Applied Machine Learning - Data Splitting, Models, and Performance

Max Kuhn (RStudio)

Data Usage

Loading

```r
library(tidymodels)
```

```
## Registered S3 method overwritten by 'xts':
##   method     from
##   as.zoo.xts zoo
```

```
## ── Attaching packages ────────────────────────────── tidymodels 0.0.2 ──
```

```
## ✔ broom     0.5.1          ✔ purrr     0.3.3
## ✔ dials     0.0.3.9002     ✔ recipes   0.1.7.9001
## ✔ dplyr     0.8.3          ✔ rsample   0.0.5
## ✔ ggplot2   3.2.1          ✔ tibble    2.1.3
## ✔ infer     0.4.0          ✔ yardstick 0.0.4
## ✔ parsnip   0.0.4
```

```
## ── Conflicts ───────────────────────────────── tidymodels_conflicts() ──
## ✖ purrr::discard()  masks scales::discard()
## ✖ dplyr::filter()   masks stats::filter()
## ✖ dplyr::lag()      masks stats::lag()
## ✖ ggplot2::margin() masks dials::margin()
## ✖ dials::offset()   masks stats::offset()
## ✖ recipes::step()   masks stats::step()
```

```r
library(AmesHousing)
```

Data Splitting and Spending

How do we "spend" the data to find an optimal model?

We *typically* split data into training and test data sets:

- ***Training Set***: these data are used to estimate model parameters and to pick the values of the complexity parameter(s) for the model.

- ***Test Set***: these data can be used to get an independent assessment of model efficacy. They should not be used during model training.

Mechanics of Data Splitting

There are a few different ways to do the split: simple random sampling, *stratified sampling based on the outcome*, by date, or methods that focus on the distribution of the predictors.

For stratification:

- **classification**: this would mean sampling within the classes to preserve the distribution of the outcome in the training and test sets
- **regression**: determine the quartiles of the data set and sample within those artificial groups

Ames Housing Data

Let's load the example data set and split it. We'll put 75% into training and 25% into testing.

```
# rsample loaded with tidyverse or tidymodels package
ames <-
  make_ames() %>%
  # Remove quality-related predictors
  dplyr::select(-matches("Qu"))
nrow(ames)
```

```
## [1] 2930
```

```
# resample functions
# Make sure that you get the same random numbers
set.seed(4595)
data_split <- initial_split(ames, strata = "Sale_Price")

ames_train <- training(data_split)
ames_test  <- testing(data_split)

nrow(ames_train)/nrow(ames)
```

```
## [1] 0.7505119
```

Ames Housing Data

What do these objects look like?

```
# result of initial_split()
# <training / testing / total>
data_split
```

```
## <2199/731/2930>
```

```
training(data_split)
```

```
## # A tibble: 2,199 x 81
##    MS_SubClass MS_Zoning Lot_Frontage Lot_Area Street Alley Lot_Shape Land_Contour Utilities Lot_Config Land_Slope
##    <fct>       <fct>            <dbl>    <int> <fct>  <fct> <fct>     <fct>        <fct>     <fct>      <fct>
##  1 One_Story_… Resident…          141    31770 Pave   No_A… Slightly… Lvl          AllPub    Corner     Gtl
##  2 Two_Story_… Resident…           74    13830 Pave   No_A… Slightly… Lvl          AllPub    Inside     Gtl
##  3 Two_Story_… Resident…           78     9978 Pave   No_A… Slightly… Lvl          AllPub    Inside     Gtl
##  4 One_Story_… Resident…           43     5005 Pave   No_A… Slightly… HLS          AllPub    Inside     Gtl
##  5 One_Story_… Resident…           39     5389 Pave   No_A… Slightly… Lvl          AllPub    Inside     Gtl
## # … and many more rows and columns
## # …
```

Creating Models in R

Specifying Models in R Using Formulas

To fit a model to the housing data, the model terms must be specified. Historically, there are two main interfaces for doing this.

The **formula** interface using R formula rules to specify a *symbolic* representation of the terms:

Variables + interactions

```
model_fn(Sale_Price ~ Neighborhood + Year_Sold + Neighborhood:Year_Sold, data = ames_train)
```

Shorthand for all predictors

```
model_fn(Sale_Price ~ ., data = ames_train)
```

Inline functions / transformations

```
model_fn(log10(Sale_Price) ~ ns(Longitude, df = 3) + ns(Latitude, df = 3), data = ames_train)
```

This is very convenient but it has some disadvantages.

Downsides to Formulas

- You can't nest in-line functions such as `model_fn(y ~ pca(scale(x1), scale(x2), scale(x3)), data = dat)`.
- All the model matrix calculations happen at once and can't be recycled when used in a model function.
- For very *wide* data sets, the formula method can be extremely inefficient.
- There are limited *roles* that variables can take which has led to several re-implementations of formulas.
- Specifying multivariate outcomes is clunky and inelegant.
- Not all modeling functions have a formula method (consistency!).

Specifying Models Without Formulas

Some modeling function have a non-formula (XY) interface. This usually has arguments for the predictors and the outcome(s):

```
# Usually, the variables must all be numeric
pre_vars <- c("Year_Sold", "Longitude", "Latitude")
model_fn(x = ames_train[, pre_vars],
         y = ames_train$Sale_Price)
```

This is inconvenient if you have transformations, factor variables, interactions, or any other operations to apply to the data prior to modeling.

Overall, it is difficult to predict if a package has one or both of these interfaces. For example, `lm` only has formulas.

There is a **third interface**, using *recipes* that will be discussed later that solves some of these issues.

A Linear Regression Model

Let's start by fitting an ordinary linear regression model to the training set. You can choose the model terms for your model, but I will use a very simple model:

```
simple_lm <- lm(log10(Sale_Price) ~ Longitude + Latitude, data = ames_train)
```

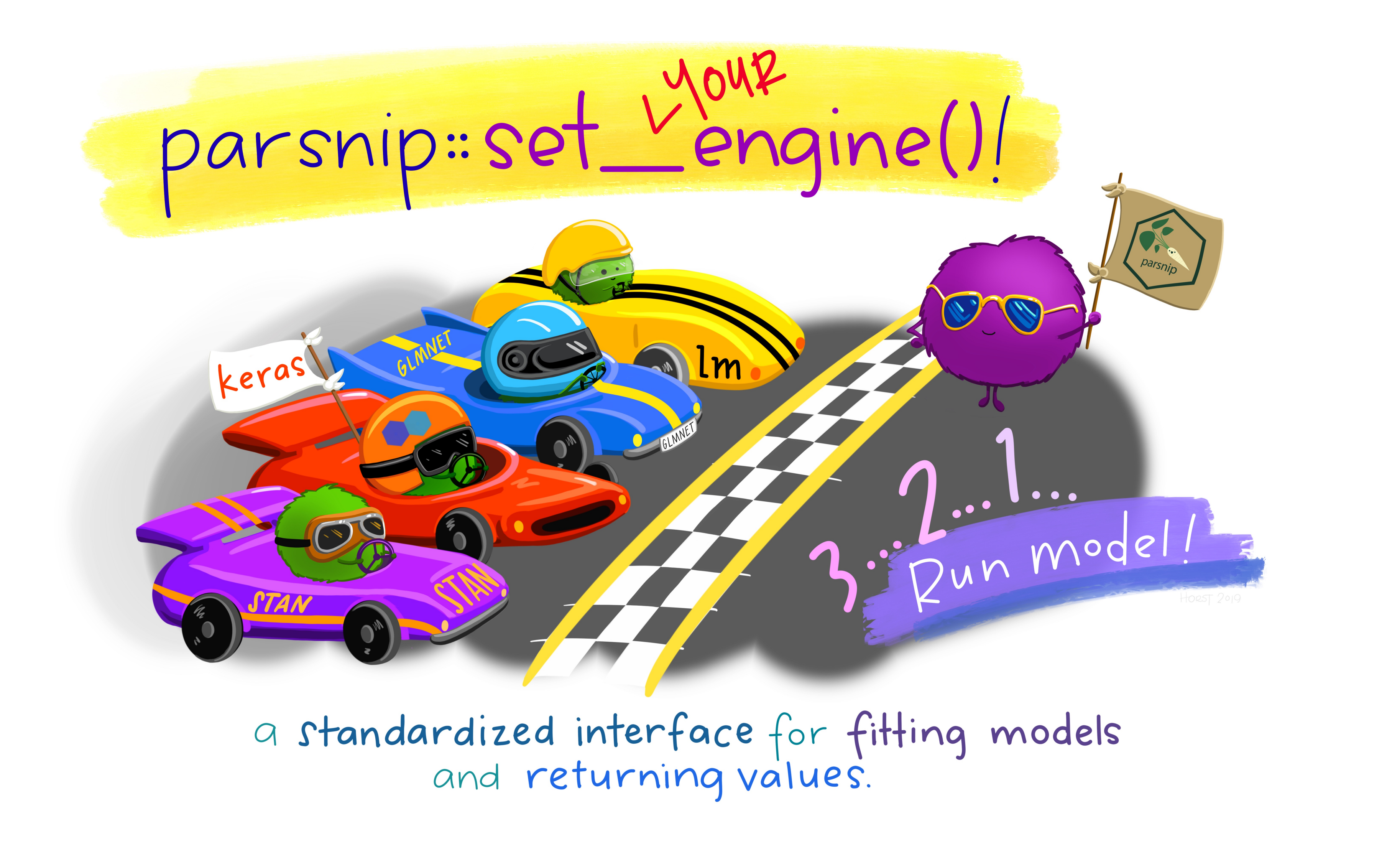
Before looking at coefficients, we should do some model checking to see if there is anything obviously wrong with the model.

To get the statistics on the individual data points, we will use the awesome `broom` package:

```
simple_lm_values <- augment(simple_lm)
names(simple_lm_values)
```

```
##  [1] "log10.Sale_Price." "Longitude"         "Latitude"
##  [4] ".fitted"           ".se.fit"           ".resid"
##  [7] ".hat"              ".sigma"            ".cooksd"
## [10] ".std.resid"
```

parsnip


parsnip::set_your engine()!
keras
GLMNET
lm
GLMNET
STAN
STAN
parsnip
3...2...1...
Run model!
a standardized interface for fitting models
and returning values.

parsnip

- A tidy unified *interface* to models
- `lm()` isn't the only way to perform linear regression
    - `glmnet` for regularized regression
    - `stan` for Bayesian regression
    - `keras` for regression using tensorflow
    - `spark` for large data sets
- But...remember the consistency slide?
    - Each interface has its own minutae to remember
    - `parsnip` standardizes all that!

parsnip in Action

1) Create specification

2) Set the engine

3) Fit the model

```
spec_lin_reg <- linear_reg()
spec_lin_reg
```

```
## Linear Regression Model Specification (regression)
```

```
spec_lm <- set_engine(spec_lin_reg, "lm")
spec_lm
```

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

```
fit_lm <- fit(
  spec_lm,
  log10(Sale_Price) ~ Longitude + Latitude,
  data = ames_train
)

fit_lm
```

```
## parsnip model object
##
## Fit in:  5ms
## Call:
## stats::lm(formula = formula, data = data)
##
## Coefficients:
## (Intercept)    Longitude     Latitude
##    -306.688       -2.032        2.893
```

Different interfaces

`parsnip` is not picky about the interface used to specify terms. Remember, `lm()` only allowed the formula interface!

```
ames_train_log <- ames_train %>%
  mutate(Sale_Price_Log = log10(Sale_Price))

fit_xy(
  spec_lm,
  y = ames_train_log$Sale_Price_Log,
  x = ames_train_log %>% dplyr::select(Latitude, Longitude)
)
```

```
## parsnip model object
##
## Fit in:  3ms
## Call:
## stats::lm(formula = formula, data = data)
##
## Coefficients:
## (Intercept)     Latitude    Longitude
##    -306.688        2.893       -2.032
```

Alternative Engines

With `parsnip`, it is easy to switch to a different engine, like Stan, to run the same model with alternative backends.

```
spec_stan <-
  spec_lin_reg %>%
  # Engine specific arguments are passed through here
  set_engine("stan", chains = 4, iter = 1000)

# Otherwise, looks exactly the same!
fit_stan <- fit(
  spec_stan,
  log10(Sale_Price) ~ Longitude + Latitude,
  data = ames_train
)
```

```
coef(fit_stan$fit)
```

```
## (Intercept)   Longitude    Latitude
## -306.335846   -2.030861    2.884487
```

```
coef(fit_lm$fit)
```

```
## (Intercept)   Longitude    Latitude
## -306.688470   -2.032306    2.892838
```

Different models

Switching *between* models is easy since the interfaces are homogenous.

For example, to fit a 5-nearest neighbor model:

```
fit_knn <-
  nearest_neighbor(mode = "regression", neighbors = 5) %>%
  set_engine("kknn") %>%
  fit(log10(Sale_Price) ~ Longitude + Latitude, data = ames_train)
fit_knn
```

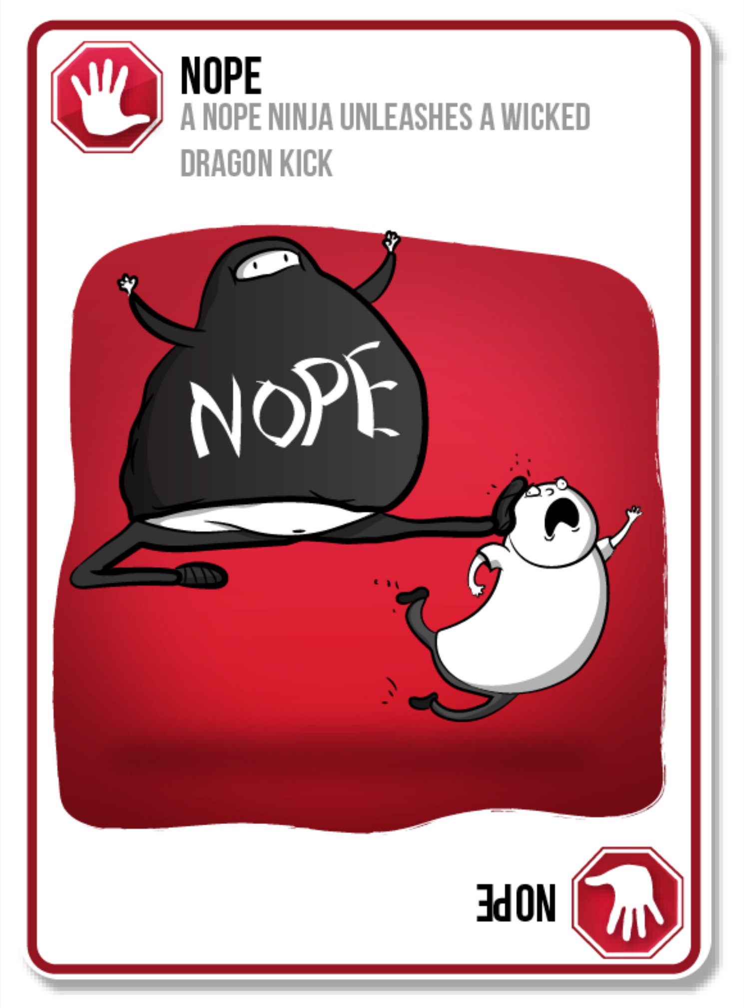
```
## parsnip model object
##
## Fit in:  36ms
## Call:
## kknn::train.kknn(formula = formula, data = data, ks = ~5)
##
## Type of response variable: continuous
## minimal mean absolute error: 0.06753097
## Minimal mean squared error: 0.009633708
## Best kernel: optimal
## Best k: 5
```

DANGER

In a real scenario, we would use *resampling* methods (e.g. cross-validation, bootstrapping, etc) or a validation set to evaluate how well the model is doing.

`tidymodels` has a great infrastructure to do this with the `rsample` package and we will talk about this soon to demonstrate how we should *really* evaluate models.

In general, we would **not** want to predict the test set at this point. I'll do so to illustrate how the code to make predictions works.



Predictions

Now, let's compute predictions and performance measures:

```
# Numeric predictions always in a df
# with column `.pred`
test_pred <-
  fit_lm %>%
  predict(ames_test) %>%
  bind_cols(ames_test) %>%
  mutate(log_price = log10(Sale_Price))

test_pred %>%
  dplyr::select(log_price, .pred) %>%
  slice(1:3)
```

```
## # A tibble: 3 x 2
##   log_price .pred
##       <dbl> <dbl>
## 1      5.33  5.23
## 2      5.02  5.23
## 3      5.27  5.27
```

`parsnip` tools are very standardized.

- `predict` always produces the same data structure (a tibble) with a row for each row of `new_data`.
- The column names are also predictable. For (univariate) regression predictions, the prediction column is always `.pred`.

So, for the KNN model, just change the argument to `fit_knn` and everything works.

Estimating Performance

The `yardstick` package is a tidy interface for computing measures of performance.

There are individual functions for specific metrics (e.g. `accuracy()`, `rmse()`, etc.).

When more than one metric is desired, `metric_set()` can create a new function that wraps them.

Note that these metric functions work with `group_by()`.

```
# yardstick loaded by tidymodels

perf_metrics <- metric_set(rmse, rsq, ccc)

# A tidy result back:
test_pred  %>%
  perf_metrics(truth = log_price, estimate = .pred)
```

```
## # A tibble: 3 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 rmse    standard       0.155
## 2 rsq     standard       0.181
## 3 ccc     standard       0.306
```

There are sometimes different ways to estimate these statistics; `.estimator` is not always "standard".