Chapter 2 - the modelling process

Prerequisites

Note that `h20.init()` gives a warning about the version being old, but can be ignored.

```
# Helper packages
library(dplyr)     # for data manipulation
library(ggplot2)   # for awesome graphics

# Modeling process packages
library(rsample)   # for resampling procedures
library(caret)     # for resampling and model training
suppressWarnings(suppressMessages(library( h2o )))
```

```
# just print
# h2o set-up
h2o.no_progress()  # turn off h2o progress bars
h2o.init()        # launch h2o
```

Load the data and convert it into an h2o object. Why? Well apparently h2o objects are what ML peeps use instead of data.frames. What is h2o? Apparently an open source platform for ML (or AI as they call it), that also of course interfaces with R, python etc. We will use it later on. But I'm still not clear on what the objects asre

```
# load ames housing data
ames <- AmesHousing::make_ames()
ames.h2o <- as.h2o(ames)

# Job attrition data
# load and change ordered factors into regular factors
churn <- rsample::attrition %>%
  mutate_if(is.ordered, .funs = factor, ordered = FALSE)

churn.h2o <- as.h2o(churn)
```

Data splitting

Goal: find $f(X)$ that best predicts $\hat{Y}$ based on some X. You want the alg to generalise to X values not seen already. So we split our dataset into

- **Training data** – used to develop feature sets, train the alg, tune the hyperparameters
- **Testing data** – used to estimate the unbiased estimate of our model's performance - the *generalization error*.

Too much data in the training set (>80%) leads to overfitting and we don't get a good assesment of predictive performace.

Too much on testing (>40%) can also mean we don't get good estimates of model parameters.

Also in very large data sets increasing the training set may lead to only marginal gains, but take a lot longer, so unnecessary.

We split the data usually using either *simple random sampling* or *stratified sampling*

Simple random sampling

Simple random sampling ignores the distribution of features, including the outcome variable.

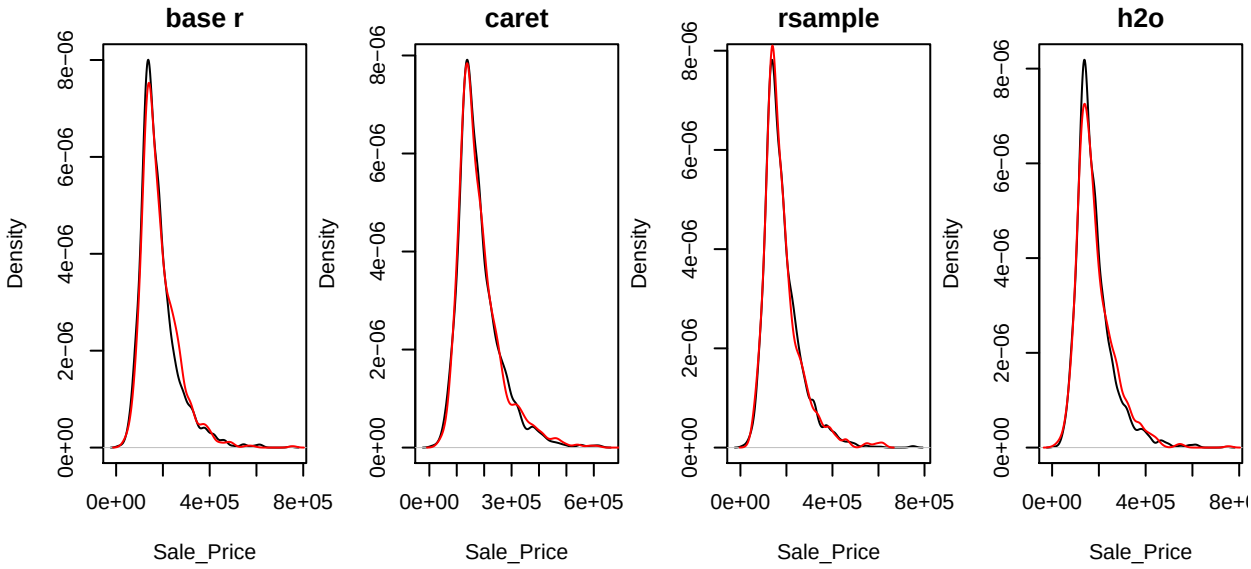

Figure 1: Distribution of target variable in training (black) and testing (red) sets

```r
set.seed(123)
# Using base R
index_1 <- sample(1:nrow(ames), round(nrow(ames) * 0.7))
train_1 <- ames[index_1, ]
test_1  <- ames[-index_1, ]

# Using caret package
set.seed(123)
index_2 <- createDataPartition(ames$Sale_Price, p = 0.7,
                               list = FALSE)
train_2 <- ames[index_2, ]
test_2  <- ames[-index_2, ]

# Using rsample package
set.seed(123)  # for reproducibility
split_1  <- initial_split(ames, prop = 0.7)
train_3  <- training(split_1)
test_3   <- testing(split_1)

# Using h2o package
split_2 <- h2o.splitFrame(ames.h2o, ratios = 0.7,
                          seed = 123)
train_4 <- split_2[[1]]
test_4  <- split_2[[2]]
```

If the dataset is large enough it is usually sufficient to get a similar distribution of the target variable Y in the training set and the test set.

Stratified sampling

To make sure both training and testing sets have similar distributions of Y, we can use stratified sampling.

You'd usually do this in classification problems, especially ones wih an unbalanced response variable to make

sure you have enough of the small category in both sets. But it's also useful for continuous response variables with small samples and a highly non-normal distribution. You do this by splitting the response into quantiles and sampling from each to ensure a balanced representation in both sets.

Here we use the `rsample` package to stratify the response `Attrition` using the arugment `strata`.

```
# orginal response distribution
table(churn$Attrition) %>% prop.table()
```

```
##
##        No       Yes
## 0.8387755 0.1612245
```

```
# stratified sampling with the rsample package
set.seed(123)
split_strat  <- initial_split(churn, prop = 0.7,
                              strata = "Attrition")
train_strat  <- training(split_strat)
test_strat   <- testing(split_strat)

# consistent response ratio between train & test
rbind(table(train_strat$Attrition) %>% prop.table(),

table(test_strat$Attrition) %>% prop.table())
```

```
##             No       Yes
## [1,] 0.8388350 0.1611650
## [2,] 0.8386364 0.1613636
```

Class imbalances

In classificaiton problems with severely imbalanced outcome variables ensuring that this is remedied is done using *up-sampling* or *down-sampling*.

Down-sampling: if you have a lot of data you can reduce the prevalance of the more abundant class by keeping the less abundant class and sampling fewer cases from the more abundant class. That way you get a similar number of cases from both. This also reduces the size of the testing and training sets, which speeds up the calculations.

Up-sampling: if the data is insufficient then you keep all of the abundant class data and use repeated sampling or bootstrapping** to increase the size of the less abundant class.

SMOTE Synthetic Minority Over-sampling technique is a combination of under and over sampling, which is often also successful.

Many ML algorithms (e.g. most h2o ones) have internal class weighting schemes to remedy these imbalances as well (`weight_columns` and `balance_classes` arguments).

Creating models in R

There are dozens of implementations of ML algs in R, they can vary by efficiency, parameter tuning options etc. This gives you a lot of options, but there are also inconsistenices in how you define the formulas or how the outputs are supplied (but not the results i hope?).

Many formula interfaces

Traditional R style formula interfaces

```
# Sale price as function of neighborhood and year sold
Sale_Price ~ Neighborhood + Year_Sold,

# Variables + interactions
Sale_Price ~ Neighborhood + Year_Sold + Neighborhood:Year_Sold

# Shorthand for all predictors
Sale_Price ~ .

# Inline functions / transformations (ns = natural spline)
log10(Sale_Price) ~ ns(Longitude, df = 3) + ns(Latitude, df = 3)
```

But there are disadvantages to this interface: * You cannot nest inline functions like pca() * the matrix calculations happen at once and cannot be recycled * can be inefficient for wide datasets * can be inelegant for large multivariate models * is inconsistent, since not all modelling functions have a formula mehtod

Some modelling functions have a non-formula interface:

```
# Use separate inputs for X and Y
features <- c("Year_Sold", "Longitude", "Latitude")
model_fn(x = ames[, features], y = ames$Sale_Price)
```

This can be more efficient, but also inconvenient if you have transformations of features, or they are factors, interactions etc..

Not sure what the difference is here really, but `h2o` uses a *variable name specification* like so:

```
model_fn(
  x = c("Year_Sold", "Longitude", "Latitude"),
  y = "Sale_Price",
  data = ames.h2o)
```

Many engines

Meta engines help provide consistency. There are many packages that can run the same model (different implementation). A meta engive kind of funnels it through it's own engine, as an aggregator to get consistent outputs.

For example the following all produces the same model, although the outputs are not structured the same way:

```
lm_lm    <- lm(Sale_Price ~ ., data = ames)
lm_glm   <- glm(Sale_Price ~ ., data = ames,
                family = gaussian)
lm_caret <- train(Sale_Price ~ ., data = ames,
                  method = "lm")
```

`lm` and `glm` are two different engines. `caret` allows you to use almost anything as the `mehtod`. Using engines directly is more flexible, but you need to learn the specifics of each implementationand syntax. Meta engines make it easier to extract outputs using consistent syntax. `caret` is a popular meta engine, `parsnip` is a new one as well.

I think `parsnip` is simply a `tidyverse` interface for `caret`? And yes, `parsnip` is part of `tidymodels`, which includes `rsample` as well, and is a tidy interface, written by the same guy, Max Kuhn, of `caret` fame. See here for a great write up and comparison in practice.

Resampling methods

Assessing model performance: where? Not on the training set obviously. And apparently assessing it on the test data is not a great idea, since it can be biased and lead to the selection of a model that does not have great generalizability.

An alternative is to use a **validation set**, which is a third part of the data, also sometimes called the hold-out set. Train on training, estimate performace on the validation set and then test the chosen model on the test set. Of course if you don't have a lot of data this is also problematic.

A third option is **resampling methods** which allow us to repeatedly fit a mode on parts of the training set. The most popular are *k-fold cross* and *bootstrapping*.

k-fold cross validation

You split your *training* set into k folds. Then train the model on $k-1$ folds and test it on the k^{th} fold to get the error. Repeat k times and take the average. You still have the testing set left, on which to test the final model.

Setting $k = 10$ is pretty good, performs close to LOOCV - leave one out CV, which is the most extreme k-fold version, where $k = n$.

Instead of increasing k it's better to repeat k-fold CV many times, e.g. k times. Also if you repeat it, you not only get better accuracy, but also an estimate of its variability.

You can often perform CV directly from the ML functions.

Alternatively you can perform CV externally e.g. with `rsmaple::vfold_cv()`

Bootstrapping

Bootstrapping is random sampling with replacement. A bootstrap sample is the same size as the original dataset, and contains approximately the same distribution as the original. Because of replacement, there are many duplicates. On average only about 63% of the original ends up in the bootstrapped sample. The others are considerred *out-of-bag* (OOB).

So you can train the model on the bootstrap and test it on the OOB set, which is often done e.g. in random forest. (why only there?).

Because of duplication there tends to be less variability in bootstrapped error estimates complared to k-fold CV and this can increase its bias. But apparently for datasets > 1000 this doesn't matter much.

Implementation is easy externally with `rsample::bootstraps()`. But it is typically more of an internal procedure built into many algorithms already.

Alternatives

There are specific approaches especially for time series data.

Also the 632 method, aimed at minimising biases experienced by bootstrapping in small datasets. with a great explanation here.

Bias-variance trade-off

Prediciton errors can be decomposed into bias and variance subcomponents. There is often a trade-off between a model's ability to reduce one or the other.

Bias

Bias is the difference between the expected and prediction and the correct value. So in general (on average), how far are the model's predicitons from the true values.

If you have high bias, then even the noise introduced by resampling will not change that. It will remain consistent.

Variance is the variability of a model prediction for a given data point. Very adaptable models can fit many different patterns, but this leads to overfitting on the training data, which may not generalize to the test data.

If you have a high variance model then adding noisy resampling is crucial to reduce the risk of overfitting.

The tradeoff is in complex models with many hyperparameters that need to be tuned. (?)

Hyperparameter tuning

Hyperparameters control the complexity of the model and the bias-variance tradeoff. Not all models have them (e.g. OLS doesn't), but most have at least one.

How you set them depends on the data and the problem *but it isn't always possible to set them from the training data alone*. So how do you decide the optimal values for the parameters? E.g. a k-nearest neighbours model has only one parameter, k which defines how many similar observations are averaged to get the prediciton. If k is small, the models have high variance (but low bias). Increasing k reduces the variance, but increases the bias. So which k is right?

You can fiddle with k and measure how it affects the predictive accuracy e.g. with k-fold CV. This might be sensible if you have only a few hyperparameters, but what if you have loads? Then you can perform a **grid search**, which is an automated way of searching through the hyperparameter space. In the k-nearest neightbours example, with only one parameter, you can perform a *full Ccartesian gird search*, because you can try all possibilities within a reasonable range.

But with more hyperparameters this is too computationally expensive, so you need to adjust the *grid settings* to search more efficiently. You can also perform *random grid searches*, *early stopping*, where you stop searching once the improvement stops being important, *adaptive resampling via futility analysis* etc.

Model evaluation

Model evaluation used to be performed using some sort of goodness-of-fit test and the analysis of residuals. But apparenlty this can lead to misleading models even if they pass these tests, so nowadays it is more common to use a **loss function** to assess performance. *Loss functions* are metrics that compare the predicted values to the actual value. The output is called the *error* or the *pseudo residual*.

(I don't get this, how is this different?)

So there are different loss functions, each with a different definition of the predictive accuracy, most obviously these differ for regression and classification problems. Different loss funcitons will emphasise different types of errors, so they can lead to drastically different models being chosen as optimal. So the context of the modelling is very important in determening the chosen metric. Obviously only compare models using the same loss function.

Regression model loss functions

MSE mean squared error $MSE = \frac{1}{n}\sum_{n_i}^{1} = 1(y_i - \hat{y}_i)^2$. Average of the squared errors, objective is to minimise. Similar to the error in OLS, but here we divide with n instead of $n - p$ to adjust for bias. SUper popular

RMSE root mean squared error: $RMSE = \sqrt{\sum_{n_i}^{1} = 1(y_i - \hat{y}_i)^2}$, same as before, but square root of the squared error, this way the error has the same unit as the response variable (not the square). Objective is to minimise.

Deviance - mean residual deviance. Like MSE, but only take the absolute difference. This gives all errors equal weight, not more to bigger errors like MSE $MAE = \frac{1}{n}\sum_{n_i}^{1} |y_i - y_i|$. Also minimise.

RMSLE root mean squared logarithmic error. If your response has a large range of values, large values will dominate MSE or RMSE. RMSLE minimises this impact so that small values with large errors have just as much impact as large values with large errors. Minimise.

R^2 Proportion of variance of the response variable that is predictable from the independent variables. But major limitation is comparability: two models from two datasets can have the same RMSE, but if one has a response with a lot less variability, it will also have a smaller R^2. So don't use it. objective is to maximise.

Classification model loss functions

Missclassification Percentage of observations classed in the wrong class. Objective to minimise.

mean per class error - average error rate for each class. If the classes are balanced, this is the same as missclassification. Otherwise the small class error rates have more weight and large ones less.

MSE - works because the preditctions are actually probabilities. Take the difference from 1 to the predicted probability for that class and square it. Large differences get amplified.

cross entropy a.k.a. **log loss** or **deviance**. similar to MSE, but incorporates the log of the predicted probability multiplied by the true class. This disproportionally punishes low probability predicitons for the true class.

Gini index commonly used in tree-based methods, also referred to as **purity**, where a small value means a node contains mainly observations from the same class.

For classificaiton models we often look at the **confusion matrix** to see the true and false positives/negatives. If we have a binary outcome variable there are a few other metrics we can use:

Accuracy (oposite of misclassification): how often is the classifier correct? True positives + true negatives / total.

Precision: maximising the true positive vs false positive ratio: TP/(TP+FP). How accurately does the classifyer predict events.

Sensitivity a.k.a. **recall**: how accurately does the classifier classify actual events. TP/(TP+FN).

Specificity: how accurately does the classifier predict non-events: TN/(TN+FP)

AUC - area under the curve. A good binary classifier has high precision and high sensitivity: so it's good at predicting an event will occur and that it won't. So it minimises false positives and false negatives. We use an ROC curve (*reciever operating charactheristic*), which plots the false positive rate on x and the false negative rate on y. The diagonal is random guessing, so the more the curve diverges to the top left, the better. The area under the curve needs to be maximised. (Hmm, i'll need a more worked out example)

Putting the processes together.

data splitting

Use `caret::initial_split()` to split the training and testing sets, and stratify it by the outcome variable to ensure it is consistently distributed between both.

```
# 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)
```

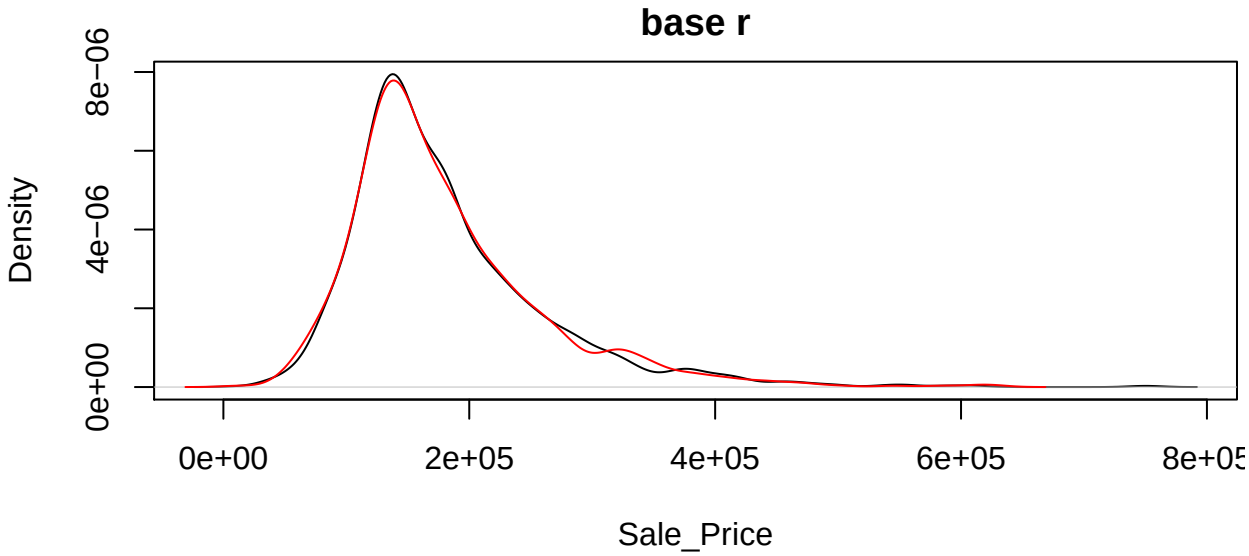

Figure 2: Distribution of target variable in training (black) and testing (red) sets

Next we will use *k-nearest neighbours* to model the price. The `caret::train()` function will allow us to at the same time also

- resample: using 10-fold crossvalidation repeated 5 times
- do a grid search for the optimal k from specified ranges
- train and validate the model using our preferred loss function (RMSE)

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

# Create grid of hyperparameter values
# column name is same as tuning parameter name = k
hyper_grid <- expand.grid(k = seq(2, 10, by = 1))

# Tune a knn model using grid search
knn_fit <- train(
  Sale_Price ~ .,  # outcome and independent variables defined with formula
  data = ames_train,
  method = "knn", # n-nearest neighbours
  trControl = cv, # resampling
  tuneGrid = hyper_grid, # possible tuning values
  metric = "RMSE"
)
```

So for each value of k (here from 2 to 10 to shorten the calculation), `train` performs five repeats of a 10-fold cross validation (i.e. train on 9/10 of the training data and test on the remaining 1/10), calculates the RMSE for each and takes the average of the 50 values for that k. Then compares all the ks and figures out which is the best i.e. has the smallest RMSE.

```
## k-Nearest Neighbors
##
```

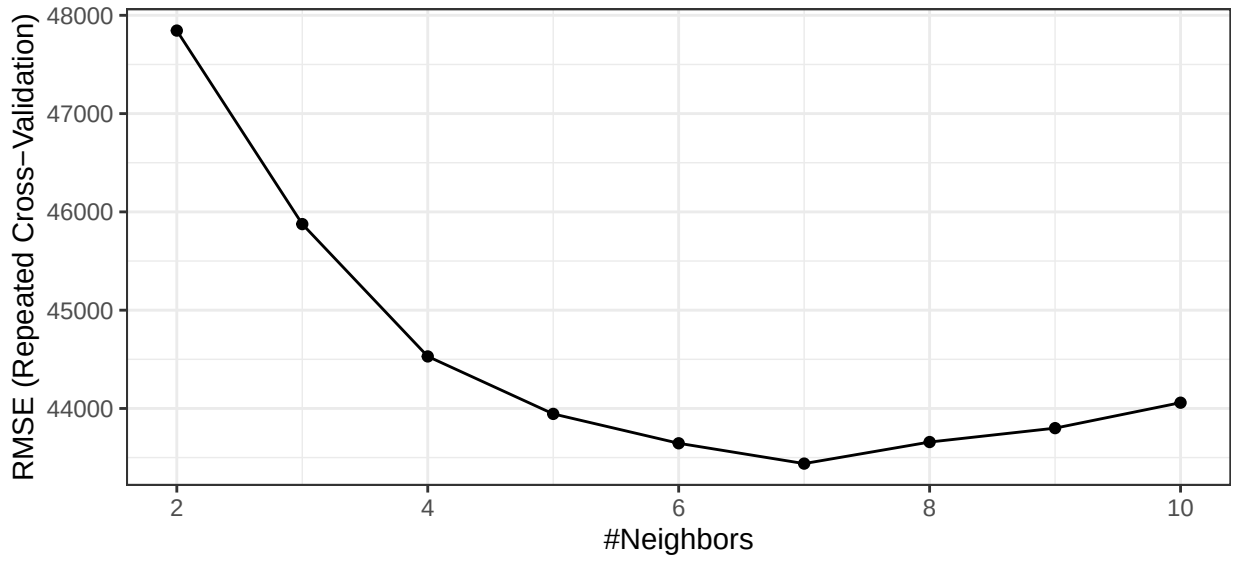

Figure 3: Plot of tuning parameters and metric for trained model: k and RMSE

```
## 2053 samples
##   80 predictor
##
## No pre-processing
## Resampling: Cross-Validated (10 fold, repeated 5 times)
## Summary of sample sizes: 1848, 1848, 1848, 1847, 1849, 1847, ...
## Resampling results across tuning parameters:
##
##   k   RMSE      Rsquared   MAE
##    2  47844.53  0.6538046  31002.72
##    3  45875.79  0.6769848  29784.69
##    4  44529.50  0.6949240  28992.48
##    5  43944.65  0.7026947  28738.66
##    6  43645.76  0.7079683  28553.50
##    7  43439.07  0.7129916  28617.80
##    8  43658.35  0.7123254  28769.16
##    9  43799.74  0.7128924  28905.50
##   10  44058.76  0.7108900  29061.68
##
## RMSE was used to select the optimal model using the smallest value.
## The final value used for the model was k = 7.
```