Lab 004

Data Cleaning and Workflow with Tidymodels

Connor Lennon
21 February 2020

Welcome to Parsnip

Welcome to Parsnip

Why bother with Tidymodels?

- As Ed said: **It's the future, Marty**

No but really: it is extremely intuitive, and streamlines a lot of the annoying parts of data science (data cleaning, cross validation, mutliple model testing, etc.)

Welcome to Parsnip

Why bother with Tidymodels?

- Why use this over `caret()`?

Two reasons:

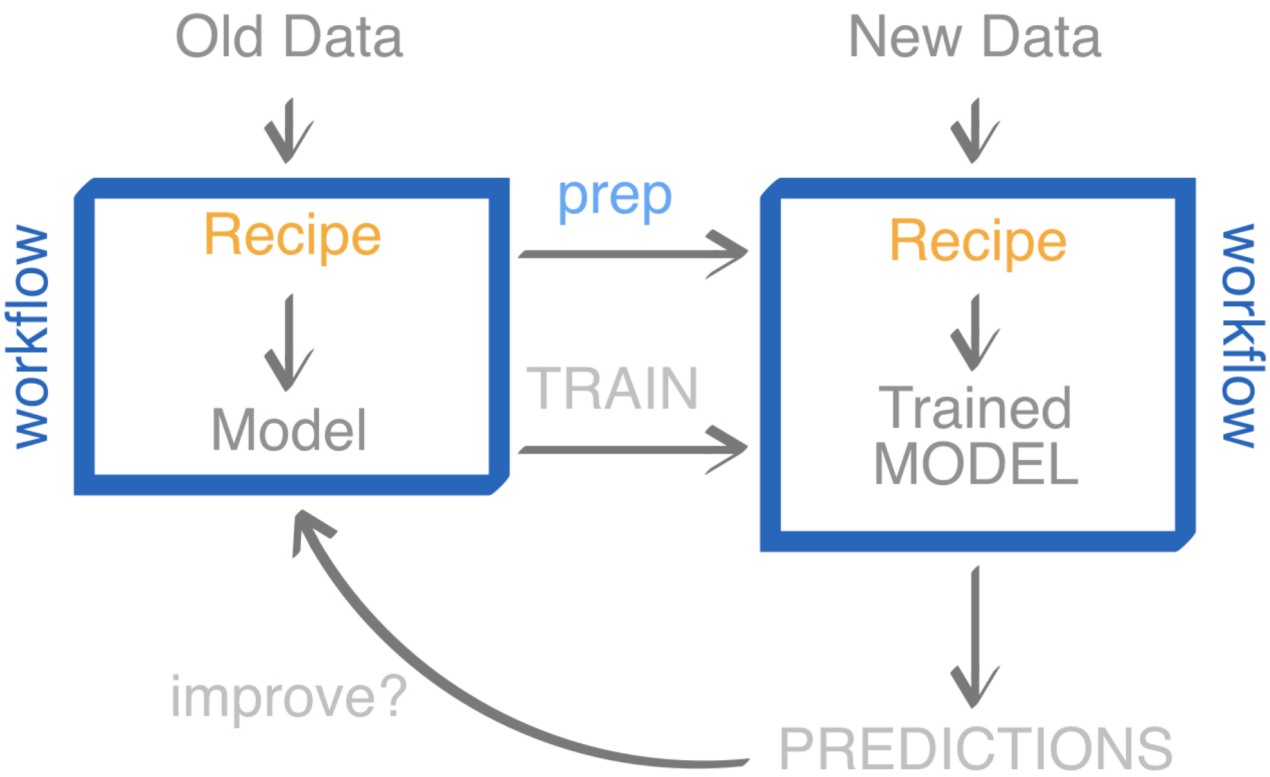
1.) It plays well with the tidyverse, and uses the very nice pipes system from dplyr

2.) It lets you quickly substitute in new parts to reuse old code - making your life a LOT simpler

But, to use `Parsnip` or `tidymodels`, you first have to have a good sense of the 'data science workflow' according to the makers

Workflow

Workflows



* Source: https://education.rstudio.com/blog/2020/02/conf20-intro-ml/.

Example with just parsnip

Parsnip lets us build a model much like Caret, but in the **smallest possible** building blocks.

These include, in order:

- A model type. This is a broad category of model, like **linear regression** or **classification tree**.
- An "engine." This is the package that will run the model type you provided above. This could be either `lm` or `glmnet` for linear regression
- A fit object. This fits the data to the model you built above

Let's do a worked example, using your homework data from the last project.

Models with Parsnip

Let's read in the heart data, and then we can use the `parsnip()` package to run a simple regression

```
#Let's build a model in steps
lazzo = linear_reg()
```

Models with Parsnip

Let's read in the heart data, and then we can use the `parsnip()` package to run a simple regression

```
#Let's build a model in steps
lazzo = linear_reg() %>%
  set_engine('lm')
```

Models with Parsnip

Let's read in the heart data, and then we can use the `parsnip()` package to run a simple regression

```
#Let's build a model in steps
lazzo = linear_reg() %>%
  set_engine('lm') %>%
    parsnip::fit(
    heart_disease ~ sex + chest_pain + resting_bp,
    data = heart_data_tr
    )

rez = predict(lazzo, new_data = heart_data_tr) %>% rename(lm = .pred)
mean((rez[[1]] %>% round() - heart_data_tr$heart_disease)^2, na.rm = T)
```

```
#> [1] 0.2587719
```

Not great. Maybe a lasso would help us? All that work again? **No!**

Models with Parsnip

I can just overwrite an engine/model and then rerun all of this with one simple update

```
#build initial model
lazzo_mod = linear_reg() %>%
  set_engine('lm')
```

```
#switch to lasso/ridge/elasticnet. We can set hyperparameters in 'set_args'
results=lazzo_mod %>% set_engine('glmnet') %>%
  set_args(penalty = 0.001, mixture = 0.5) %>%
    parsnip::fit(
    heart_disease ~ sex + chest_pain + resting_bp,
    data = heart_data_tr
    )

rez = predict(results, new_data = heart_data_tr)
mean((rez[[1]] %>% round() - heart_data_tr$heart_disease)^2, na.rm = T)
```

```
#> [1] 0.2587719
```

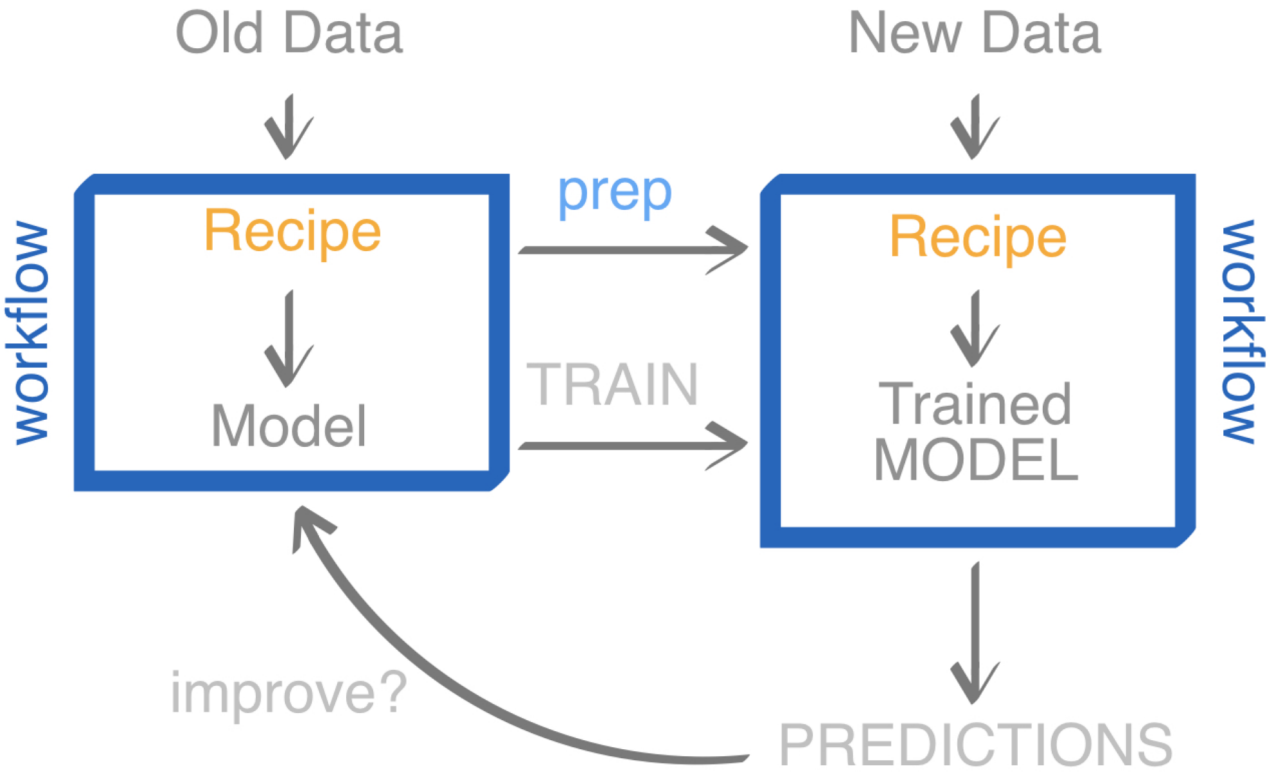
Models with Parsnip

But, from your homework, you know you **really** need to do this using classification methods. And that requires data cleaning, to at the very least get some factors built. **Ugh**

That's where `recipes()` is going to save the day.

But - quickly, we need to revisit our workflow diagram, because we're going to organize everything according to a **workflow** object. Yup, that's a package too

Workflows()



* Source: https://education.rstudio.com/blog/2020/02/conf20-intro-ml/.

Improving your workflow with workflow

Improving your workflow with workflow

Using workflow is quite simple. You just make a `workflow()` object, and tack on steps using a `%>%`.

```
#workflow always starts with workflow
my_big_tuna = workflow()
```

Improving your workflow with workflow

Using workflow is quite simple. You just make a `workflow()` object, and tack on steps using a `%>%`.

```
#let's use a formula to tell our model what to do
my_big_tuna %<>% add_formula(heart_disease ~ sex + chest_pain + resting_bp)
```

Improving your workflow with workflow

Using workflow is quite simple. You just make a `workflow()` object, and tack on steps using a `%>%`.

```
#now we can add a parsnip model from before onto our workflow to predict some stuf
my_big_tuna %<>% add_model(lazzo_mod)
```

Improving your workflow with workflow

Using workflow is quite simple. You just make a `workflow()` object, and tack on steps using a `%>%`.

```
#now we can just pass this thing to fit, and get our fitted object back.
parsnip::fit(my_big_tuna, data = heart_data_tr)
```

```
#> ══ Workflow [trained] ══════════════════════════════════════════════════════
#> Preprocessor: Formula
#> Model: linear_reg()
#>
#> ── Preprocessor ────────────────────────────────────────────────────────────
#> heart_disease ~ sex + chest_pain + resting_bp
#>
#> ── Model ───────────────────────────────────────────────────────────────────
#>
#> Call:
#> stats::lm(formula = formula, data = data)
#>
#> Coefficients:
#> (Intercept)          sex   chest_pain   resting_bp
#>   -1.134294     0.304522     0.219247     0.005328
```

Improving your workflow with workflow

Oh oops. That was an `lm()` model. We might want our `glmnet` back. Ok, no problem

```
#Just create a new model
real_lazzo = lazzo_mod %>%
  set_engine('glmnet') %>%
  set_args(penalty = 0.001, mixture = 0.5)
```

```
#...and update our workflow accordingly
my_big_tuna %<>% update_model(real_lazzo)
parsnip::fit(my_big_tuna, data = heart_data_tr) %>% predict(heart_data_tr) %>% rms
```

.metric	.estimator	.estimate
rmse	standard	0.425

Right... back to `recipes`

Recipes

Recipes

Imagine You are baking muffins...

The idea behind recipes is to treat **data processing** and **model building** as an understandable process (**recipe**) for a program to follow to 'bake' your predictions into the model you want.

- **Step 1:** Separate your ingredients: tell your program what variables are which. This is done using formula and model objects
- **Step 2:** Provide a list of directions to mix your ingredients (change them) in order.
- **Step 3:** Prep your data using your instructions (mixing the recipe)

Recipes essentially act like super sophisticated **formulae** objects, and let you tell your model how it needs to treat any new data it sees.

Recipes

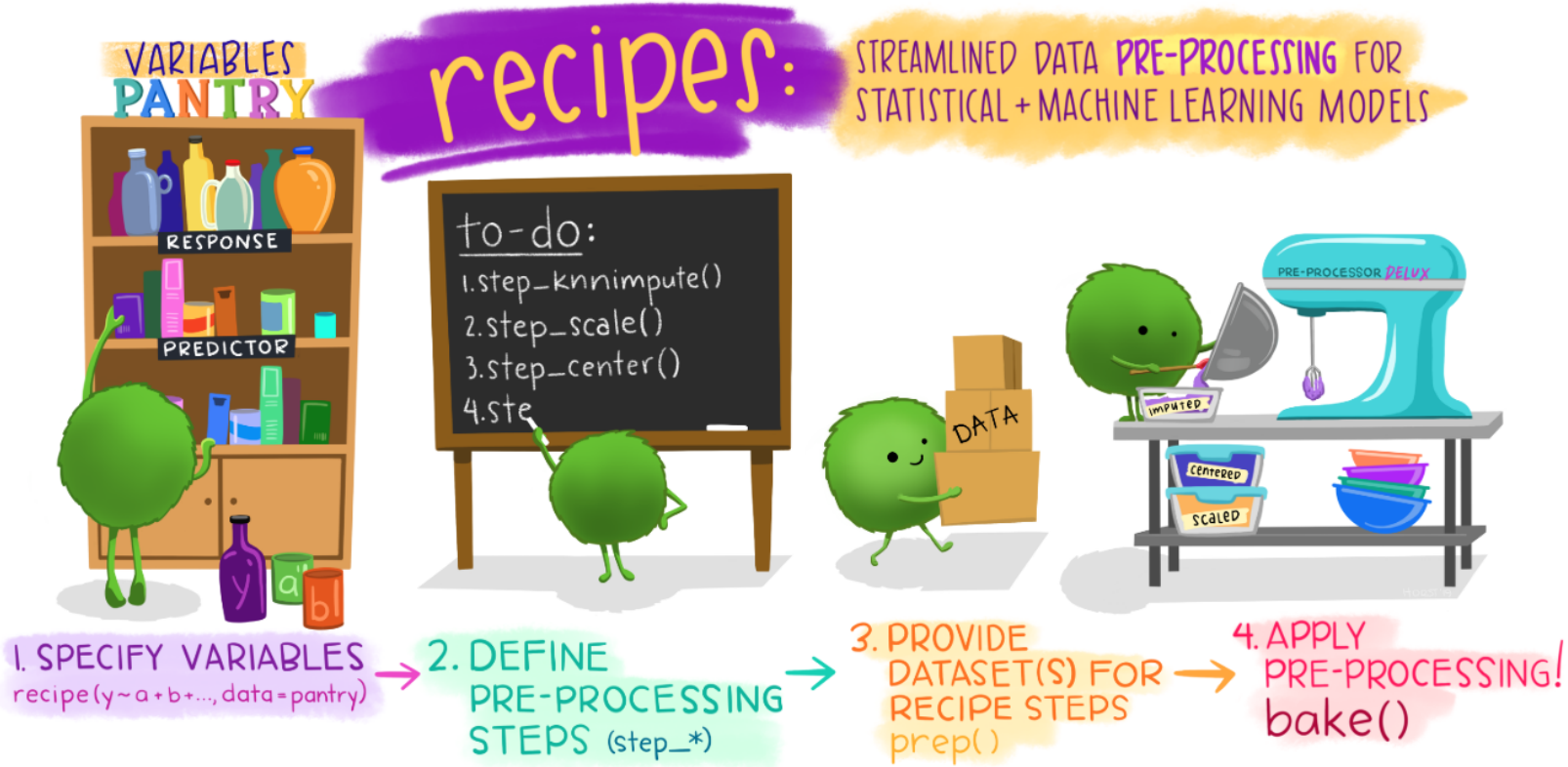
Example

If you recall, we don't want to use ID to do our predicting! Maybe we could keep that variable, and exclude it from all of our data cleaning, but still hold onto it in our data frame... Yup. We can do that.

`Recipes` uses **roles** to separate your **ingredients (variables)** into separate buckets. I'll give in and use the baking analogy: you don't want to sift your eggs and beat your salt.

First though - here's an adorable graphic.

Recipe



* Source: https://education.rstudio.com/blog/2020/02/conf20-intro-ml/.

Recipe

A recipe begins its life, usually, as a formula object that takes some information from our data.

Let's make one and see what it looks like

```
hd_recipe = recipe(heart_disease ~ ., data = heart_data_tr)

hd_recipe %>% summary() %>% head(3)
```

variable	type	role	source
id	numeric	predictor	original
age	numeric	predictor	original
sex	numeric	predictor	original

Recipe

Roles

We can change or add roles (variables can have multiple) by either `update_role` or `add_role`

```
hd_recipe %<>%
  update_role(id, new_role = 'id_variables') %>%
  add_role(sex, exercise_angina, high_sugar, chest_pain, slope, new_role = 'sneaky
```

Recipes

Now what?

Ok, this might not seem useful to you... **but just wait.** We are essentially creating different 'dry ingredients/wet ingredients' shelves in our cabinet so we can tell R to treat them differently in preprocessing (or for another reason.)

- Let's do our preprocessing using this recipe analogy. We can use what are called **selectors** to pick and choose what variables to transform and how. Here's a brief snapshot of some of them that are available

Recipes

- **By name**: `tidyselect::starts_with()`, `tidyselect::ends_with()`, `tidyselect::contains()`, `tidyselect::matches()`, `tidyselect::num_range()`, and `tidyselect::everything()`
- **By role**: `has_role()`, `all_predictors()`, and `all_outcomes()`
- **By type**: `has_type()`, `all_numeric()`, and `all_nominal()`

These then get passed to **'step_*'** methods, of which there are a TON. The cool thing is, is that these roughly follow the naming convention:

step_your_friendly_preProcess_method_from_caret()

Let's see these in action

Recipes

- We want to build our preprocessing as if we were writing a recipe for someone else to replicate, using a series of steps.

```
hd_recipe %<>% step_mutate_at(has_role('sneaky_dummies'), -all_outcomes(), fn = as
```

Recipes

- We want to build our preprocessing as if we were writing a recipe for someone else to replicate, using a series of steps.

```
hd_recipe %<>% step_mutate_at(has_role('sneaky_dummies'), -all_outcomes(), fn = as
  step_center(all_predictors(),
                           -all_nominal())
```

Recipes

- We want to build our preprocessing as if we were writing a recipe for someone else to replicate, using a series of steps.

```
hd_recipe %<>% step_mutate_at(has_role('sneaky_dummies'), -all_outcomes(), fn = as
  step_center(all_predictors(),
                           -all_nominal())  %>%
  step_scale(all_predictors(), -all_nominal()) %>%
  step_nzv(all_predictors()) %>%
  step_mutate(heart_disease = as.factor(heart_disease))
```

Recipes

However, we probably also want to do imputation on our data. Let's do that too.

```
hd_recipe %<>% step_medianimpute(all_numeric()) %>% step_modeimpute(all_nominal())
```

Great, we have done a ton of... **what?**

- We've now given a set of directions to follow when we train a model.
- what model? Any model we want, so long as we are using the data cleaning process we just described.

Training preprocessing?

Before we do anything more, we actually have to set up our ingredients in our shelves. This is what `prep()` does.

```
prepped_recipe = prep(hd_recipe)
```

Why do we need to train a preprocessing step, you might ask?

So we can use the SAME process to transform any dataset we like

- for instance, our testing data.

Recipes

This uses the **bake** functionality of recipes.

By **prepping** the recipe, we now have a set of directions with numeric transformations that can be generalized out of sample.

```
heart_data_ts$heart_disease = -1
bake(prepped_recipe, new_data = heart_data_ts) %>%
  head(2) %>%
  select(age:ecg)
```

age	sex	chest_pain	resting_bp	cholestoral	high_sugar	ecg
-1.26	1	3	-0.111	1.35	0	-1.03
-1.15	1	2	-0.672	0.297	0	-1.03

Getting the data out of the tidymodels sphere

Now, if we're just interested in the training data, we can actually get the transformed data using the `juice` function. I'll show you that in a second.

- Think of `juice()` as **bake light**

Let's bring this all together

Tidymodel process

Just like with our model object, we can pass our recipe object to the workflow directly, in place of the formula we had before.

```
#Classification tree?
ctree_mod = decision_tree() %>% set_engine('rpart') %>%
  set_args(min_n = 8) %>%
  set_mode('classification')

my_big_tuna = workflow() %>%
  add_recipe(hd_recipe) %>% add_model(ctree_mod)
```

Tidymodel process

Now, we can just fit this data here!

```
truevals = (prepped_recipe %>% juice())$heart_disease

hdpred = parsnip::fit(my_big_tuna,
      data = heart_data_tr) %>% predict(new_data = heart_data_tr, type = 'prob') %
  mutate(true_pred = truevals,
         pred = as.numeric(.pred_1))
```

i;gbuj,j