Machine Learning with R

tidymodels

June 27th, 2023

So far...

Exploratory data analysis: Wrangling, visualization $\rightarrow$ tidyverse

Unsupervised learning: K-means clustering (stats), Kmeans++ (flexclust), Hierarchical clustering (stats), minimax linkage (protoclust), Gaussian mixture models (mclust)

Linear regression: simple/multiple linear regression (stats), ridge / lasso / elastic net (glmnet)

Within supervised learning we've only really seen two methods: `lm` and `glmnet`, but they've already had differences:

- x and y together? separate?
- cross-validation?

R is free, open-source, and user-contributed

R comes out of the **S** programming language (Bell Labs)... object oriented!

But many (if not most) of the things we do in **R** come from "packages" rather than from "base R", which includes the `base` and `stats` packages

Packages are created by users:

Pro

- You come up with a method, you can code it right up!
- Then I can use your code to fit a model!

Con

- Say what? `G` = `k` = `centers`? nzero?
- Many ways to do one thing may not be ideal

Into the tidy~~verse~~ models

Like the **tidyverse**, **tidymodels** is a suite of packages dedicated to fitting and using models in a *tidy* way, enabling interaction with the other packages we've used so far (`dplyr`, `ggplot`, etc.)

- **parsnip** simplifies model parameters and interfaces
- **recipes** enables feature engineering and preprocessing
- **workflows** allow you to store and execute steps together
- **tune** optimizes hyperparameters

(Also like the tidyverse, tidymodels has fantastic documentation)

Data: NFL teams summary

Created dataset using `nflfastR` summarizing NFL team performances from 1999 to 2021

```
library(tidyverse)
nfl_teams_data <- read_csv("https://shorturl.at/uwAV2")
nfl_model_data <- nfl_teams_data %>%
  mutate(score_diff = points_scored - points_allowed) %>%
  filter(season >= 2006) %>%
  dplyr::select(-wins, -losses, -ties, -points_scored, -points_allowed, -season, -team)
head(nfl_model_data)
```

```
## # A tibble: 6 × 49
##   offense_completion_percentage offense_total_yards_gai…¹ offense_total_yards_…²
##                           <dbl>                     <dbl>                  <dbl>
## 1                         0.561                      3662                   1350
## 2                         0.480                      2371                   2946
## 3                         0.612                      3435                   1667
## 4                         0.564                      2718                   1555
## 5                         0.569                      3264                   1674
## 6                         0.525                      3286                   1940
## # ℹ abbreviated names: ¹offense_total_yards_gained_pass,
## #   ²offense_total_yards_gained_run
## # ℹ 46 more variables: offense_ave_yards_gained_pass <dbl>,
## #   offense_ave_yards_gained_run <dbl>, offense_total_air_yards <dbl>,
```

Using `glmnet`

Divide data into matrix of predictors and vector of response

```
model_x <- model.matrix(score_diff ~ ., nfl_model_data)[, -1]
model_y <- nfl_model_data$score_diff
```

Fit linear regression models:

```
# Base R
init_reg_base <- lm(score_diff ~ .,
                    nfl_model_data)
```

```
library(glmnet)
init_ridge_glmnet <- glmnet(model_x, model_y,
                            alpha = 0)
```

Let's leave all that behind...

```
library(tidymodels)
```

Instead, specify the model with `parsnip`

1. Pick a **model**: in this case, all we need is linear regression, but we could be using e.g. random forests

2. Set the **engine**: which package should we look to for this method?

3. Set the **mode** (if needed): are we performing classification or regression?

This lets us use a similar syntax with many, many different kinds of models by just pointing to their own implementation

According to the tidymodels website, `parsnip` currently supports 119 types of models and engines!

For today...

`linear_reg()` specifies a model that uses linear regression

```
linear_reg(
  mode = "regression",   # this is the "default" mode, but could change
  engine = "lm",         # default is to use base R linear model
  penalty = NULL,        # default is no penalty term... OLS
  mixture = NULL         # default is no mixture of penalties (patience, grasshopper)
)
```

We could also use **pipes** to specify different arguments here

```
linear_reg() %>%
  set_mode(mode = "regression") %>%
  set_engine(engine = "lm")
```

Save this model specification in an object to use later

```
lm_spec <- linear_reg()
```

Train a model based on your specifications: `fit()`

`fit()` returns a parsnip model fit on the data, using your model form

```
simple_parsnip <- lm_spec %>%
  fit(    # formula for the model
    score_diff ~ offense_n_plays_run,
    data = nfl_model_data
  )
simple_parsnip
```

```
## parsnip model object
##
##
## Call:
## stats::lm(formula = score_diff ~ offense_n_plays_run, data = data)
##
## Coefficients:
##         (Intercept)  offense_n_plays_run
##           -323.5461               0.7705
```

Compare with `lm()` output

```
simple_base <- lm(score_diff ~
                    offense_n_plays_run,
   data = nfl_model_data)
simple_base
```

```
##
## Call:
## lm(formula = score_diff ~ offense_n_plays_run, dat
##
## Coefficients:
##         (Intercept)  offense_n_plays_run
##           -323.5461               0.7705
```

Generate predictions

Just like with a linear model object, we can get predictions using `predict()`

```
preds <- predict(simple_parsnip,
        new_data = nfl_model_data)
head(preds)
```

```
## # A tibble: 6 × 1
##   .pred
##   <dbl>
## 1 -6.12
## 2 84.8
## 3 20.1
## 4 -2.26
## 5 -9.20
## 6 48.6
```

But with `parsnip` models, this returns a tibble rather than a vector

Assess model using RMSE

`yardstick` package has its own `rmse()` function!

```
nfl_lm_mod_assess <- simple_parsnip %>%
  predict(new_data = nfl_model_data) %>%
  mutate(obs_score_diff =
           nfl_model_data$score_diff)

rmse(data = nfl_lm_mod_assess,
     truth = obs_score_diff,
     estimate = .pred)
```

```
## # A tibble: 1 × 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 rmse    standard        94.6
```

Which kind of error was that?

Training error! We predicted based on the same data we used to fit the model

But tidymodels makes it really easy to compare train vs. test error as well… Construct a train-test split using `initial_split()`

```
set.seed(1234)
nfl_split <- initial_split(nfl_model_data,
                           prop = 0.75)  # 3/4 split is default but could change
nfl_split
```

```
## <Training/Testing/Total>
## <360/120/480>
```

Recover the training and test sets using `training()` and `testing()`

```
nfl_train <- training(nfl_split)
nfl_test <- testing(nfl_split)
```

Fit and assess based on holdout performance

```
simple_train <- lm_spec %>%
  fit(score_diff ~ offense_n_plays_run, nfl_train)

predict(simple_train, new_data = nfl_test) %>%
  mutate(obs_score_diff = nfl_test$score_diff) %>%
  rmse(truth = obs_score_diff, estimate = .pred)
```

```
## # A tibble: 1 × 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 rmse    standard        95.7
```

Fantastic, we've determined that the test error was greater than the training error.

But this was just for simple linear regression...

Simple $\longrightarrow$ multiple linear regression

All we need to do is specify more variables in our formula

```
two_var_parsnip <- lm_spec %>%
  fit(    # formula for the model
    score_diff ~ offense_n_plays_run +
      offense_n_plays_pass,
    data = nfl_train
  )

predict(two_var_parsnip, new_data = nfl_test) %>%
  mutate(obs_score_diff = nfl_test$score_diff) %>%
  rmse(truth = obs_score_diff, estimate = .pred)
```

```
## # A tibble: 1 × 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 rmse    standard        95.0
```

But this only used one train-test split? What if I want cross-validation?

vfold_cv()

Rather than using a simple train-test split, assign observations to one of ten folds

Then, fit the model using `fit_resamples()`

```
set.seed(52)
folds <- vfold_cv(nfl_model_data, v = 10)
two_var_cv <- lm_spec %>%
  fit_resamples(score_diff ~ offense_n_plays_run +
                  offense_n_plays_pass,
                folds)
```

This gives us a really big model object, with performance metrics on all the folds, which we can view using `collect_metrics()`

```
collect_metrics(two_var_cv)
```

```
## # A tibble: 2 × 6
##   .metric .estimator   mean     n std_err .config
##   <chr>   <chr>       <dbl> <int>   <dbl> <chr>
## 1 rmse    standard   93.4      10  2.32   Preprocessor1_Model1
## 2 rsq     standard    0.168    10  0.0285 Preprocessor1_Model1
```

But why tho???

Didn't we do all this the other day???

What we did the other day:

With `glmnet`, we could fit lasso and ridge regression models with 10-fold cross-validation

`cv.glmnet` would determine the best value for λ, given a value for α, but if we wanted to use ***elastic net*** we had to code the CV ourselves:

```
set.seed(2020)
fold_id <- sample(rep(1:10, length.out = nrow(model_x)))

cv_en_25 <- cv.glmnet(model_x, model_y, foldid = fold_id, alpha = .25)
cv_en_50 <- cv.glmnet(model_x, model_y, foldid = fold_id, alpha = .5)
cv_ridge <- cv.glmnet(model_x, model_y, foldid = fold_id, alpha = 0)
cv_lasso <- cv.glmnet(model_x, model_y, foldid = fold_id, alpha = 1)

which.min(c(min(cv_en_25$cvm), min(cv_en_50$cvm), min(cv_ridge$cvm), min(cv_lasso$cvm)))
```

With `tidymodels`, specifically `tune`, we can do this all in one go!

From multiple linear regression to regularized regression...

Add a penalization term!

Remember our initial model form using `parsnip`? All we need to do is modify the arguments for `engine`, `penalty`, and `mixture`!

```
ridge_example <- linear_reg(
  mode = "regression",
  engine = "glmnet",          # instead of `lm` change to `glmnet`
  penalty = 0.1,              # set lambda here
  mixture = 0                 # "mixture" == "alpha"
)
```

`penalty` = "lambda"

`mixture` = "alpha" from yesterday

But you picked those penalty and mixture values yourself!?

`tune` picks the best for you!

```
elastic_net_spec <- linear_reg(
  mode = "regression",
  engine = "glmnet",
  penalty = tune(),
  mixture = tune()
)
```

`tune()` acts like a placeholder when specifying the hyperparameters for the model

Specify values to try using the `dials` package

I want evenly spaced values for the penalty term λ and the mixture term α, and I want five of each:

```
elnet_grid <- grid_regular(penalty(), mixture(),
                           levels = 5)
```

Model tuning with a grid

First, create the folds as before

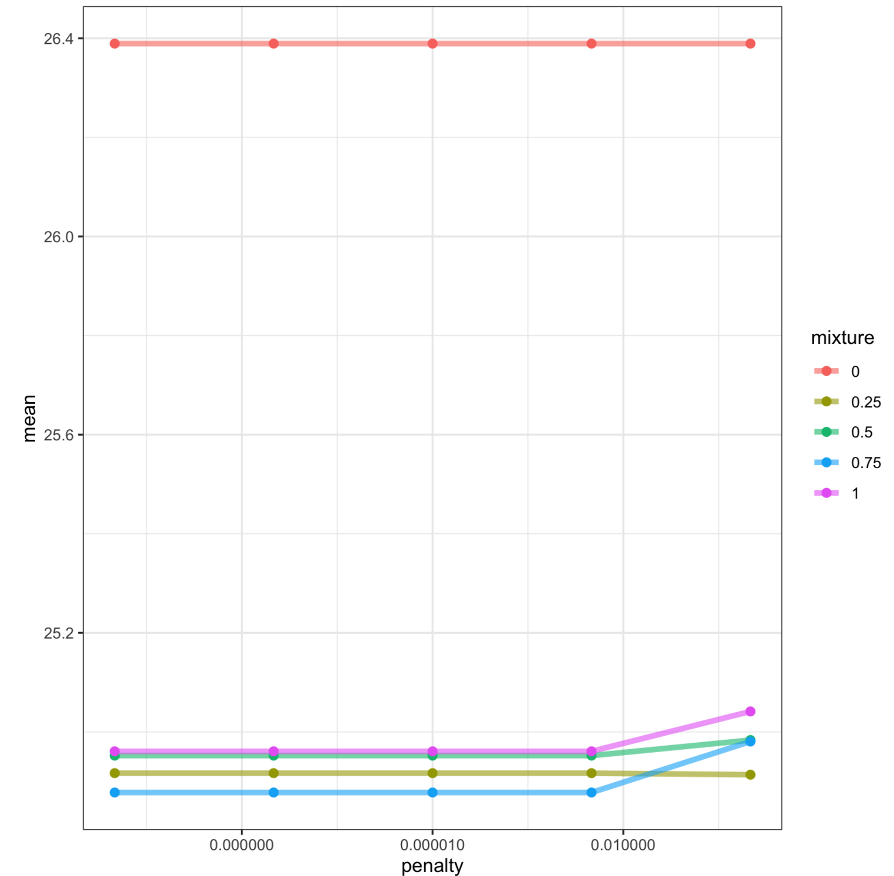
```
set.seed(52)
folds <- vfold_cv(nfl_model_data, v = 10)
```

Then, fit models using `tune_grid()` to use each combination of penalty and mixture values

```
elnet_resample <- tune_grid(
  elastic_net_spec,
  score_diff ~ .,
  resamples = folds,
  grid = elnet_grid
)
```

Which models (which hyperparameter values) performed best?

```
elnet_resample %>%
  collect_metrics() %>%
  filter(.metric == "rmse") %>%
  mutate(mixture = factor(mixture)) %>%
  ggplot(aes(x = penalty,
             y = mean,
             color = mixture)) +
  geom_line(size = 1.5, alpha = 0.6) +
  geom_point(size = 2) +
  scale_x_log10(labels =
                  scales::label_number()) +
  theme_bw()
```



Picking the best one...

```
best_elnet <- elnet_resample %>%
  select_best("rmse")

final_elnet <- linear_reg(
  engine = "glmnet", penalty = best_elnet$penalty, mixture = best_elnet$mixture
) %>%
  fit(score_diff ~ ., data = nfl_train)
```

So then we can use this model to generate predictions, etc.

Recipes and Workflows

Two more packages within tidymodels that allow you to keep track of and recreate your data wrangling and model specification process

Recipe = `formula` + `data`

- Can specify "roles" for different variables, e.g. tell that a particular column acts as an ID
- Turn a factor variable into dummy variables, e.g. what if we wanted a predictor for each team in the NFL?

Workflow = `model` + `recipe`

- Can combine different preprocessing setups (recipes) with different model specs (from parsnip) to compare
- Stores all model results, predictions, etc. in an object where they can be **extracted**

All in all, **tidymodels** allows for great flexibility, while minimizing headaches