Feature and Target Engineering

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

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)
```

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

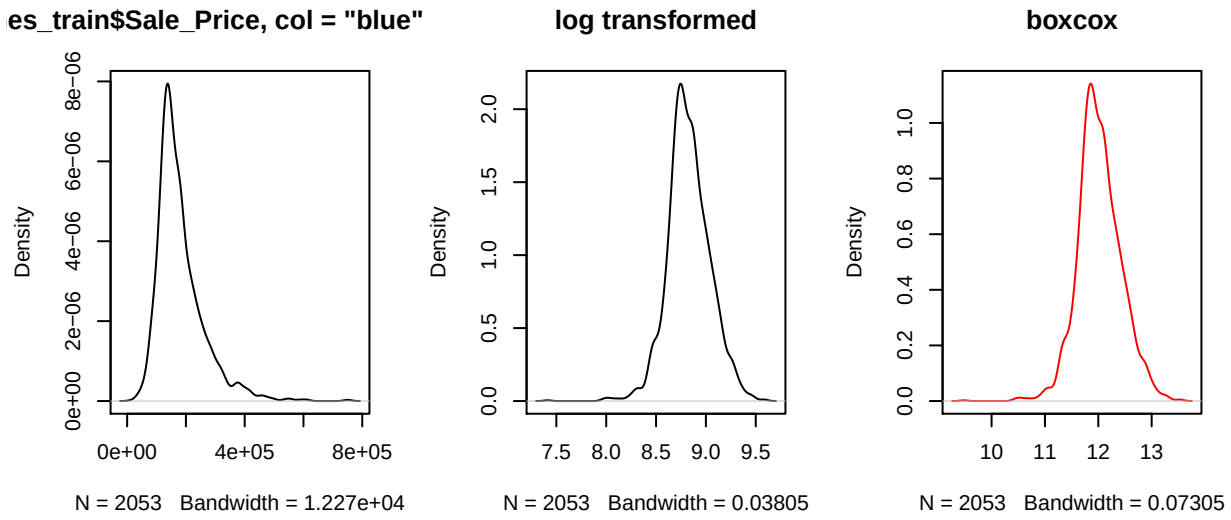

Figure 1: Distribution of target variable in train set untransformed, log transfrormed and box cox

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.!!

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.

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

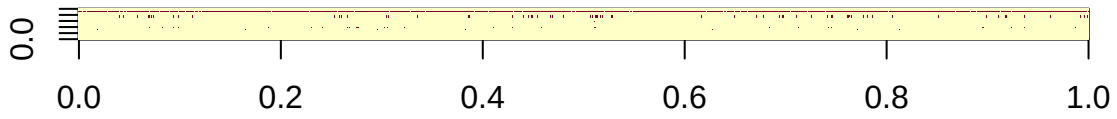

Figure 2: Distribution of missing values in raw Ames data

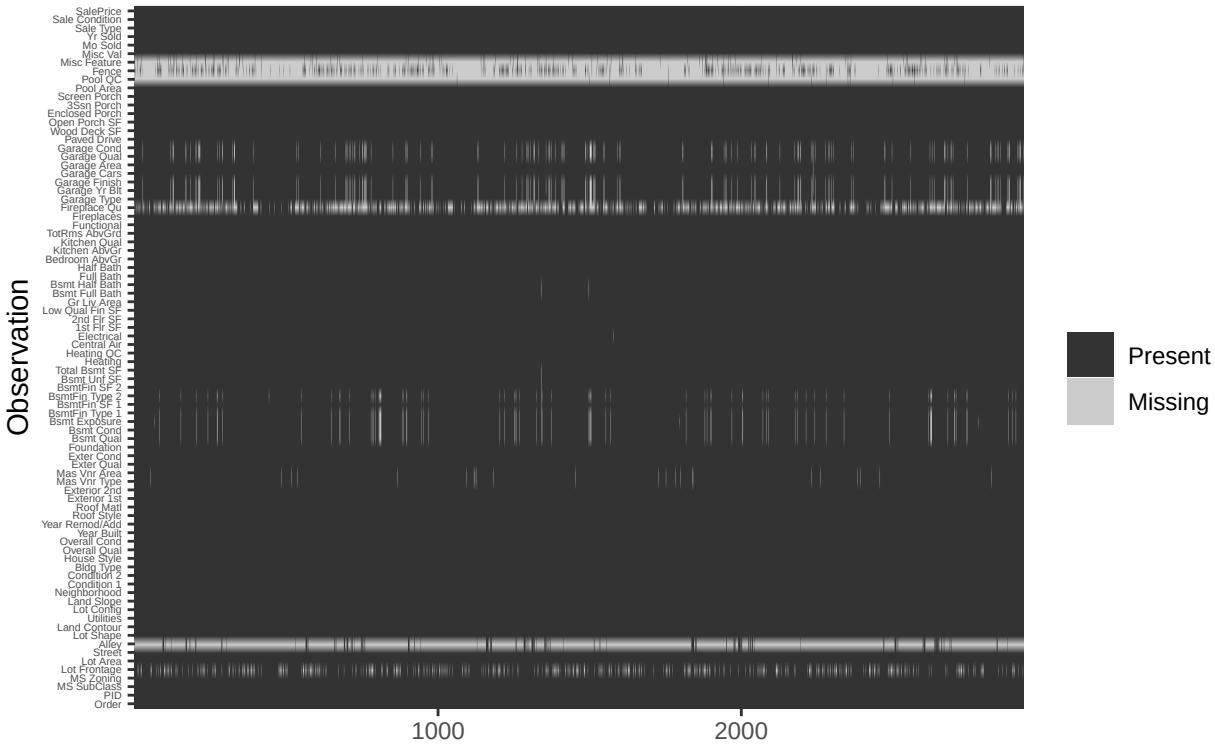

Figure 3: Distribution of missing values in raw Ames data