Feature Engineering

AU STAT-427/627

Emil Hvitfeldt

2021-09-27

What happens to the data between `read_data()` and `fit_model()`?

Prices of 54,000 round-cut diamonds

```r
library(ggplot2)
diamonds
```

```
## # A tibble: 53,940 × 10
##    carat cut       color clarity depth table price     x     y     z
##    <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
##  1  0.23 Ideal     E     SI2      61.5    55   326  3.95  3.98  2.43
##  2  0.21 Premium   E     SI1      59.8    61   326  3.89  3.84  2.31
##  3  0.23 Good      E     VS1      56.9    65   327  4.05  4.07  2.31
##  4  0.29 Premium   I     VS2      62.4    58   334  4.2   4.23  2.63
##  5  0.31 Good      J     SI2      63.3    58   335  4.34  4.35  2.75
##  6  0.24 Very Good J     VVS2     62.8    57   336  3.94  3.96  2.48
##  7  0.24 Very Good I     VVS1     62.3    57   336  3.95  3.98  2.47
##  8  0.26 Very Good H     SI1      61.9    55   337  4.07  4.11  2.53
##  9  0.22 Fair      E     VS2      65.1    61   337  3.87  3.78  2.49
## 10  0.23 Very Good H     VS1      59.4    61   338  4     4.05  2.39
## # … with 53,930 more rows
```

Formula expression in modeling

```
model <- lm(price ~ cut:color + carat + log(depth),
            data = diamonds)
```

- Select outcome & predictors

Formula expression in modeling

```
model <- lm(price ~ cut:color + carat + log(depth),
            data = diamonds)
```

- Select outcome & predictors

- Operators to matrix of predictors

Formula expression in modeling

```
model <- lm(price ~ cut:color + carat + log(depth),
            data = diamonds)
```

- Select outcome & predictors
- Operators to matrix of predictors
- Inline functions

Work under the hood - model.matrix

```
model.matrix(price ~ cut:color + carat + log(depth) + table,
             data = diamonds)
```

```
## Rows: 53,940
## Columns: 39
## $ `(Intercept)`           <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, …
## $ carat                   <dbl> 0.23, 0.21, 0.23, 0.29, 0.31, 0.24, 0.24, 0.26, …
## $ `log(depth)`            <dbl> 4.119037, 4.091006, 4.041295, 4.133565, 4.147885…
## $ table                   <dbl> 55, 61, 65, 58, 58, 57, 57, 55, 61, 61, 55, 56, …
## $ `cutFair:colorD`        <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, …
## $ `cutGood:colorD`        <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, …
## $ `cutVery Good:colorD`   <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, …
## $ `cutPremium:colorD`     <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, …
## $ `cutIdeal:colorD`       <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, …
## $ `cutFair:colorE`        <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, …
## $ `cutGood:colorE`        <dbl> 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, …
## $ `cutVery Good:colorE`   <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, …
## $ `cutPremium:colorE`     <dbl> 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, …
## $ `cutIdeal:colorE`       <dbl> 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, …
## $ `cutFair:colorF`        <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, …
## $ `cutGood:colorF`        <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, …
## $ `cutVery Good:colorF`   <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, …
```

Downsides

- Tedious typing with many variables

Downsides

- Tedious typing with many variables
- **Functions have to manually be applied to each variable**

```
lm(y ~ log(x01) + log(x02) + log(x03) + log(x04) + log(x05) + log(x06) + log(x07) +
       log(x08) + log(x09) + log(x10) + log(x11) + log(x12) + log(x13) + log(x14) +
       log(x15) + log(x16) + log(x17) + log(x18) + log(x19) + log(x20) + log(x21) +
       log(x22) + log(x23) + log(x24) + log(x25) + log(x26) + log(x27) + log(x28) +
       log(x29) + log(x30) + log(x31) + log(x32) + log(x33) + log(x34) + log(x35),
   data = dat)
```

Downsides

- Tedious typing with many variables
- Functions have to manually be applied to each variable
- **Operations are constrained to single columns**

```
# Not possible
lm(y ~ pca(x01, x02, x03, x04, x05), data = dat)
```

Downsides

- Tedious typing with many variables
- Functions have to manually be applied to each variable
- Operations are constrained to single columns
- **Everything happens at once**

You can't apply multiple transformations to the same variable.

Downsides

- Tedious typing with many variables
- Functions have to manually be applied to each variable
- Operations are constrained to single columns
- Everything happens at once
- **Connected to the model, calculations are not saved between models**

One could manually use `model.matrix` and pass the result to the modeling function.

recipes

www.rstudio.com

Recipes

New package to deal with this problem

Benefits:

- Modular

Recipes

New package to deal with this problem

Benefits:

- Modular
- pipeable

Recipes

New package to deal with this problem

Benefits:

- Modular
- pipeable
- Deferred evaluation

Recipes

New package to deal with this problem

Benefits:

- Modular
- pipeable
- Deferred evaluation
- Isolates test data from training data

Recipes

New package to deal with this problem

Benefits:

- Modular
- pipeable
- Deferred evaluation
- Isolates test data from training data
- **Can do things formulas can't**

Modularity and pipeability

```
price ~ cut + color + carat + log(depth) + table
```

Taking the formula from before we can rewrite it as the following recipe

```
rec <- recipe(price ~ cut + color + carat + depth + table,
              data = diamonds) %>%
  step_log(depth) %>%
  step_dummy(cut, color)
```

Modularity and pipeability

```
price ~ cut + color + carat + log(depth) + table
```

Taking the formula from before we can rewrite it as the following recipe

```
rec <- recipe(price ~ cut + color + carat + depth + table,
              data = diamonds) %>%
  step_log(depth) %>%
  step_dummy(cut, color)
```

formula expression to specify variables

Modularity and pipeability

```
price ~ cut + color + carat + log(depth) + table
```

Taking the formula from before we can rewrite it as the following recipe

```
rec <- recipe(price ~ cut + color + carat + depth + table,
              data = diamonds) %>%
  step_log(depth) %>%
  step_dummy(cut, color)
```

then apply log transformation on `depth`

Modularity and pipeability

```
price ~ cut + color + carat + log(depth) + table
```

Taking the formula from before we can rewrite it as the following recipe

```
rec <- recipe(price ~ cut + color + carat + depth + table,
              data = diamonds) %>%
  step_log(depth) %>%
  step_dummy(cut, color)
```

lastly we create dummy variables from `cut` and `color`

Deferred evaluation

If we look at the recipe we created we don't see a dataset, but instead, we see a specification

```
rec
```

```
## Data Recipe
##
## Inputs:
##
##       role #variables
##    outcome          1
##  predictor          5
##
## Operations:
##
## Log transformation on depth
## Dummy variables from cut, color
```

Deferred evaluation

recipes gives a specification of the intent of what we want to do.

No calculations have been carried out yet.

First, we need to `prep()` the recipe. This will calculate the sufficient statistics needed to perform each of the steps.

```
prepped_rec <- prep(rec)
```

Deferred evaluation

```
prepped_rec
```

```
## Data Recipe
##
## Inputs:
##
##       role #variables
##    outcome          1
##  predictor          5
##
## Training data contained 53940 data points and no missing data.
##
## Operations:
##
## Log transformation on depth [trained]
## Dummy variables from cut, color [trained]
```

Baking

After we have prepped the recipe we can `bake()` it to apply all the transformations

```
bake(prepped_rec, new_data = diamonds)
```

```
## Rows: 53,940
## Columns: 14
## $ carat   <dbl> 0.23, 0.21, 0.23, 0.29, 0.31, 0.24, 0.24, 0.26, 0.22, 0.23, 0.…
## $ depth   <dbl> 4.119037, 4.091006, 4.041295, 4.133565, 4.147885, 4.139955, 4.…
## $ table   <dbl> 55, 61, 65, 58, 58, 57, 57, 55, 61, 61, 55, 56, 61, 54, 62, 58…
## $ price   <int> 326, 326, 327, 334, 335, 336, 336, 337, 337, 338, 339, 340, 34…
## $ cut_1   <dbl> 0.6324555, 0.3162278, -0.3162278, 0.3162278, -0.3162278, 0.000…
## $ cut_2   <dbl> 0.5345225, -0.2672612, -0.2672612, -0.2672612, -0.2672612, -0.…
## $ cut_3   <dbl> 3.162278e-01, -6.324555e-01, 6.324555e-01, -6.324555e-01, 6.32…
## $ cut_4   <dbl> 0.1195229, -0.4780914, -0.4780914, -0.4780914, -0.4780914, 0.7…
## $ color_1 <dbl> -3.779645e-01, -3.779645e-01, -3.779645e-01, 3.779645e-01, 5.6…
## $ color_2 <dbl> 9.690821e-17, 9.690821e-17, 9.690821e-17, 0.000000e+00, 5.4554…
## $ color_3 <dbl> 4.082483e-01, 4.082483e-01, 4.082483e-01, -4.082483e-01, 4.082…
## $ color_4 <dbl> -0.5640761, -0.5640761, -0.5640761, -0.5640761, 0.2417469, 0.2…
## $ color_5 <dbl> 4.364358e-01, 4.364358e-01, 4.364358e-01, -4.364358e-01, 1.091…
## $ color_6 <dbl> -0.19738551, -0.19738551, -0.19738551, -0.19738551, 0.03289758…
```

Baking / Juicing

Since the dataset is already calculated after running `prep()` can we use `juice()` to extract it

```
juice(prepped_rec)
```

```
## Rows: 53,940
## Columns: 14
## $ carat   <dbl> 0.23, 0.21, 0.23, 0.29, 0.31, 0.24, 0.24, 0.26, 0.22, 0.23, 0.…
## $ depth   <dbl> 4.119037, 4.091006, 4.041295, 4.133565, 4.147885, 4.139955, 4.…
## $ table   <dbl> 55, 61, 65, 58, 58, 57, 57, 55, 61, 61, 55, 56, 61, 54, 62, 58…
## $ price   <int> 326, 326, 327, 334, 335, 336, 336, 337, 337, 338, 339, 340, 34…
## $ cut_1   <dbl> 0.6324555, 0.3162278, -0.3162278, 0.3162278, -0.3162278, 0.000…
## $ cut_2   <dbl> 0.5345225, -0.2672612, -0.2672612, -0.2672612, -0.2672612, -0.…
## $ cut_3   <dbl> 3.162278e-01, -6.324555e-01, 6.324555e-01, -6.324555e-01, 6.32…
## $ cut_4   <dbl> 0.1195229, -0.4780914, -0.4780914, -0.4780914, -0.4780914, 0.7…
## $ color_1 <dbl> -3.779645e-01, -3.779645e-01, -3.779645e-01, 3.779645e-01, 5.6…
## $ color_2 <dbl> 9.690821e-17, 9.690821e-17, 9.690821e-17, 0.000000e+00, 5.4554…
## $ color_3 <dbl> 4.082483e-01, 4.082483e-01, 4.082483e-01, -4.082483e-01, 4.082…
## $ color_4 <dbl> -0.5640761, -0.5640761, -0.5640761, -0.5640761, 0.2417469, 0.2…
## $ color_5 <dbl> 4.364358e-01, 4.364358e-01, 4.364358e-01, -4.364358e-01, 1.091…
```

recipes workflow

```
recipe -> prepare -> bake/juice

(define) -> (estimate) -> (apply)
```

Isolates test & training data

When working with data for predictive modeling it is important to make sure any information from the test data leaks into the training data.

This is avoided by using **recipes** by making sure you only prep the recipe with the training dataset.

Can do things formulas can't

selectors

It can be annoying to manually specify variables by name.

The use of selectors can greatly help you!

```
rec <- recipe(price ~ ., data = diamonds)
%>%
  step_dummy(all_nominal()) %>%
  step_zv(all_numeric()) %>%
  step_center(all_predictors())
```

selectors

`all_nominal()` is used to select all the nominal variables.

```
rec <- recipe(price ~ ., data = diamonds)
%>%
  step_dummy(all_nominal()) %>%
  step_zv(all_numeric()) %>%
  step_center(all_predictors())
```

selectors

`all_numeric()` is used to select all the numeric variables.

Even the ones generated by `step_dummy()`

```
rec <- recipe(price ~ ., data = diamonds)
%>%
  step_dummy(all_nominal()) %>%
  step_zv(all_numeric()) %>%
  step_center(all_predictors())
```

selectors

`all_predictors()` is used to select all predictor variables.

Will not break even if a variable is removed with `step_zv()`

```
rec <- recipe(price ~ ., data = diamonds)
%>%
  step_dummy(all_nominal()) %>%
  step_zv(all_numeric()) %>%
  step_center(all_predictors())
```

Roles

`update_role()` can be used to give variables roles.

That then can be selected with `has_role()`

Roles can also be set with `role =` argument inside steps

```
rec <- recipe(price ~ ., data = diamonds)
%>%
  update_role(x, y, z, new_role = "size")
%>%
  step_log(has_role("size")) %>%
  step_dummy(all_nominal()) %>%
  step_zv(all_numeric()) %>%
  step_center(all_predictors())
```

PCA extraction

```
rec <- recipe(price ~ ., data = diamonds) %>%
  step_dummy(all_nominal()) %>%
  step_scale(all_predictors()) %>%
  step_center(all_predictors()) %>%
  step_pca(all_predictors(), threshold = 0.8)
```

You can also write a recipe that extract enough principal components to explain 80% of the variance

Loadings will be kept in the prepped recipe to make sure other datasets are transformed correctly

Imputation

recipes does by default NOT deal with missing data.

There are many steps to perform imputation, some include `step_knnimpute()`, `step_meanimpute()` and `step_medianimpute()` for numerics and `step_unknown()` for factors.

What kind of transformations should I do?

Many kinds of transformation, but a few of the important categories are here:

- Dummy
- Zero Variance
- Impute
- Decorrelate
- Normalize
- Transform

Dummy

Do qualitative predictors require a numeric encoding (e.g. via dummy variables or other methods).

When you have a categorical variables, there are times where we need to turn them into numbers to use them inn our model

Considerr the `MS_Zoning` variable from the `ames` data set

```
library(modeldata)
data(ames)
levels(ames$MS_Zoning)
```

```
## [1] "Floating_Village_Residential" "Residential_High_Density"
## [3] "Residential_Low_Density"      "Residential_Medium_Density"
## [5] "A_agr"                        "C_all"
## [7] "I_all"
```

Dummy

One way to deal with this is to **dummify** the data, this shows a 1 if the level is present and 0 otherwise. We use `step_dummy()`

Notice how there is a column for each category

```
## Rows: 2,930
## Columns: 7
## $ MS_Zoning_Floating_Village_Residential <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0…
## $ MS_Zoning_Residential_High_Density     <dbl> 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0…
## $ MS_Zoning_Residential_Low_Density      <dbl> 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1…
## $ MS_Zoning_Residential_Medium_Density   <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0…
## $ MS_Zoning_A_agr                        <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0…
## $ MS_Zoning_C_all                        <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0…
## $ MS_Zoning_I_all                        <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0…
```

Dummy

Having a column for each category is not necessary and can cause rank issues, setting `one_hot = FALSE` sets the first level as the reference level

```
## Rows: 2,930
## Columns: 6
## $ MS_Zoning_Residential_High_Density   <dbl> 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, …
## $ MS_Zoning_Residential_Low_Density    <dbl> 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, …
## $ MS_Zoning_Residential_Medium_Density <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, …
## $ MS_Zoning_A_agr                      <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, …
## $ MS_Zoning_C_all                      <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, …
## $ MS_Zoning_I_all                      <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, …
```

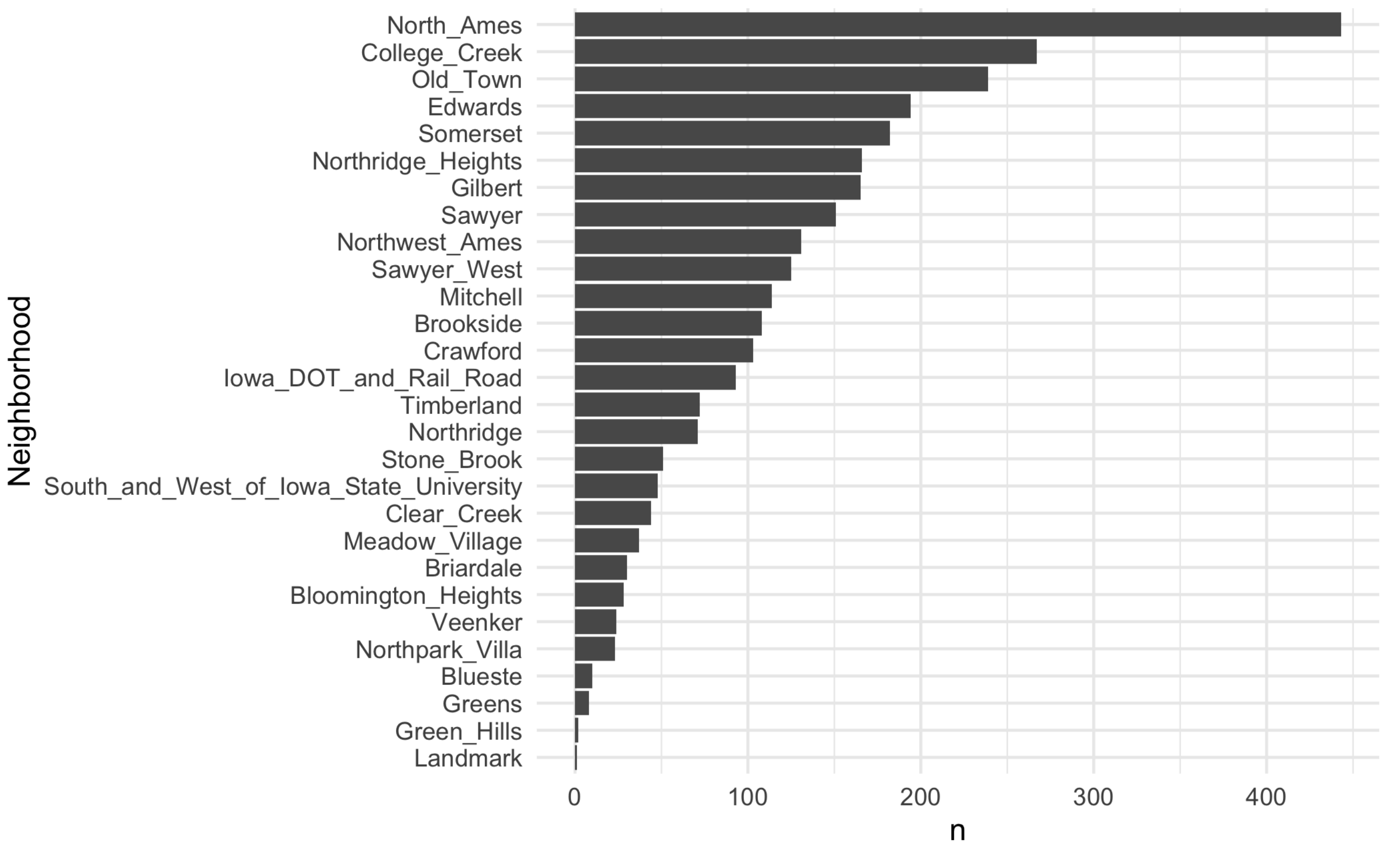
Dummy

Consider now the `Neighborhood` column, this column contains 29 levels, which gives us a lot of dummy variables

```
## Rows: 2,930
## Columns: 28
## $ Neighborhood_College_Creek                         <dbl> 0, 0, 0, 0, 0, 0,…
## $ Neighborhood_Old_Town                              <dbl> 0, 0, 0, 0, 0, 0,…
## $ Neighborhood_Edwards                               <dbl> 0, 0, 0, 0, 0, 0,…
## $ Neighborhood_Somerset                              <dbl> 0, 0, 0, 0, 0, 0,…
## $ Neighborhood_Northridge_Heights                    <dbl> 0, 0, 0, 0, 0, 0,…
## $ Neighborhood_Gilbert                               <dbl> 0, 0, 0, 0, 1, 1,…
## $ Neighborhood_Sawyer                                <dbl> 0, 0, 0, 0, 0, 0,…
## $ Neighborhood_Northwest_Ames                        <dbl> 0, 0, 0, 0, 0, 0,…
## $ Neighborhood_Sawyer_West                           <dbl> 0, 0, 0, 0, 0, 0,…
## $ Neighborhood_Mitchell                              <dbl> 0, 0, 0, 0, 0, 0,…
## $ Neighborhood_Brookside                             <dbl> 0, 0, 0, 0, 0, 0,…
## $ Neighborhood_Crawford                              <dbl> 0, 0, 0, 0, 0, 0,…
## $ Neighborhood_Iowa_DOT_and_Rail_Road                <dbl> 0, 0, 0, 0, 0, 0,…
## $ Neighborhood_Timberland                            <dbl> 0, 0, 0, 0, 0, 0,…
## $ Neighborhood_Northridge                            <dbl> 0, 0, 0, 0, 0, 0,…
## $ Neighborhood_Stone_Brook                           <dbl> 0, 0, 0, 0, 0, 0,…
```

Dummy

Many of the levels don't appear that often!



Dummy

Many of the levels don't appear that often! One way to deal with this is to **lump** together all the infrequent levels together

Dummy

This can be done using the `step_other()` and the `threshold` argument

```
## Rows: 2,930
## Columns: 8
## $ Neighborhood_College_Creek      <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,…
## $ Neighborhood_Old_Town           <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,…
## $ Neighborhood_Edwards            <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,…
## $ Neighborhood_Somerset           <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,…
## $ Neighborhood_Northridge_Heights <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,…
## $ Neighborhood_Gilbert            <dbl> 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 1, 1,…
## $ Neighborhood_Sawyer             <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,…
## $ Neighborhood_other              <dbl> 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0,…
```

Dummy

Novel levels. The use of `step_novel()`

If there is a level that doesn't appear in the training data set but does appear later, you get into a problem. `step_novel()` handles that problem by assigning previously unseen factor levels to a new value.

Zero Variance

Should columns with a single unique value be removed?

We can do this with `step_vz()`

or with `step_nzv()` to remove the variables with Near Zero Variance

Impute

If some predictors are missing, should they be estimated via imputation?

Normalize

Should predictors be centered and scaled?

In recipes we have

- `step_center()` Centering numeric data
- `step_normalize()` Center and scale numeric data
- `step_range()` Scaling Numeric Data to a Specific Range
- `step_scale()` Scaling Numeric Data