Notes on Ch8: Feature Engineering with Recipes

The Caveman Coder

2025-08-20

Prerequisites

```r
library(tidymodels)
```

```
## -- Attaching packages ------------------------------------- tidymodels 1.3.0 --

## v broom        1.0.8     v recipes      1.3.1
## v dials        1.4.1     v rsample      1.3.1
## v dplyr        1.1.4     v tibble       3.3.0
## v ggplot2      3.5.2     v tidyr        1.3.1
## v infer        1.0.9     v tune         1.3.0
## v modeldata    1.5.0     v workflows    1.2.0
## v parsnip      1.3.2     v workflowsets 1.1.1
## v purrr        1.1.0     v yardstick    1.3.2

## -- Conflicts ---------------------------------------- tidymodels_conflicts() --
## x purrr::discard() masks scales::discard()
## x dplyr::filter()  masks stats::filter()
## x dplyr::lag()     masks stats::lag()
## x recipes::step()  masks stats::step()
```

```r
tidymodels_prefer()

# Data prep from previous chapter
data(ames)

ames <- mutate(ames, Sale_Price = log10(Sale_Price))

# Data split
set.seed(502)
ames_split <- initial_split(ames, prop = 0.80, strata = Sale_Price)
ames_train <- training(ames_split)
ames_test <- testing(ames_split)
```

A simple `recipe()` for the Ames Housing Data

Sample recipe:

```r
simple_ames <-
  recipe(
    Sale_Price ~ Neighborhood + Gr_Liv_Area + Year_Built + Bldg_Type,
    data = ames_train
) |>
  step_log(Gr_Liv_Area, base = 10) |>
```

```r
  step_dummy(all_nominal_predictors())

simple_ames
```

```
##
## -- Recipe ----------------------------------------------------------------------
##
## -- Inputs
## Number of variables by role
## outcome:   1
## predictor: 4
##
## -- Operations
## * Log transformation on: Gr_Liv_Area
## * Dummy variables from: all_nominal_predictors()
```

Advantages of using recipes for modeling:

- computations can be recycled across models since they are not coupled to the modeling function.
- enables a broader set of data processing choices than what formulas can offer.
- the syntax can be very compact (like when using `all_nominal_predictors()`, etc).
- all data preprocessing can be captured in a single R object, instead of being in scripts that are scattered across different files.

Using recipes

Attaching the recipe object to the workflow:

```r
lm_model <- linear_reg() |> set_engine("lm")

lm_wflow <-
  workflow() |>
  add_model(lm_model) |>
  add_recipe(simple_ames)

lm_wflow
```

```
## == Workflow ====================================================================
## Preprocessor: Recipe
## Model: linear_reg()
##
## -- Preprocessor ----------------------------------------------------------------
## 2 Recipe Steps
##
## * step_log()
## * step_dummy()
##
## -- Model -----------------------------------------------------------------------
## Linear Regression Model Specification (regression)
##
## Computational engine: lm
```

Fitting the training data to the model and recipe:

```
lm_fit <- fit(lm_wflow, ames_train)
```

Making a prediction using the model and recipe:

```
predict(lm_fit, ames_test |> slice(1:3))
```

```
## Warning in predict.lm(object = object$fit, newdata = new_data, type =
## "response", : prediction from rank-deficient fit; consider predict(.,
## rankdeficient="NA")

## # A tibble: 3 x 1
##   .pred
##   <dbl>
## 1  5.08
## 2  5.32
## 3  5.28
```

Extracting details from the bare model object or recipe:

```
lm_fit |> extract_recipe(estimated = TRUE)
```

```
##

## -- Recipe ----------------------------------------------------------------------

##

## -- Inputs

## Number of variables by role

## outcome:   1
## predictor: 4

##

## -- Training information

## Training data contained 2342 data points and no incomplete rows.

##

## -- Operations

## * Log transformation on: Gr_Liv_Area | Trained

## * Dummy variables from: Neighborhood Bldg_Type | Trained
```

Extract model details in tidy format:

```
lm_fit |>
  # This returns the parsnip object:
  extract_fit_parsnip() |>
  # Now tidy the linear model object:
  tidy() |>
  slice(1:5)
```

```
## # A tibble: 5 x 5
##   term                         estimate std.error statistic   p.value
##   <chr>                           <dbl>     <dbl>     <dbl>     <dbl>
## 1 (Intercept)                    -0.669    0.231      -2.90 3.80e-  3
## 2 Gr_Liv_Area                     0.620    0.0143     43.2  2.63e-299
```

```
## 3 Year_Built                   0.00200  0.000117     17.1  6.16e- 62
## 4 Neighborhood_College_Creek  0.0178   0.00819       2.17 3.02e-  2
## 5 Neighborhood_Old_Town      -0.0330   0.00838      -3.93 8.66e-  5
```

How data are used by the `recipe()`

1. The `recipe(..., data)` is called, the dataset is used to determine the data types of each column so that selectors like `all_numeric()` or `all_numeric_predictors()` can be used.

2. When `fit(workflow, data)` to prepare the data, the training data are used for all estimation operations.

3. When `predict(workflow, new_data)` is used, no model or preprocessor parameters are re-estimated using the values in `new_data`.

Examples of recipe steps

Encoding qualitative data in a numeric format

- `step_unkown()`: used to change missing values to a dedicated factor level.
- `step_novel()`: used to anticipate a new factor level that may be encountered in future data.
- `step_other()`: used to analyze the frequencies of the factor levels in the training set and convert infrequently occurring variables to a catch-all level or "other". Thresholds can be specified by the user.

Example:

```
simple_ames <-
  recipe(
    Sale_Price ~ Neighborhood + Gr_Liv_Area + Year_Built + Bldg_Type,
    data = ames_train
) |>
  step_log(Gr_Liv_Area, base = 10) |>
  step_other(Neighborhood, threshold = 0.01) |>
  step_dummy(all_nominal_predictors())
```

Interaction terms

Interaction effects occur when one predictor has an effect on the outcome that is contingent on one or more other predictors.

Example: When trying to predict how much traffic there will be during your commute using the specific time of day and the weather, the relationship between the amount of traffic and bad weather can be different depending on the time of day.

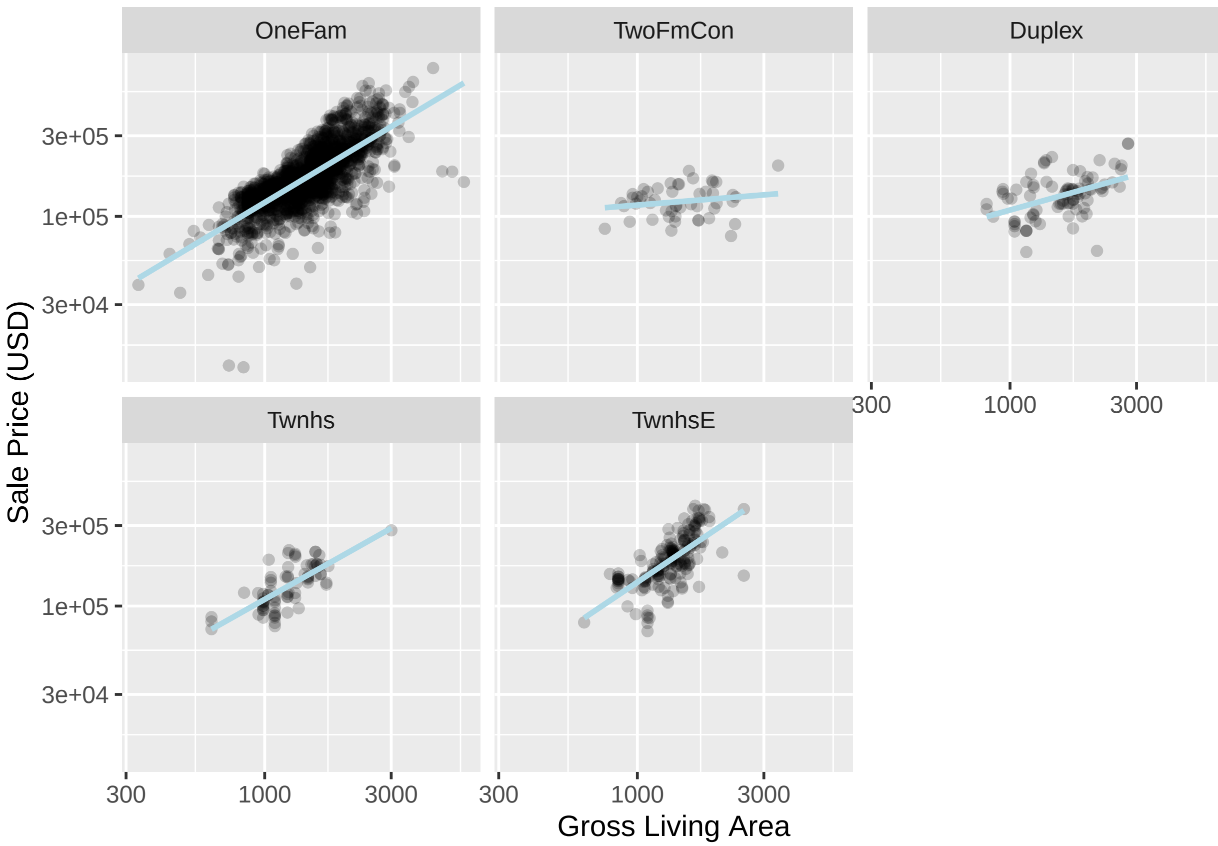
In this case, one could add to the model an interaction term between the two predictors, along with the original two predictors (which are also called "the main effects").

Numerically, the interaction term between predictors is encoded as their product.

Exploring the Ames training set (EDA):

```
ames_train |>
  ggplot(aes(x = Gr_Liv_Area, y = 10^Sale_Price)) +
  geom_point(alpha = 0.2) +
  facet_wrap(~ Bldg_Type) +
  geom_smooth(method = lm, formula = y ~ x, se = FALSE, color = "lightblue") +
  scale_x_log10() +
```

```
  scale_y_log10() +
  labs(x = "Gross Living Area", y = "Sale Price (USD)")
```



In base R formula, the interaction term is specified using a `:`.

```
Sale_Price ~ Neighborhood + log10(Gr_Liv_Area) + Bldg_Type + log10(Gr_Liv_Area):Bldg_Type

# shorhand form of the formula above:
Sale_Price ~ Neighborhood + log10(Gr_Liv_Area) * Bldg_Type
```

In recipes, interaction terms are specified in a more explicit manner (using `step_interact()`:

```
simple_ames <-
  recipe(
    Sale_Price ~ Neighborhood + Gr_Liv_Area + Year_Built + Bldg_Type,
    data = ames_train
) |>
  step_log(Gr_Liv_Area, base = 10) |>
  step_other(Neighborhood, threshold = 0.01) |>
  step_dummy(all_nominal_predictors()) |>
  # Gr_Liv_Area is on the log scale already!
  step_interact(~ Gr_Liv_Area:starts_with("Bldg_Type_"))
```

The function `starts_with()` was used here because `step_dummy()` uses an underscore as a separator between the column name and the level.

Additional interactions can be specified by separating them with `+`.

Any interaction term that interacts with itself (`var_1:var_1`), it will be ignored.

The interaction terms will be named according to the syntax `var_1_x_var_2` in order for the column name to be a valid data frame column name.

Spline functions

These functions are used to represent nonlinear relationship between the predictors and the outcome.

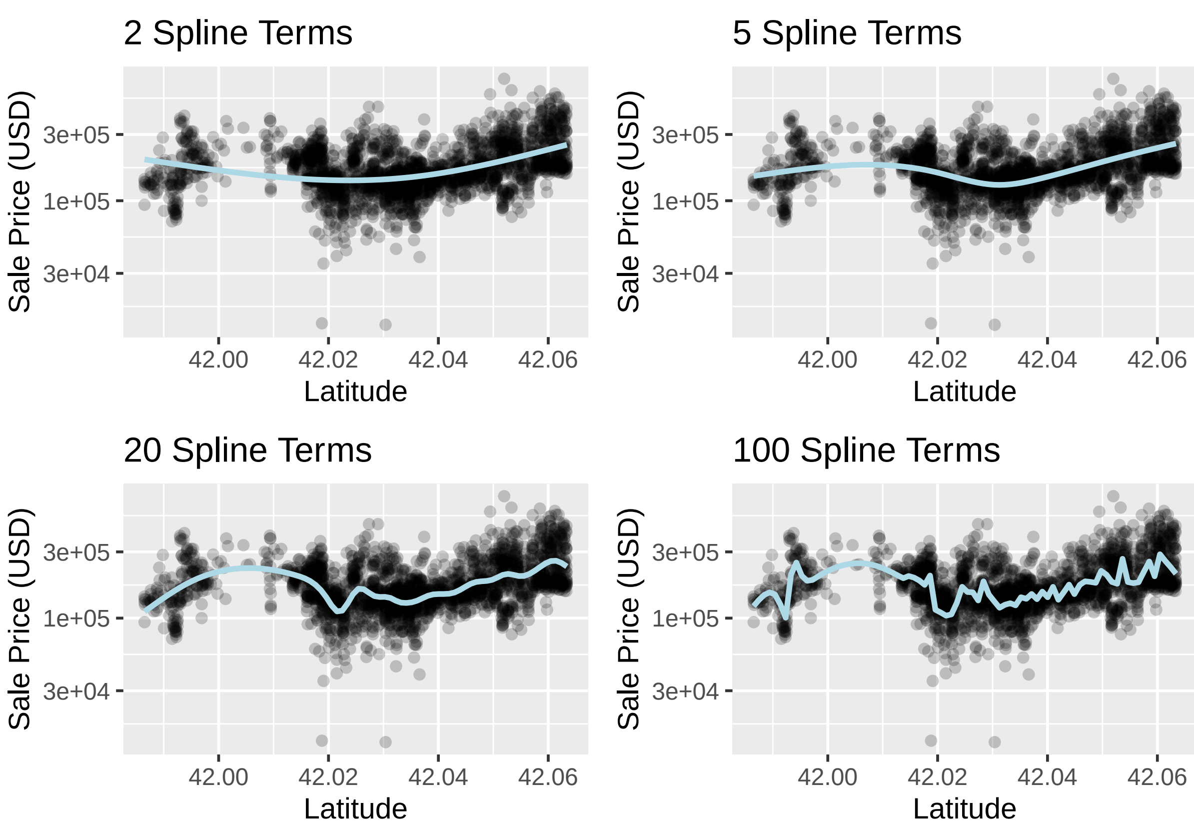
The function `geom_smooth()` in `ggplot`, is a spline representation of the data.

Example:

```
library(patchwork)
library(splines)

plot_smoother <- function(deg_free) {
  ggplot(ames_train, aes(x = Latitude, y = 10^Sale_Price)) +
    geom_point(alpha = 0.2) +
    scale_y_log10() +
    geom_smooth(
      method = lm,
      formula = y ~ ns(x, df = deg_free),
      color = "lightblue",
      se = FALSE
    ) +
    labs(title = paste(deg_free, "Spline Terms"),
         y = "Sale Price (USD)")
}

( plot_smoother(2) + plot_smoother(5) ) / ( plot_smoother(20) + plot_smoother(100) )
```



The plot shows that two spline terms underfit the data, while 100 terms overfit. Having spline terms between 5 and 20 seem reasonable (i.e. the splines are fairly smooth and catch the main patterns of the data).

Adding a natural spline (`ns`) representation in recipes:

```
recipe(
  Sale_Price ~ Neighborhood + Gr_Liv_Area + Year_Built + Bldg_Type + Latitude,
  data = ames_train
) |>
  step_log(Gr_Liv_Area, base = 10) |>
  step_other(Neighborhood, threshold = 0.01) |>
  step_dummy(all_nominal_predictors()) |>
  step_interact( ~ Gr_Liv_Area:starts_with("Bldg_Type_")) |>
  step_ns(Latitude, deg_free = 20)
```

The number of splines to specify could be considered as a *tuning parameter* for the model.

Feature extraction

This is a method for representing multiple features at once. Most methods falling under this category create new features (aka columns) that capture the information in the broader set as a whole. Common example: PCA or Principal component analysis. This method tries to extract as much of the original information in the predictor set as possible using a smaller number of features.

In the `Ames` dataset, several predictors are used to measure the size of the property – the `Total_Bsmt_SF`, `First_Flr_SF`, and `Gr_Liv_Area` (total basement size, size of first floor, and gross living area, respectively). PCA can be used to represent these potentially redundant variables as a smaller feature set.

Possible recipe step for PCA for the use case explained above:

```
step_pca(matches("(SF$)|(Gr_Liv)"))
```

Note: `step_pca()` assumes that all of the predictors are on the same scale. For this reason, this step is often preceded by `step_normalize()`, which centers and scales each column.

Other recipe steps for feature extraction include independent component analysis (ICA), non-negative matrix factorization (NNMF), multidimensional scaling (MDS), uniform manifold approximation and projection (UMAP), etc.

Row sampling steps

Subsampling techniques are popularly used to deal with class imabalance. Common subsampling approaches:

- *Downsampling*: the data keeps the minority class and takes a random sample of the majority class. Aim is to make the class frequencies balance.
- *Upsampling*: replicates samples from the minority class to balance the classes.
- *Hybrid*: combination of both.

Subsampling syntax using the **themis** package:

```
step_downsample(outcome_column_name)
```

General transformations

The `step_mutate()` can be used to conduct a variety of basic operations to the data, similar to the **dplyr** `mutate()`.

Natural language processing

Functions from the **textrecipes** package can apply NLP methods to the data.

Skipping steps for new data

Sometimes, there will be a step in the modeling process that should not be applied to new data. Example: the subsampling step performed to handle class imbalances during training - this step should not be performed when predicting on new data.

Solution: Each function in the `step_*()` family has a `skip` argument that when set to `TRUE`, will be ignored by the `predict()` function.

`tidy()` method for recipes

Sample recipe using the steps discussed above:

```
ames_rec <-
  recipe(
    Sale_Price ~ Neighborhood + Gr_Liv_Area + Year_Built + Bldg_Type + Latitude + Longitude,
    data = ames_train
  ) |>
  step_log(Gr_Liv_Area, base = 10) |>
  step_other(Neighborhood, threshold = 0.01) |>
  step_dummy(all_nominal_predictors()) |>
  step_interact( ~ Gr_Liv_Area:starts_with("Bldg_Type_")) |>
  step_ns(Latitude, Longitude, deg_free = 20)
```

Calling the `tidy()` method on the recipe object above:

```
tidy(ames_rec)
```

```
## # A tibble: 5 x 6
##   number operation type      trained skip  id
##    <int> <chr>     <chr>     <lgl>   <lgl> <chr>
## 1      1 step      log       FALSE   FALSE log_1iDg1
## 2      2 step      other     FALSE   FALSE other_anRnU
## 3      3 step      dummy     FALSE   FALSE dummy_Xe89S
## 4      4 step      interact  FALSE   FALSE interact_yj20u
## 5      5 step      ns        FALSE   FALSE ns_1akCq
```

This result can be very helpful for verifying the individual steps.

Note: the entries under the `id` column contains some random suffix which are not really helpful. Specifying the `id` ahead of time for `step_other()` will improve this situation:

```
ames_rec <-
  recipe(
    Sale_Price ~ Neighborhood + Gr_Liv_Area + Year_Built + Bldg_Type + Latitude + Longitude,
    data = ames_train
  ) |>
  step_log(Gr_Liv_Area, base = 10) |>
  step_other(Neighborhood, threshold = 0.01, id = "my_id") |>
  step_dummy(all_nominal_predictors()) |>
  step_interact( ~ Gr_Liv_Area:starts_with("Bldg_Type_")) |>
  step_ns(Latitude, Longitude, deg_free = 20)

tidy(ames_rec)
```

```
## # A tibble: 5 x 6
##   number operation type     trained skip  id
##    <int> <chr>     <chr>    <lgl>   <lgl> <chr>
## 1      1 step      log      FALSE   FALSE log_0CwGV
## 2      2 step      other    FALSE   FALSE my_id
## 3      3 step      dummy    FALSE   FALSE dummy_NkzTj
## 4      4 step      interact FALSE   FALSE interact_AmBz1
## 5      5 step      ns       FALSE   FALSE ns_8R5Nn
```

Refitting the workflow with the new recipe:

```
lm_wflow <-
  workflow() |>
  add_model(lm_model) |>
  add_recipe(ames_rec)

lm_fit <- fit(lm_wflow, ames_train)
```

Calling the `tidy()` method again, along with the `id` identifier specified for `step_other()`:

```
estimated_recipe <-
  lm_fit |>
  extract_recipe(estimated = TRUE)

tidy(estimated_recipe, id = "my_id")
```

```
## # A tibble: 22 x 3
##    terms        retained           id
##    <chr>        <chr>              <chr>
##  1 Neighborhood North_Ames         my_id
##  2 Neighborhood College_Creek      my_id
##  3 Neighborhood Old_Town           my_id
##  4 Neighborhood Edwards            my_id
##  5 Neighborhood Somerset           my_id
##  6 Neighborhood Northridge_Heights my_id
##  7 Neighborhood Gilbert            my_id
##  8 Neighborhood Sawyer             my_id
##  9 Neighborhood Northwest_Ames     my_id
## 10 Neighborhood Sawyer_West        my_id
## # i 12 more rows
```

If we know the `number` identifier from the result of calling `tidy()` on the recipe (i.e., `tide(ames_rec)`), we can use that instead of `id` when generating the tidied up `estimated_recipe`:

```
tidy(estimated_recipe, number = 2)
```

```
## # A tibble: 22 x 3
##    terms        retained           id
##    <chr>        <chr>              <chr>
##  1 Neighborhood North_Ames         my_id
##  2 Neighborhood College_Creek      my_id
##  3 Neighborhood Old_Town           my_id
##  4 Neighborhood Edwards            my_id
##  5 Neighborhood Somerset           my_id
##  6 Neighborhood Northridge_Heights my_id
##  7 Neighborhood Gilbert            my_id
##  8 Neighborhood Sawyer             my_id
```

```
##   9 Neighborhood Northwest_Ames     my_id
## 10 Neighborhood Sawyer_West        my_id
## # i 12 more rows
```

Each `tidy()` call for a `step_*()` function returns relevant information about that step.

Column roles

The formula used with the initial call to `recipe()` assigns *roles* to each of the columns, depending on which side of the tilde (`~`) they are on. The roles are either `predictor` or `outcome`.

Sometimes, it may be necessary to keep a column even though it is neither a predictor nor an outcome. (Example: the `address` column in the `Ames` dataset).

The function `*_role()` can be used to add, remove, or update roles. Example: updating the role of `address` in the recipe:

```
ames_rec |>
  update_role(address, new_role = "street address")
```

After which the `address` column will cease to be a predictor, but instead will be a "`street address`" in the recipe.

All functions in the `step_*()` family have a `role` field that we can tweak, depending on what is necessary for each step.

Summary

Standard code for this chapter:

```
library(tidymodels)
tidymodels_prefer()

data(ames)
ames <- mutate(ames, Sale_Price = log10(Sale_Price))

set.seed(502)
ames_split <- initial_split(ames, prop = 0.80, strata = Sale_Price)
ames_train <- training(ames_split)
ames_test <- testing(ames_split)

ames_rec <-
  recipe(
    Sale_Price ~ Neighborhood + Gr_Liv_Area + Year_Built + Bldg_Type + Longitude + Latitude,
    data = ames_train
  ) |>
  step_log(Gr_Liv_Area, base = 10) |>
  step_other(Neighborhood, threshold = 0.01) |>
  step_dummy(all_nominal_predictors()) |>
  step_interact( ~ Gr_Liv_Area:starts_with("Bldg_Type_")) |>
  step_ns(Latitude, Longitude, deg_free = 20)

lm_model <- linear_reg() |> set_engine("lm")

lm_wflow <-
  workflow() |>
  add_model(lm_model) |>
```

```r
  add_recipe(ames_rec)

lm_fit <- fit(lm_wflow, ames_train)
```