Data Preprocessing using Recipes

Max Kuhn (RStudio) @topepos

R Model Formulas

A simple example of a formula used in a linear model to predict sale prices of houses:

```r
library(AmesHousing)
ames <- make_ames()

levels(ames$Alley)
```

```
## [1] "Gravel"          "No_Alley_Access" "Paved"
```

```r
mod1 <- lm(log(Sale_Price) ~ Alley + Lot_Area, data = ames, subset = Year_Sold > 2000)
```

The purpose of this code chunk:

1. subset some of the data points (`subset`)
2. create a design matrix for 2 predictor variable (but 3 model terms)
3. log transform the outcome variable
4. fit a linear regression model

The first two steps create the *design matrix* (usually represented by X).

Recipes

We can approach the design matrix and preprocessing steps by first specifying a **sequence of steps**.

1. `Sale_Price` is an outcome
2. `Alley` and `Lot_Area` are predictors
3. log transform `Sale_Price`
4. convert `Alley` to dummy variables

A recipe is a specification of *intent*.

One issue with the formula method is that it couples the specification for your predictors along with the implementation.

Recipes, as you'll see, separates the *planning* from the *doing*.

Recipes Workflow

```
   recipe()  -->       prep()     -->  bake() and juice()

{ define } -->  { estimate }  -->       { apply }
```

Recipes

A *recipe* can be trained then applied to any data.

```r
library(recipes)
library(tidyverse)
library(rsample)

set.seed(4595)
data_split <- initial_split(ames, prop = 3/4)

ames_train <- training(data_split) # 75%
ames_test  <- testing(data_split)  # 25%

## Create an initial recipe with the same
## operations as the previous formula
rec <- recipe(Sale_Price ~ Alley + Lot_Area,
              data = ames_train %>% head()) %>%
  step_log(Sale_Price) %>%
  step_dummy(Alley)
```

```r
## `retain = TRUE` keeps the processed training set
## that is created during the estimation phase
rec_trained <-
  prep(rec, training = ames_train, retain = TRUE)


# Get the processed training set:
design_mat <- juice(rec_trained)

## Apply to any other  data set:
rec_test <- bake(rec_trained, newdata = ames_test)
```

Selecting Variables

In the previous slide, we used `dplyr`-like syntax for selecting variables such as `step_dummy(Alley)`.

In some cases, the names of the predictors may not be known at the time when you construct a recipe (or model formula). For example:

- dummy variable columns
- PCA feature extraction when you keep components that capture *X*% of the variability.
- discretized predictors with dynamic bins

`dplyr` selectors can also be used on variables names, such as

```
step_spatialsign(matches("^PC[1-9]"), all_numeric(), -all_outcomes())
```

Variables can be selected by name, role, data type, or any combination of these.

```
# Here too:
design_mat <- juice(rec_trained, all_predictors())
```

Reusing Previous Computations

Need to add more preprocessing or other operations?

```r
standardized <- rec_trained %>%
  step_center(all_numeric()) %>%
  step_scale(all_numeric())

## Only estimate the new parts:
standardized <- prep(standardized, verbose = TRUE)
```

```
## oper 1 step log [pre-trained]
## oper 2 step dummy [pre-trained]
## oper 3 step center [training]
## oper 4 step scale [training]
```

If an initial step is computationally expensive, you don't have to redo those operations to add more.

Available Steps

- **Basic**: logs, roots, polynomials, logits, hyperbolics, ReLu
- **Encodings**: dummy variable, "other" factor level collapsing, discretization, word embeddings[1], likelihood/effects encodings[1]
- **Date Features**: encodings for day/doy/month etc, holiday indicators
- **Filters**: correlation, near-zero variables, linear dependencies
- **Imputation**: bagged trees, nearest neighbor, mean/mode, limit-of-detection imputation, rolling window imputation
- **Normalization/Transformations**: center, scale, range, Box-Cox, Yeo-Johnson
- **Dimension Reduction**: PCA, kernel PCA, PLS, ICA, NNMF[1], Isomap, data depth features, class distances
- **Others**: spline basis functions, interactions, spatial sign
- **Row operations**: class imbalance subsampling, `naomit`, lags

More in process (i.e. autoencoders, more imputation methods, feature hashing, etc.)

One of the package vignettes shows how to write your own step functions.

[1] devel version

Complex Recipe for Ames Data

```
ames_rec <- recipe(Sale_Price ~ Bldg_Type + Neighborhood + Year_Built +
                     Gr_Liv_Area + Full_Bath + Year_Sold + Lot_Area +
                     Central_Air + Longitude + Latitude,
                   data = ames_train) %>%

  step_log(Sale_Price, base = 10) %>%

  # mitigate extreme skewness in some predictors
  step_YeoJohnson(Lot_Area, Gr_Liv_Area) %>%

  # some Neighborhoods are rare, pool them into "other"
  step_other(Neighborhood, threshold = 0.05)  %>%

  step_dummy(all_nominal()) %>%

  step_interact(~ starts_with("Central_Air"):Year_Built) %>%

  # geocode values have highly nonlinear relationships with price
  step_bs(Longitude, Latitude, options = list(df = 5))
```

Using with `caret`

All of these operations should be conducted inside of resampling to get proper error estimates.

The `rsample` package can easily facilitate this and there is a `recipes` interface to `caret::train`.

This defines both the variable specification as well as any preprocessing/filtering/imputation that should be applied.

```
lm_mod <- train(ames_rec, data = ames,
                method = "lm",
                trControl = trainControl(method = "cv"))
```

`caret` does the preprocessing *responsibly* by re-estimating the transformations within the resampling loop.

Similar interfaces for feature selection code are in the development version.

Future Plans

- More steps
- General `dplyr` steps for `filter`, `mutate`, `select`, `rename`, ...
- Integration with other `tidymodels` packages for grid search, model optimization, etc.
- Exportation to TF graph for better deployment