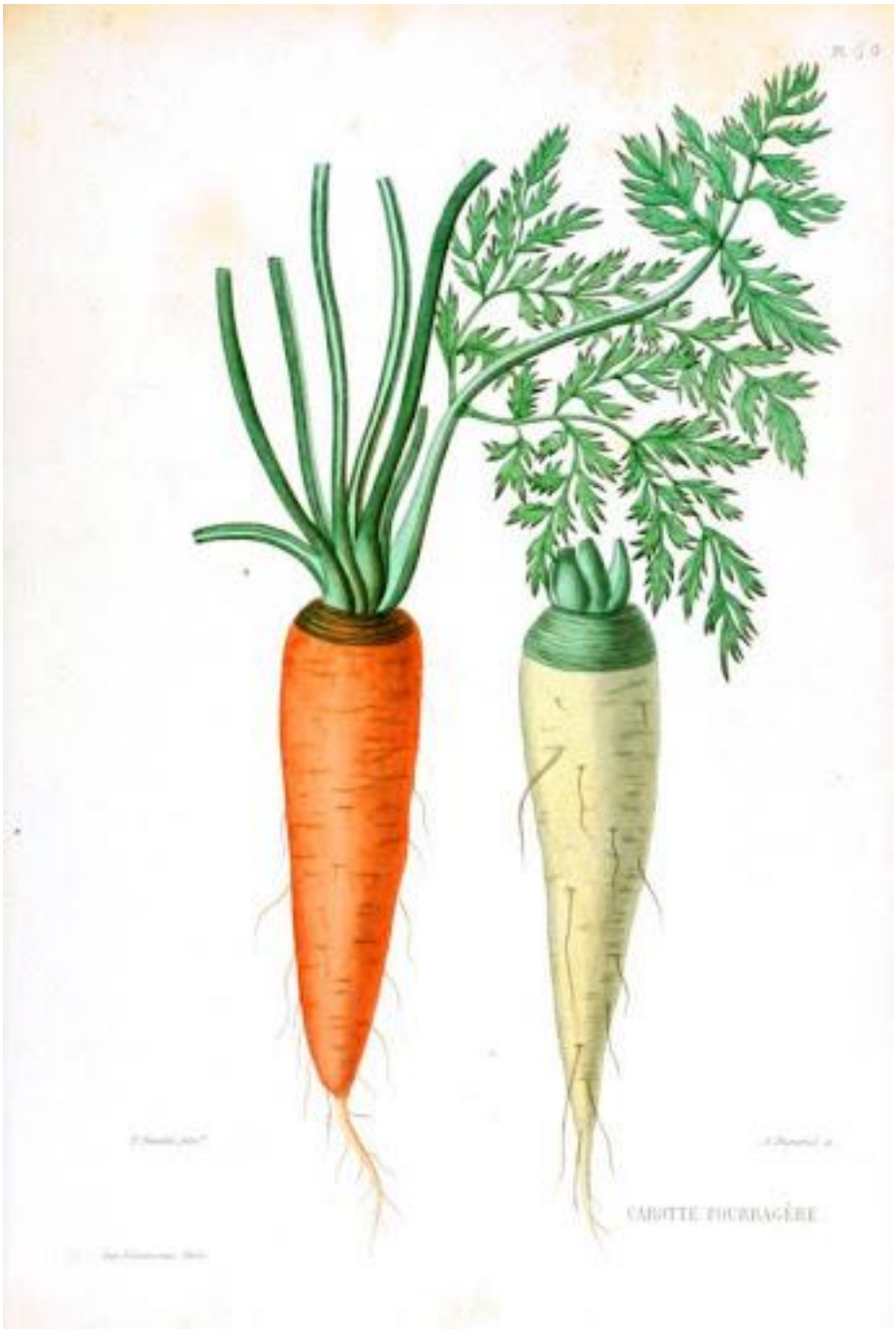
Ajuste de modelos preditivos com o pacote {parsnip}

Edgar Cutar Junior

2019-10-06

Neste segundo post da série sobre o Tidymodels, o conjunto de pacotes desenvolvido por Max Kuhn para ajuste de modelos preditivos para R, iremos falar sobre o {parsnip}. Esse post saiu quase todo da própria página do pacote parsnip assim como de posts do Max Kuhn no RStudio (notadamente este e este). Todas as eventuais piadas ruins são minhas mesmo.

Cenouras e Pastinagas



Parsnip é o nome em inglês da pastinaga, parente esquecida da cenoura. Originalmente parsnip era o pacote que iria substituir o {caret} (a pronúncia de caret é igual a “carrot”, cenoura em inglês), que foi o primeiro pacote integrado para modelagem preditiva para R, desenvolvido no começo dos anos 2000 pelo mesmo Max Kuhn, na época diretor de pesquisa não-clínica na Pfizer.

Com a evolução do pacote, a ideia originaldo Parsnip acabou se desdobrando no conjunto de pacotes que compõem o {tidymodels}, cada um deles especializado em tarefas específicas da modelagem preditiva. Com isso Parsnip ficou com um problema bem específico, a interface.

Uma interface padronizada

Diversas funções e pacotes oferecem diferentes interfaces e parâmetros para objetivos parecidos, e o parsnip padroniza essa interface para o ajuste dos modelos e também para retornar os valores preditos.

O Problema

O problema de inconsistência de interface é facilmente encontrado em diversos modelos do R. Um exemplo simples acontece na regressão logística. Talvez o mais consagrado modo de ajustar uma regressão logística seja através do pacote glm. Esse pacote usa a sintaxe padrão do R (que, na verdade, precede o R).

Para ajustar uma regressão logística no `glm` você:

- Usa o método da fórmula para definir as variáveis preditoras e o que deve ser predito.
- O método da fórmula usa o formato `Y ~ X1 + X2 + X3`
- Para especificar que é um modelo logístico usamos o argumento `family = binomial`

Agora vamos supor que eu queira aplicar alguma regularização nesse modelo. Uma escolha possível seria usando o pacote glmnet. Nesse caso:

- Esse pacote não usa o método da fórmula, deve-se entregar os preditores em uma matriz (o que implica que variáveis "dummy" devem ser pré-computadas)
- O argumento da família muda ligeiramente, e deve ser `family = "binomial"`

O problema seria ainda maior se eu tentasse usar a interface para o TensorFlow no pacote `keras`. O `keras` tem uma abordagem muito interessante para sequeciar modelos de Deep Learning mas a formulação é completamente diferente do tradicional. Ajustar um modelo usando `keras` exigiria estudar e entender uma outra forma de sintaxe de modelos.

O problema se estende a como os diferentes pacotes retornam predições. A maior parte dos pacotes em R usam a função `predict()`. Para retornar um vetor de probabilidades na nossa regressão logística, usaríamos `predict(obj, newdata, type = "response")`. Mas essa convenção varia bastante entre pacotes.

Função	Pacote	Código
glm	stats	predict(obj, type = "response")
lda	MASS	predict(obj)
gbm	gbm	predict(obj, type = "response", n.trees)

mda	mda	predict(obj, type = “posterior”)
rpart	rpart	predict(obj, type = “prob”)
Weka	RWeka	predict(obj, type = “probability”)

Numa instância maior, podemos ter ainda mais problemas: alguns modelos podem criar predições em vários submodelos de uma vez. Quando usamos Boosted Trees ajustadas com *i* árvores, podemos fazer uma predição usando menos de *i* iterações (efetivamente criando um novo modelo de predição), o que pode levar a mais inconsistências.

Esse tipo de problema, quando agregado, podia levar ao questionamento

O R está trabalhando pra mim ou eu estou trabalhando para o R?

A solução

Para demonstrar como o `parsnip` funciona, vamos seguir com o exemplo da regressão logística.

Para isso vamos usar uma base de dados chamada `Smarket`, do pacote ISLR, que tem os retornos diários do índice S&P500 entre 2001 e 2005. Vamos começar dividindo a base em treino e teste. Vamos centralizar e escalar usando uma receita simples do pacote `recipes`(falamos sobre esse pacote no último post)].

```
if (!require("pacman")) install.packages("pacman")

## Loading required package: pacman

pacman::p_load(tidymodels, ISLR)

split <- initial_split(Smarket %>% select(-Year, -Today), props = 9/10)
smarket_train <- training(split)
smarket_test  <- testing(split)

smarket_rec <- recipe(Direction ~ ., data = smarket_train) %>%
  step_center(all_predictors()) %>%
  step_scale(all_predictors()) %>%
  prep(training = smarket_train, retain = TRUE)

train_data <- juice(smarket_rec)
test_data  <- bake(smarket_rec, smarket_test)
```

Para usar o `parsnip`, você começa com a especificação de um modelo. É um objeto simples que define a intenção daquele modelo. Já que vamos seguir com nossa saga de regressão logística, nosso primeiro passo é uma função simples.

```
market_model <- logistic_reg()
market_model

## Logistic Regression Model Specification (classification)
```

Pode parecer estranho porque não entramos com absolutamente nenhum detalhe sobre o que vamos fazer, mas é isso aí mesmo! O `parsnip` oferece uma variedade de formas para ajustar esse modelo. Nós vamos usar o tradicional Método dos Mínimos Quadrados, mas poderia ser com penalização (via lasso, ridge, Bayes etc)... Diferenciamos um caso do outro através da *Engine computacional*, que é uma combinação de tipo de estimação com implementação. Pode ser através de um pacote ou uma plataforma como Spark ou TensorFlow. Para começar simples, vamos usar o glm.

```
glm_market_model <- market_model %>% set_engine("glm")
glm_market_model

## Logistic Regression Model Specification (classification)
##
## Computational engine: glm
```

Não existem mais muitos argumentos por aqui, então vamos pular direto pro ajuste do modelo. Nossas duas escolhas nesse ponto são entre usar `fit()` ou `fit_xy()`. O primeiro usa o método da fórmula, enquanto o segundo usa objetos separados para os preditores e para o resultado.

```
glm_fit <- glm_market_model %>%
  fit(Direction ~ ., data = train_data)

#ou

glm_market_model %>%
  fit_xy(x = select(train_data, -Direction), y = select(train_data,
Direction))

## parsnip model object
##
## Fit time:  0ms
##
## Call:  stats::glm(formula = formula, family = stats::binomial, data =
data)
##
## Coefficients:
## (Intercept)         Lag1         Lag2         Lag3         Lag4
Lag5
##     0.14567     -0.06443     -0.07148      0.02981     -0.01913
0.04871
##      Volume
##     0.02198
##
## Degrees of Freedom: 937 Total (i.e. Null);  931 Residual
## Null Deviance:       1295
## Residual Deviance: 1293  AIC: 1307
```

Claro que não é necessário fazer todos esses passos individualmente, e poderíamos simplesmente condensar tudo em um só objeto.

```
glm_fit <- logistic_reg() %>%
  set_engine("glm") %>%
  fit(Direction ~ ., data = train_data)
```

Mais engines!

O valor do `parsnip` começa quando queremos testar diferentes *engines*. Vamos usar o mesmo modelo e estimar os coeficientes através de estimativa Bayesiana usando `stan`. Para isso só precisamos:

```
stan_model <- logistic_reg() %>%
  set_engine("stan")

stan_model
```

```
## Logistic Regression Model Specification (classification)
##
## Computational engine: stan
```

Para ajustar esse modelo, o `parsnip` chamou a função `stan_glm()` do pacote `rstanarm`. Se você quiser passar argumentos para essa função, simplesmente adicione eles na função `set_engine()`:

```
stan_model <- logistic_reg() %>%
  set_engine("stan", iter = 5000)

stan_model
```

```
## Logistic Regression Model Specification (classification)
##
## Engine-Specific Arguments:
##   iter = 5000
##
## Computational engine: stan
```

O modelo pode ser ajustado da mesma forma. `rstanarm` printa *MUITAS* informações ao ajustar um modelo. Isso pode ser útil para diagnóstico mas vamos excluir usando uma função de controle.

```
ctrl <- fit_control(verbosity = 0)

stan_fit <- stan_model %>%
    fit(Direction ~ ., data = train_data, control = ctrl)

stan_fit
```

```
## parsnip model object
##
```

```
## Fit time:  9.9s
## stan_glm
##  family:       binomial [logit]
##  formula:      Direction ~ .
##  observations: 938
##  predictors:   7
## ------
##             Median MAD_SD
## (Intercept)  0.1    0.1
## Lag1        -0.1    0.1
## Lag2        -0.1    0.1
## Lag3         0.0    0.1
## Lag4         0.0    0.1
## Lag5         0.0    0.1
## Volume       0.0    0.1
##
## ------
## * For help interpreting the printed output see ?print.stanreg
## * For info on the priors used see ?prior_summary.stanreg
```

Predições

Além das óbvias funcionalidades para ajustar diferentes modelos, ainda temos muitas vantagens na hora de prever resultados. Isso resolve uma série de frustrações como ter um arquivo predito que PULA algum resultado quando tem valores faltando, por exemplo, além de padronizar a função.

Para regressões logísticas, por exemplo, o output é sempre um `tibble` com uma coluna `.pred_class` contendo a classe predita.

```
predict(glm_fit, test_data)
```

```
## # A tibble: 312 x 1
##    .pred_class
##    <fct>
##  1 Up
##  2 Up
##  3 Up
##  4 Up
##  5 Up
##  6 Up
##  7 Up
##  8 Up
##  9 Up
## 10 Up
## # ... with 302 more rows
```

```
predict(stan_fit, test_data)
```

```
## # A tibble: 312 x 1
##    .pred_class
```

```
##    <fct>
##  1 Up
##  2 Up
##  3 Up
##  4 Up
##  5 Up
##  6 Up
##  7 Up
##  8 Up
##  9 Up
## 10 Up
## # ... with 302 more rows
```

Isso facilita a união com os valores originais e o `.` no nome é para evitar nomes duplicados.

`parsnip` também trás diferentes tipos de previsão com uma interface padrão. Por exemplo, para estimativa de intervalos.

```
predict(glm_fit, test_data, type = "conf_int")
```

```
## # A tibble: 312 x 4
##    .pred_lower_Down .pred_upper_Down .pred_lower_Up .pred_upper_Up
##               <dbl>            <dbl>          <dbl>          <dbl>
##  1            0.405            0.533          0.467          0.595
##  2            0.403            0.517          0.483          0.597
##  3            0.410            0.526          0.474          0.590
##  4            0.378            0.530          0.470          0.622
##  5            0.395            0.543          0.457          0.605
##  6            0.372            0.514          0.486          0.628
##  7            0.410            0.574          0.426          0.590
##  8            0.387            0.542          0.458          0.613
##  9            0.426            0.547          0.453          0.574
## 10            0.287            0.583          0.417          0.713
## # ... with 302 more rows
```

Ou para as probabilidades de cada previsão:

```
predict(glm_fit, test_data, type = "prob")
```

```
## # A tibble: 312 x 2
##    .pred_Down .pred_Up
##         <dbl>    <dbl>
##  1      0.469    0.531
##  2      0.460    0.540
##  3      0.468    0.532
##  4      0.453    0.547
##  5      0.468    0.532
##  6      0.442    0.558
##  7      0.492    0.508
##  8      0.464    0.536
```

```
##  9      0.486    0.514
## 10      0.429    0.571
## # ... with 302 more rows
```

Próximos passos

Esse artigo apenas toca a superfície das possibilidades de uso do pacote `parsnip`. Nos próximos artigos vamos explorar como fazer uma *grid search* pelos melhores parâmetros, como fazer uma *k-fold Cross Validation* usando esta interface e como avaliar a performance de modelos, tudo isso usando os tidymodels.