Hands-on Machine Learning with R

3 Feature and Target Engineering

Preprocessing your data before modelling can significantly affect the model performance.

3.1 Prerequisites

```
# Helper packages
library(dplyr)    # for data manipulation
library(ggplot2)  # for awesome graphics
library(visdat)   # for additional visualizations
library(rsample)  # for splitting data

# Feature engineering packages
library(caret)    # for various ML tasks
library(recipes)  # for feature engineering tasks

# load ames housing data
ames <- AmesHousing::make_ames()

# Stratified sampling with the rsample package
set.seed(123)
split <- initial_split(ames, prop = 0.7,
                       strata = "Sale_Price")
ames_train  <- training(split)
ames_test   <- testing(split)
```

3.2 Target engineering

Especially with parametric models, you might want to transform your target variable e.g. to make it normal with a log-transformation if the model's assumptions are that the errors are normally distributed (and therefore the target as well).

Additionally if you log-transform the response, this means that errors on high and low vales are treated equally – this is equivalent to using RMSLE loss funciton instead of RMSE.

Option 1: log-transform the outcome. Either directly in the dataset. Alternatively, think of preprocessing as creating a blueprint that will be applied later. Using the `recipe` package:

```
# log transformation
ames_recipe <- recipe(Sale_Price ~ ., data = ames_train) %>%
  step_log(all_outcomes(), offset = 1)

ames_recipe
```

```
## Data Recipe
##
## Inputs:
##
##       role #variables
##    outcome          1
##  predictor         80
##
## Operations:
##
```

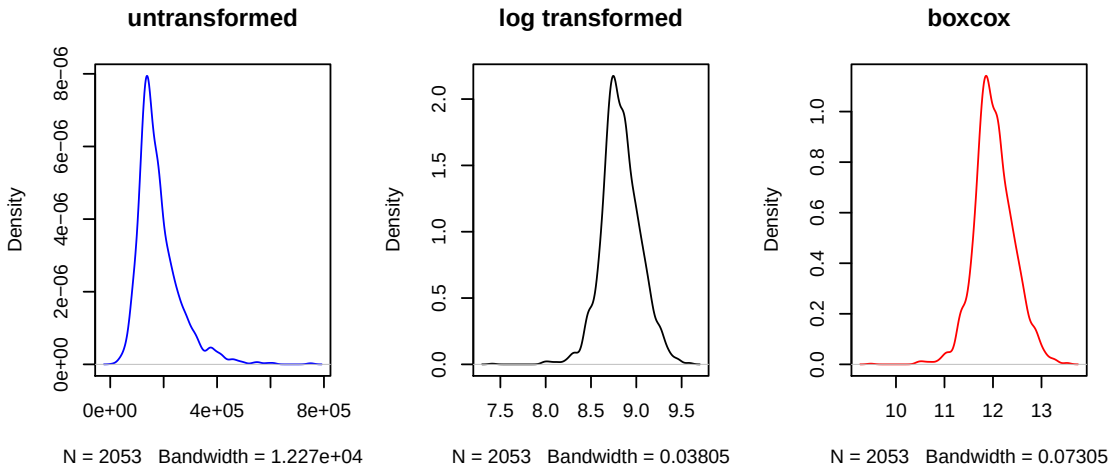


Figure 1: Distribution of target variable in train set untransformed, log transfrormed and box cox

```
## Log transformation on all_outcomes
```

You can add an `offset` in the `step_log()` function to add +1 to all values if you have zeros or small negative values you are logging. If the values are more negative, then you can use the Yeo-Johnson transformation described below.

Option 2: use a *Box-Cox* transformation. It's more powerful than just a log (which is a special case of it anyway). The transformation uses an exponent *lambda* (λ), and the optimal value is estimated from the training data, to produce a transformation closest to normal. You want to make sure you use the same lanbda in the training and test sets, `recipes` automates this for you though.

Of course if you transform your response, you will want to undo that when you're interpreting your results, don't forget that.

!! code error: `lambda` instead of `lambda = "auto"` in the Box Cox call.!!

3.3 Dealing with missingness

Distinguish between *informative missingness* and *random missingness*. The reason behind the missing data will drive how we treat them. For example we might give informative missing values their own category e.g. "none" and let them be a predictor in their own right. Random missing values can either be deleted or imputed.

Most ML algs cannot handle missing values, so you need to deal with them beforehand. Some models, mainly tree-based ones, have procedures built in to handle them though. But if you are comparing multiple models you will want to deal with NAs before, so you can compare the models fairly based on the same data quality assumptions.

3.3.1 Visualising missing values

The raw, uncleaned ames housing dataset actually has almost 14,000 missing values.

```
sum(is.na(AmesHousing::ames_raw))
```

```
## [1] 13997
```

Visualising the distribution of missing values is the first step to figuring out how to deal with them. We can use base graphics `heatmap()` to do this.

Or ggplot

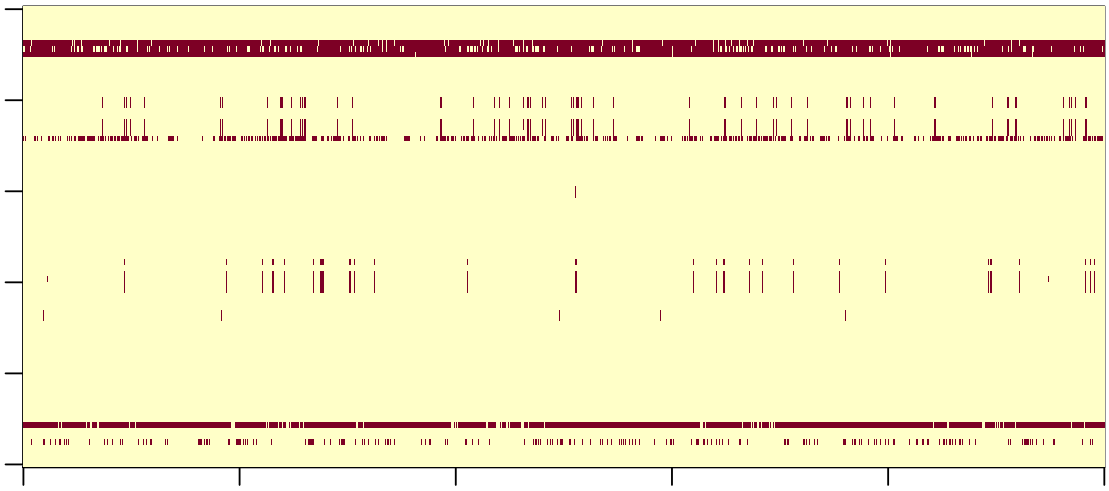


Figure 2: Distribution of missing values in raw Ames data

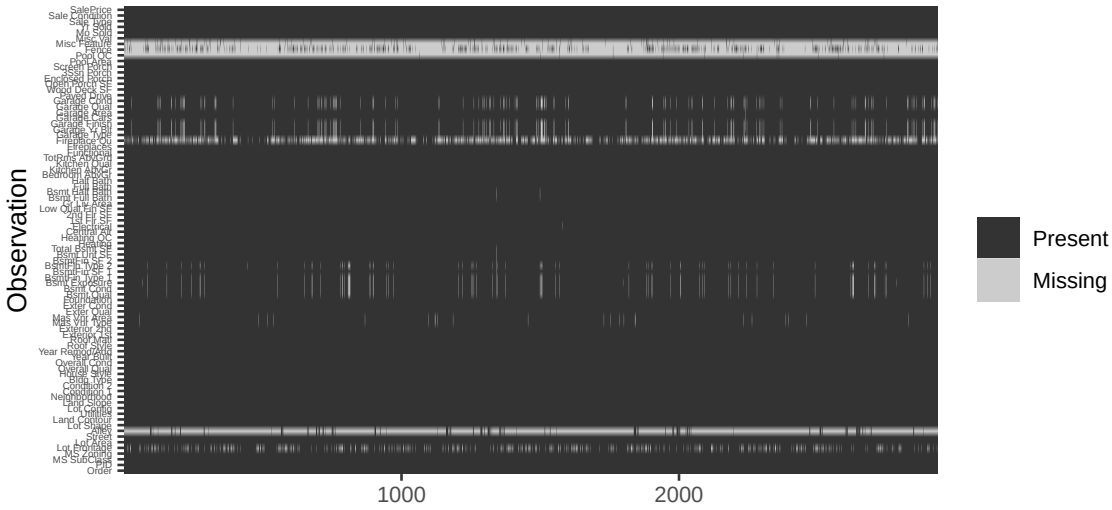


Figure 3: Distribution of missing values in raw Ames data

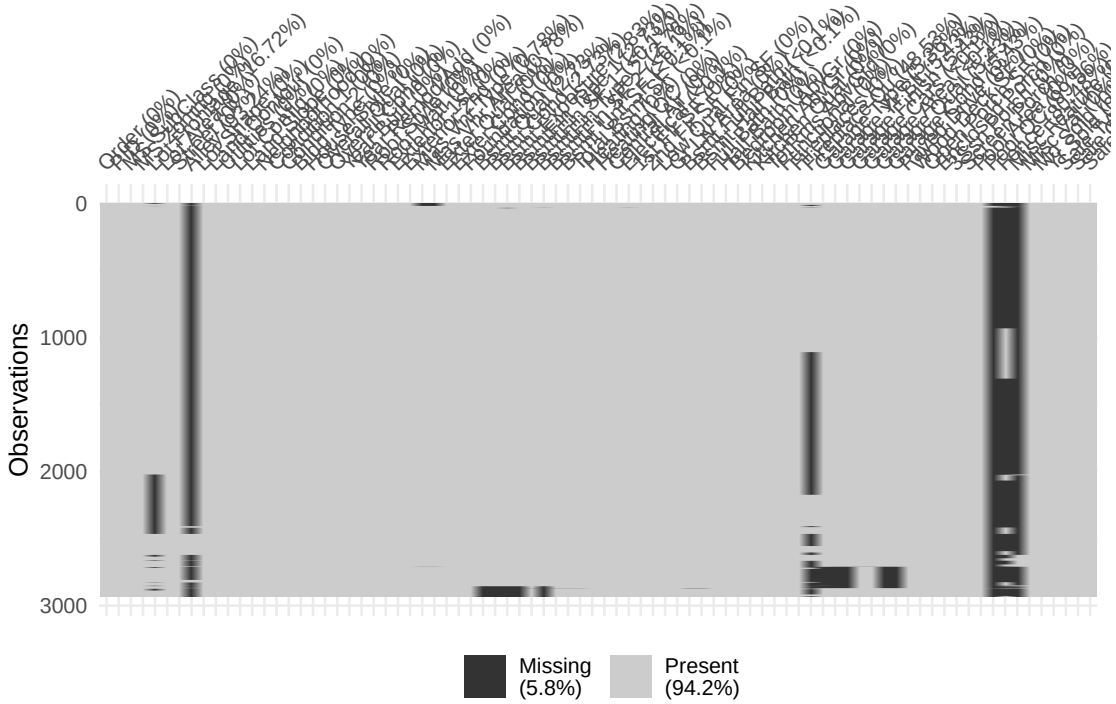


Figure 4: Distribution of missing values in raw Ames data

Looking more closely we can see that whenever the `Garage_type` is NA, the `Garage_area` and other associated variables are 0.

```
raw_ames <- AmesHousing::ames_raw
raw_ames %>%
  filter(is.na(`Garage Type` ) )%>%
  select(`Garage Type`,`Garage Area`, `Garage Cars`) %>%
  head()
```

```
## # A tibble: 6 x 3
##   `Garage Type` `Garage Area` `Garage Cars`
##   <chr>                 <int>         <int>
## 1 <NA>                      0             0
## 2 <NA>                      0             0
## 3 <NA>                      0             0
## 4 <NA>                      0             0
## 5 <NA>                      0             0
## 6 <NA>                      0             0
```

This could mean that the missingness is informative and means there is no garage, not that the data isn't available, so we might want to reclassify those NAs as "None" or sth.

Another way to visualise the missingness is using the `vis_mis` from `visdata`. Using `cluster = TRUE` groups the observations with missing data together

3.3.2 Imputation

This is a *feature engineering* step, one that should be one of the first you undertake, because it affects everything downstream.

3.3.2.1 Estimated statistic

- an elementary approach is to calculate a mean, or mode or median, and use that to replace the NAs. But htis ignores the other attributes of an observation we are imputing

```
# add a simple median imputation to the recipe
ames_recipe %>%
  step_medianimpute(Gr_Liv_Area) -> ames_recipe
```

- an alternative is to used grouped statistics to capture the expected values for smaller groups. But this becomes unfeasible with large datasets. (why?)

Before we get to the more efficient approaches, note that model based imputation needs to be performed **within the resampling process**, which means repeatedly, so be careful about how much of this you want to do.

3.3.2.2 K-nearest neighbours imputation

Identifies missing observaitons, finds most similar observations based on other attributes and uses these neighbours to assign a value.

KNN imputation treats the missing observation as the targeted response and predicts it based on the neighbours features.

If all feaures are quantitative, then standard Euclidean distance is usually used. And if there is a mix, then *Gower's distance* is usually used.

Of course k is also a tunable parameter, the default used by `step_knnimpute()` is 5, but can be changed using the `neighbours` argument.

```
ames_recipe %>%
  step_knnimpute(all_predictors(), neighbors = 6)
```

```
## Data Recipe
##
## Inputs:
##
##       role #variables
##    outcome          1
##  predictor         80
##
## Operations:
##
## Log transformation on all_outcomes
## Median Imputation for Gr_Liv_Area
## K-nearest neighbor imputation for all_predictors
```

3.3.2.3 Tree-based

A lot of tree based models can be constructed in the presence of missing values. Inddividual trees have high variance, but aggregating them is m,ore robust. Random forest imputation is too costly though, but bagging seems like a good compromise.

Same as KNN imputation, the observation with the missing value is identified and treated as the target, and is predicted using bagged decision trees.

```
ames_recipe %>%
  step_bagimpute(`Gr Liv Area`)
```

```
## Data Recipe
##
## Inputs:
##
##       role #variables
##    outcome          1
##  predictor         80
```

```
##
## Operations:
##
## Log transformation on all_outcomes
## Median Imputation for Gr_Liv_Area
## Bagged tree imputation for Gr Liv Area
```

3.4 Feature Filtering

Too many predictors, especially non-informative ones, can negatively affect model performace. Not for all models, e.g. tree based or lasso are fine. But even so, it will affect the time needed to run the models.

Easiest to eliminate are zero variance variables, or ones close to zero. They offer no discriminating power, no infomration. For some algs this doesn't matter, but it slows others down.

The ones with low variance obvs have some infomration, but not a lot and can cause problems with resampling, when they can become effectively zero variance in individual samples.

Good rules of thumbs for detecting variables with low vairance is :

- the fraction of unique values is unde 10%
- the ration of the most prevalent to the second most prevalent value is large (> 20%) (what does the twenty percent mean? the inverse?) [Actually no, in the function below the ratio is 95/5. !!!]

If both of these conditions are met, it might be good to remove the variables. You can use `caret::nearZeroVar()` to investigate which have both.

```
caret::nearZeroVar(ames_train, saveMetrics = TRUE) %>%
  tibble::rownames_to_column() %>%
  filter(nzv)
```

```
##               rowname  freqRatio percentUnique zeroVar  nzv
## 1              Street  292.28571    0.09741841   FALSE TRUE
## 2               Alley   20.52688    0.14612762   FALSE TRUE
## 3       Land_Contour   22.28916    0.19483682   FALSE TRUE
## 4          Utilities 1025.00000    0.14612762   FALSE TRUE
## 5         Land_Slope   22.76744    0.14612762   FALSE TRUE
## 6        Condition_2  203.10000    0.34096444   FALSE TRUE
## 7          Roof_Matl  126.50000    0.24354603   FALSE TRUE
## 8          Bsmt_Cond   19.93478    0.29225524   FALSE TRUE
## 9      BsmtFin_Type_2   21.50617    0.34096444   FALSE TRUE
## 10            Heating  101.05000    0.24354603   FALSE TRUE
## 11    Low_Qual_Fin_SF 1013.00000    1.31514856   FALSE TRUE
## 12      Kitchen_AbvGr   23.68675    0.19483682   FALSE TRUE
## 13         Functional   38.18000    0.34096444   FALSE TRUE
## 14     Enclosed_Porch  100.94118    7.40379932   FALSE TRUE
## 15 Three_season_porch  674.66667    1.16902094   FALSE TRUE
## 16       Screen_Porch  234.87500    4.52995616   FALSE TRUE
## 17          Pool_Area 2045.00000    0.43838285   FALSE TRUE
## 18            Pool_QC  681.66667    0.24354603   FALSE TRUE
## 19       Misc_Feature   30.49231    0.19483682   FALSE TRUE
## 20           Misc_Val  165.33333    1.41256698   FALSE TRUE
```

You can add `step_zv` or `step_nzv` to the recipe and remove the variables with zero or near zero variance.

```
ames_recipe %>%
  step_nzv(all_predictors())
```

```
## Data Recipe
##
## Inputs:
##
##       role #variables
##    outcome          1
##  predictor         80
##
## Operations:
##
## Log transformation on all_outcomes
## Median Imputation for Gr_Liv_Area
## Sparse, unbalanced variable filter on all_predictors
```

3.5 Numeric feature engineering

Issues with skewness and different magnitudes of variable value ranges can cause a lot of issues with some models, although not really with tree-based ones. Normalizing and standardizing aleviates these problems.

3.5.1 Normalizing skewness

This is important especially for parametric models (although it won't hurt for non-parametric ones). If doing many variables best use BoxCox (if the values are all positive) or Yeo-Johnson (if they are negative), as they identify the optimal transformation.

```
ames_recipe %>%
  step_nzv(all_predictors()) %>%
step_YeoJohnson(all_numeric())
```

```
## Data Recipe
##
## Inputs:
##
##       role #variables
##    outcome          1
##  predictor         80
##
## Operations:
##
## Log transformation on all_outcomes
## Median Imputation for Gr_Liv_Area
## Sparse, unbalanced variable filter on all_predictors
## Yeo-Johnson transformation on all_numeric
```

3.5.2 Standardization

Do the scales of the inputs vary a lot? Models that use smooth (linear) functions of input features (some more obviously than others) will be sensitive to this. Not only GLMs, but also NNs, SVM, PCA.. Also ones that use distance e.g. k-nearest neighbours, k-means clustering..

So for those cases it's a good idea to strandardize the variables, centering so they have zero mean and scaling so they have unit variance.

Some packages e.g. `glmnet` or `caret` have built in functionality for standardizing, but you really want to standardize the data in the recipe, so that both training and test standardization are based on the same mean and variance. *This helps minimise data leakage.*

```
ames_recipe %>%
  step_center(all_numeric()) %>%
  step_scale(all_numeric())
```

```
## Data Recipe
##
## Inputs:
##
##       role #variables
##    outcome          1
##  predictor         80
##
## Operations:
##
## Log transformation on all_outcomes
## Median Imputation for Gr_Liv_Area
## Centering for all_numeric
## Scaling for all_numeric
```

3.6 Categorical feature engineering

Most models require features be numeric, but some can handle categorical variables as well, especially tree-based ones, but even these can benefit from preprocessing them.

3.6.1 Lumping

For example you can have categories with very few observations like here:

```
count(ames_train, Neighborhood) %>% arrange(n)
```

```
## # A tibble: 27 x 2
##    Neighborhood                                n
##    <fct>                                   <int>
##  1 Green_Hills                                 2
##  2 Greens                                      7
##  3 Blueste                                     8
##  4 Northpark_Villa                            17
##  5 Briardale                                  18
##  6 Veenker                                    20
##  7 Bloomington_Heights                        21
##  8 South_and_West_of_Iowa_State_University    27
##  9 Meadow_Village                             29
## 10 Clear_Creek                                31
## # ... with 17 more rows
```

This happens with numeric variables as well, e.g here, where most observations have a zero and only 8 % have a valid number.

```
count(ames_train, Screen_Porch)
```

```
## # A tibble: 93 x 2
##    Screen_Porch     n
##           <int> <int>
##  1            0  1879
##  2           40     1
##  3           63     1
##  4           80     1
##  5           90     3
##  6           92     1
```

```
##  7           94     1
##  8           95     2
##  9           99     1
## 10          100     3
## # ... with 83 more rows
```

In these cases you can benefit from collapsing these small categories (why, are you not loosing important data here?). But yeah, you should use this sparingly "as there is often a loss in model performace".

You can use `step_other` to merge small categories into an "Other" category.

```
# Lump levels for two features
lumping <- recipe(Sale_Price ~ ., data = ames_train) %>%
  step_other(Neighborhood, threshold = 0.01,
             other = "other") %>%
  step_other(Screen_Porch, threshold = 0.1,
             other = ">0")

# Apply this blue print --> you will learn about this at
# the end of the chapter
apply_2_training <- prep(lumping, training = ames_train) %>%
  bake(ames_train)

# New distribution of Neighborhood
count(apply_2_training, Neighborhood) %>% arrange(n)
```

```
## # A tibble: 22 x 2
##    Neighborhood                                n
##    <fct>                                   <int>
##  1 Bloomington_Heights                        21
##  2 South_and_West_of_Iowa_State_University    27
##  3 Meadow_Village                             29
##  4 Clear_Creek                                31
##  5 Stone_Brook                                34
##  6 Northridge                                 48
##  7 Timberland                                 55
##  8 Iowa_DOT_and_Rail_Road                     62
##  9 Crawford                                   72
## 10 other                                      72
## # ... with 12 more rows
```

```
# new distribution of Screen Porch
count(apply_2_training, Screen_Porch)
```

```
## # A tibble: 2 x 2
##   Screen_Porch     n
##   <fct>        <int>
## 1 0             1879
## 2 >0             174
```

3.6.2 One-hot and dummy implementation

If your models require numeric variables you need to transform the categorical ones into them. `h2o` and `caret` handle this internally, but `keras` and `glmnet` do not.

One-hot encoding is the most common type, where you transform a categorical var into several boolean variables with 1 for the category and 0 for !category. *Full rank* encoding transforms a variable into as many variables as there are categories. But this makes the new variables

perfectly colinear, which causes problems with some models (OLS, NN). *Dummy encoding* drops one of the variables to remove the colinearity.

Use `step_dummy` to do either of these, where `one_hot = TRUE` makes it full rank, and `FALSE` means it's dummy.

```
recipe(Sale_Price ~ ., data = ames_train) %>%
  step_dummy(all_nominal(), one_hot = TRUE)
```

```
## Data Recipe
##
## Inputs:
##
##       role #variables
##    outcome          1
##  predictor         80
##
## Operations:
##
## Dummy variables from all_nominal
```

Careful, since a lot of categorical variables, with a lot of categories means that one-hot encoding can explode the number of features! In that case look at other alternatives such as those below.

3.6.3 Label encoding

This is pure numeric encoding of a categorical variable (!). If the var had levels, then they will be used, otherwise it will be alphabetical. `step_integer` does this, e.g. on this variable:

```
count(ames_train, MS_SubClass)
```

```
## # A tibble: 16 x 2
##    MS_SubClass                                n
##    <fct>                                  <int>
##  1 One_Story_1946_and_Newer_All_Styles      753
##  2 One_Story_1945_and_Older                  91
##  3 One_Story_with_Finished_Attic_All_Ages     5
##  4 One_and_Half_Story_Unfinished_All_Ages     11
##  5 One_and_Half_Story_Finished_All_Ages      211
##  6 Two_Story_1946_and_Newer                 395
##  7 Two_Story_1945_and_Older                  98
##  8 Two_and_Half_Story_All_Ages               17
##  9 Split_or_Multilevel                       75
## 10 Split_Foyer                               32
## 11 Duplex_All_Styles_and_Ages                66
## 12 One_Story_PUD_1946_and_Newer             145
## 13 One_and_Half_Story_PUD_All_Ages            1
## 14 Two_Story_PUD_1946_and_Newer              96
## 15 PUD_Multilevel_Split_Level_Foyer          14
## 16 Two_Family_conversion_All_Styles_and_Ages  43
```

```
recipe(Sale_Price ~ ., data = ames_train) %>%
  step_integer(MS_SubClass) %>%
  prep(ames_train) %>%
  bake(ames_train) %>%
  count(MS_SubClass)
```

```
## # A tibble: 16 x 2
##    MS_SubClass     n
##          <dbl> <int>
```

```
##  1           1   753
##  2           2    91
##  3           3     5
##  4           4    11
##  5           5   211
##  6           6   395
##  7           7    98
##  8           8    17
##  9           9    75
## 10          10    32
## 11          11    66
## 12          12   145
## 13          13     1
## 14          14    96
## 15          15    14
## 16          16    43
```

But be careful, since most models will now treat this as an ordered numeric feature, which of course it isn't. This is fine for ordinal variables, e.g. these ones:

```
ames_train %>% select(contains("Qual"))
```

```
## # A tibble: 2,053 x 6
##    Overall_Qual Exter_Qual Bsmt_Qual Low_Qual_Fin_SF Kitchen_Qual
##    <fct>        <fct>      <fct>               <int> <fct>
##  1 Above_Avera~ Typical    Typical                 0 Typical
##  2 Average      Typical    Typical                 0 Typical
##  3 Above_Avera~ Typical    Typical                 0 Good
##  4 Above_Avera~ Typical    Typical                 0 Good
##  5 Very_Good    Good       Good                    0 Good
##  6 Very_Good    Good       Good                    0 Good
##  7 Good         Typical    Typical                 0 Good
##  8 Above_Avera~ Typical    Good                    0 Typical
##  9 Above_Avera~ Typical    Good                    0 Typical
## 10 Good         Typical    Good                    0 Good
## # ... with 2,043 more rows, and 1 more variable: Garage_Qual <fct>
```

An example of label encoding of one of these looks like this:

```
count(ames_train, Overall_Qual)
```

```
## # A tibble: 10 x 2
##    Overall_Qual       n
##    <fct>          <int>
##  1 Very_Poor          4
##  2 Poor               8
##  3 Fair              26
##  4 Below_Average    170
##  5 Average          564
##  6 Above_Average    511
##  7 Good             439
##  8 Very_Good        231
##  9 Excellent         77
## 10 Very_Excellent    23
```

```
recipe(Sale_Price ~ ., data = ames_train) %>%
  step_integer(Overall_Qual) %>%
  prep(ames_train) %>%
  bake(ames_train) %>%
```

```
  count(Overall_Qual)
```

```
## # A tibble: 10 x 2
##    Overall_Qual     n
##           <dbl> <int>
##  1            1     4
##  2            2     8
##  3            3    26
##  4            4   170
##  5            5   564
##  6            6   511
##  7            7   439
##  8            8   231
##  9            9    77
## 10           10    23
```

3.6.4 Alternatives

Target encoding is where instead of a level number, the category gets the mean (if it's a regression problem) or proportion (for classification problems) of the target value for that group. !!! not sure why these numbers are not the same as in the book, i have the same seed for the `initial_split`.

```
ames_train %>%
  group_by(Neighborhood) %>%
  summarize(Neighborhood.target = mean(Sale_Price))
```

```
## # A tibble: 27 x 2
##    Neighborhood       Neighborhood.target
##    <fct>                            <dbl>
##  1 North_Ames                     144563.
##  2 College_Creek                  199832.
##  3 Old_Town                       122737.
##  4 Edwards                        130652.
##  5 Somerset                       227380.
##  6 Northridge_Heights             323289.
##  7 Gilbert                        192163.
##  8 Sawyer                         136461.
##  9 Northwest_Ames                 187328.
## 10 Sawyer_West                    188645.
## # ... with 17 more rows
```

This represents a danger of *data leakage*, since you are using the target variable as a feature.

Alternatively, you can change the value of the feature to the proportion it represents for each category:

```
ames_train %>%
  group_by(Neighborhood) %>%
  summarize(n = n()) %>%
  mutate(Neighborhood.prop = n/sum(n))
```

```
## # A tibble: 27 x 3
##    Neighborhood           n Neighborhood.prop
##    <fct>              <int>             <dbl>
##  1 North_Ames           298            0.145
##  2 College_Creek        187            0.0911
##  3 Old_Town             171            0.0833
##  4 Edwards              146            0.0711
```

```
##  5 Somerset             128           0.0623
##  6 Northridge_Heights   115           0.0560
##  7 Gilbert              116           0.0565
##  8 Sawyer               102           0.0497
##  9 Northwest_Ames        96           0.0468
## 10 Sawyer_West           85           0.0414
## # ... with 17 more rows
```

Other options include *effect* or *likelihood encoding*, *empirical Bayes methods*, *word and entity embeddings* etc.

3.7 Dimension Reduction

This is covered later, but is a common way of pre-processing the data to filter out non-informative features. You could e.g. use PCA to select only the components that explain 95% of the variance and remove the others.

```
recipe(Sale_Price ~ ., data = ames_train) %>%
  step_center(all_numeric()) %>%
  step_scale(all_numeric()) %>%
  step_pca(all_numeric(), threshold = .95)
```

```
## Data Recipe
##
## Inputs:
##
##       role #variables
##    outcome          1
##  predictor         80
##
## Operations:
##
## Centering for all_numeric
## Scaling for all_numeric
## No PCA components were extracted.
```

3.8 Proper implementation

OK, so the idea is to prepare a blueprint for the feature engineering steps, which mforces us into thinking sequentially and appropriately applying it during the resampling process.

3.8.1 Sequential steps

Think things through and do them in the right order. Here are some tips:

- If using BoxCox, don't do anything that might make the values negative, like standardizing before. Or use Yeo-JOhnson instead and don't worry about it.
- One-hot or dummy encoding creates sparseness in the data, that makes some algs very efficient. **But** if you then standardize the data, this will make it dense (what?), which will affect the model performance. So if you want to standardize, first standardize numeric varz and only then dummy code the categorical ones.
- Obviosuly if you will do any sort of lumping, do that before one-hot encoding
- although you can do dimension reduction on categorical features it is common to do it primarily on numerical ones (not sure hwo this fits into sequential steps?!!!)

Here is a outline of steps you might want to consider:

1. filter out zero or near-zero variance features.

2. Perform imputation if required.
3. Normalize to resolve numeric feature skewness.
4. Standardize (center and scale) numeric features.
5. Perform dimension reduction (e.g., PCA) on numeric features.
6. One-hot or dummy encode categorical features.

3.8.2 Data leakage

Data leakage is when information from outside the training set is used to train the model. This often happens during pre-processing. So you need to be careful to apply the preprocessing to each resample of the training set separately, so you are not leaking data from one resample to the other. Only this way will you have a good estimate of the generalizable prediction error.

So e.g. if you're standardizing features, you should apply the mean and variance of each sample to the training data, and to the test from that sample set. This imitates how the model will be used in practice, when it will only have the current data's means and variance.

3.8.3 Putting the process together

The `recipes` package allows us to develop the blueprint of our feature engineering, and do it sequentialy. There are three main steps in creating and applying feature engineering with recipes:

1. `recipe`: where you define your feature engineering steps to create your blueprint.
2. `prepare`: estimate feature engineering parameters based on training data.
3. `bake`: apply the blueprint to new data.

In the `recipe` we supply the formula and the desired feature engineering steps in sequence. E.g. using `ames`, we want the price to be the target and all other features predictors. Then * remove all near zero varz that are nominal, * ordinally encode all features which are quality based (have Qual in the name, means they are ordinal) * center and scale all numeric variables * preform dimension reduction with pca.

```
recipe(Sale_Price ~ ., data = ames_train) %>%
         step_nzv(all_nominal()) %>%
         step_integer(matches("Qual|Cond|QC|Qu")) %>%
         step_center(all_numeric(), -all_outcomes()) %>%
         step_scale(all_numeric(), -all_outcomes())  %>%
         step_pca(all_numeric(), -all_outcomes()) -> blueprint
```

Now we need to train this blueprint on some training data. This estimates the parameters.

```
prepared <- prep(blueprint, data = ames_train)
```

Now we can apply it to new data. The training data or the new test data.

```
baked_train <- bake(prepared, new_data = ames_train)
baked_test <- bake(prepared, new_data = ames_test)
```

So this developed bluepring, we want to prep and bake it on each resample. Luckily `caret` makes this easy. You just need to specify the bluepring and it will automatically prep and bake it during resampling.

Set up the cross validation and grid search for the hyperparameter (k) as before, then call `train()`, but instead of the formula, just pass it the blueprint instead.

```
# a slightly dfferent blueprint
blueprint <- recipe(Sale_Price ~ ., data = ames_train) %>%
  step_nzv(all_nominal()) %>%
  step_integer(matches("Qual|Cond|QC|Qu")) %>%
  step_center(all_numeric(), -all_outcomes()) %>%
```

```
  step_scale(all_numeric(), -all_outcomes()) %>%
  step_dummy(all_nominal(), -all_outcomes(), one_hot = TRUE)

# Specify resampling plan
cv <- trainControl(
  method = "repeatedcv",
  number = 10,
  repeats = 5
)

# Construct grid of hyperparameter values
hyper_grid <- expand.grid(k = seq(5, 20, by = 1))

# Tune a knn model using grid search
knn_fit2 <- train(
  blueprint,
  data = ames_train,
  method = "knn",
  trControl = cv,
  tuneGrid = hyper_grid,
  metric = "RMSE"
)
```

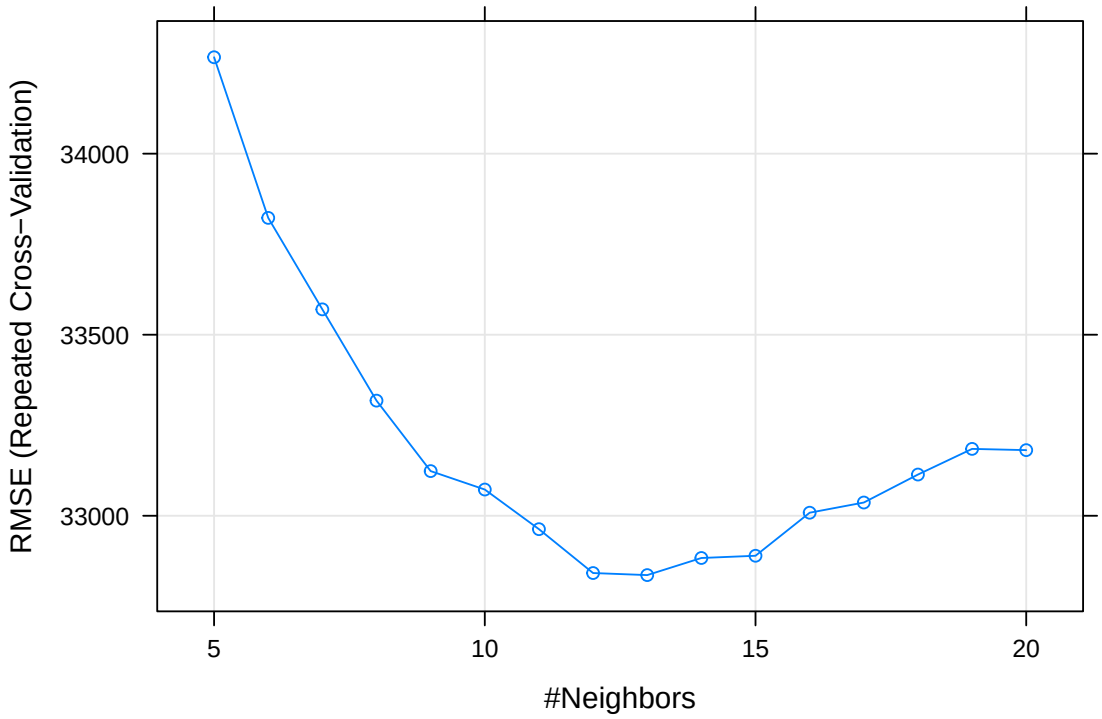
Now have a look at the summary and pring the grid search for k, which seems to be 13. The RMSE is then 32,836, compared to 43,439 in the first model.

```
#print summary
knn_fit2
```

```
## k-Nearest Neighbors
##
## 2053 samples
##   80 predictor
##
## Recipe steps: nzv, integer, center, scale, dummy
## Resampling: Cross-Validated (10 fold, repeated 5 times)
## Summary of sample sizes: 1847, 1848, 1848, 1849, 1848, 1848, ...
## Resampling results across tuning parameters:
##
##   k   RMSE      Rsquared   MAE
##    5  34266.21  0.8271187  21040.95
##    6  33822.62  0.8335150  20848.82
##    7  33569.98  0.8385875  20751.07
##    8  33317.93  0.8429423  20622.48
##    9  33123.32  0.8461772  20539.10
##   10  33072.41  0.8479929  20574.05
##   11  32962.99  0.8503286  20534.89
##   12  32841.80  0.8529277  20516.75
##   13  32836.13  0.8541457  20563.82
##   14  32883.33  0.8550141  20636.75
##   15  32889.49  0.8563039  20655.89
##   16  33008.39  0.8561262  20750.81
##   17  33036.51  0.8565350  20849.69
##   18  33113.81  0.8563328  20942.80
##   19  33184.69  0.8560424  21024.81
##   20  33181.25  0.8565618  21093.40
##
```

```
## RMSE was used to select the optimal model using the smallest value.
## The final value used for the model was k = 13.
```

```
# plot
plot(knn_fit2)
```



Because this is RMSE, the units are the same as the target, so basically adding these preprocessing methods has reduced our error by ove $10,000.