

Lecture 4: Training and evaluating models with package caret

Isabel Casas
icasas@deusto.es

- So you've got a clean MPG dataset, picked the statistical model you're going to use, trained your model and got some results. They may even look pretty good!
- But how do you know if your model is the best that it can be? One way to improve a model is through **hyperparameter** tuning.

Definition: Parameter vs hyperparameter

Parameter: What the algorithm or model is learning during training.

Example: coefficients in a regression.

Hyperparameters: We set them in the model. Example: the number of trees in a random forest, the number of variables in each of these trees.

- The importance of hyperparameters in building robust models will specifically depend on the type of model you're training.

Hyperparameters

- We can try to pick the best values for our hyperparameters by hand, fitting several models with different settings and choosing the combination with the smallest error (**Remember how we chose the number of trees in the last lab**).
- Luckily for us, R has the *caret* package, designed to optimize hyperparameter choices quickly and systematically for many different models.
- In this class, we will learn how to use the *caret* package to select the best-performing hyperparameters for a specific model.

Machine learning techniques in the training phase to pick the best model:

- Bootstrap
- Holdout
- Cross-validation
- Monte Carlo

Split the dataset

- 1 Upload the data into our memory and do any cleansing necessary.
- 2 Split dataset into training and testing sets using the appropriate splitting methodology.

```
# 1. Upload data  
mpg.data <- read.table("../Lecture02/data/mpg_new.csv",  
                        sep=";", header = TRUE)
```

Holdout Methodology

- The **holdout** method consists of randomly dividing the available data into training and test samples, usually with a ratio of 70% or 80% for training and 30% or 20% for testing.
- The holdout method is often used with large datasets because there is a danger of either having a test set that is too small (unreliable prediction) or too small a training set (worse model than the one with the whole dataset) for small datasets.
- Because our *MPG* dataset is small, we will use 80% of the data for training and the rest for testing.
- The ratio could be different if we had a very large dataset. Perhaps the opposite.
- Commonly, we get more accurate models the bigger the dataset we're training on, but more training data also leads to models taking longer to train and the issue of overfitting.

Testing vs Predicting samples

- The testing sample (or validation sample) is not a prediction sample.
- The training sample is divided in two parts, as we mentioned before, the first part for training and the second for testing.
- Knowing which model predicts best in both phases, we could use it for prediction in another related dataset.

Holdout data split in R

Package *caret* has the function *createDataPartition()* that randomly samples the proportion of the indexes of a vector you pass on.

```
library(caret)
set.seed(1234)
# 2. Splitting into training and test samples
training_indexes <- createDataPartition(mpg.data$mpg, p = 0.8, list = FALSE)
training <- mpg.data[training_indexes, ]
testing <- mpg.data[-training_indexes, ]
training_indexes[1:10]
```

```
## [1] 2 3 4 5 6 7 8 9 10 11
```

The number of rows of *mpg.data* is 398, we use 321 rows for *training* and 77 rows for *testing*.

Cross-validation Methodology

- Measuring the estimation MAE or RMSE is a rough way of measuring model fitting accuracy, and it will depend heavily on the quality of the training sample.
- To avoid this caveat, we can use the k -Fold cross-validation (CV).
- When $k = 1$, it is denoted by leave-one-out cross-validation (LOOCV). LOOCV is one of the most common methods to evaluate the performance of a model. It consists of repeating k times a train/test cycle, but where the test sample is carefully chosen instead of randomly selected as in k repetitions of random subsampling.

Cross-validation Methodology

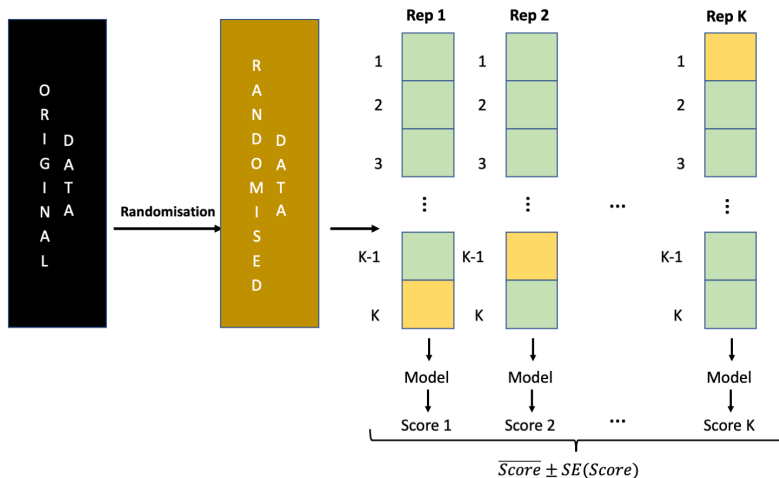


Figure 1: Steps in k-Fold cross-validation.

Figure 1 shows graphically the CV steps:

- 1 Randomly re-shuffling the original data
- 2 Split the dataset into k equal-sized partitions of which we pick one partition for testing (gold coloured) and $k - 1$ for training (green colour). This is repeated B times. The final model score will be the average of every repetition score.

Remark 1. k -Fold CV may take a good while for large datasets.

Cross-validation in R

- We can use *caret* to split the data with k-Fold CV by using the *trainControl()* function.
- Below, you can see an example of a 5-Fold CV with **training**. CV gives us the opportunity to use several subsamples of *training*, in fact k subsamples, in the hope to get trained model.

Cross-validation in R

The chunk below:

- 1 Performs 5-fold cv in the training set 20 times, i.e., there are 100 model evaluations
- 2 Train and tune three models (regression, tree and RF)

```
ctrl <- caret::trainControl(method = "repeatedcv", number = 5,  
                             repeats = 20, savePredictions = "final")  
model_cv <- caret::train (mpg ~ . , data = training,  
                           methodList=c("lm", "rpart", "rf"),  
                           trControl = ctrl, metric = "RMSE")  
model_cv
```

Cross-validation in R

Output from previous slide. It choose RF automatically and with different values of the hyperparameter *mtry*. From the RMSE, it chooses $mtry = 4$

```
## Random Forest
##
## 321 samples
## 7 predictor
##
## No pre-processing
## Resampling: Cross-Validated (5 fold, repeated 20 times)
## Summary of sample sizes: 256, 258, 258, 256, 256, 257, ...
## Resampling results across tuning parameters:
##
##  mtry  RMSE      Rsquared    MAE
##  2     2.912923  0.8559528  2.003561
##  4     2.894200  0.8569121  1.985620
##  7     2.925685  0.8533429  1.988120
##
## RMSE was used to select the optimal model using the smallest value.
## The final value used for the model was mtry = 4.
```

Training an untuned model

The *caret* package allows us currently to fit 238 types of models using its function *train*.

The *train()* function can be used to:

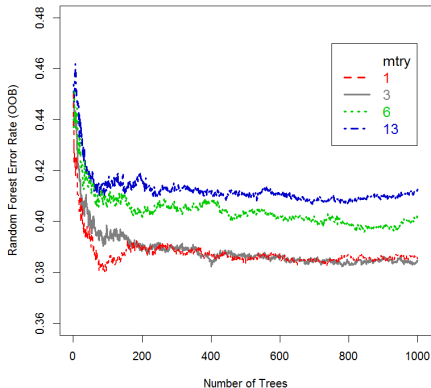
- evaluate and tune the parameters of a model using resampling
- choose the “optimal” model across several hyperparameters
- estimate model performance from a training set

In our regression problem (predicting *mpg*), we use the randomforest, which has two hyperparameters: *ntree* and *mtry*.

- *ntree*: This is the total number of trees in your final ensemble model.
- *mtry*: The number of variables (features, predictors) to use in each tree of the random forest.

Training an untuned model

- *caret* chooses the hyperparameter value that produces the lowest overall error for a given model.
- Figure 2 below, which is the Fig A4 of Buskirk & Kolenikov (2015)



- We saw the *Bagging* methodology in Lab3 and used it to understand how the randomforest works.
- We find in 2 (previous slide) the term OOB in the y-axis.
 - ▶ What does it stand for?
 - ▶ Explain how OOB is used to tune models that have been created with *Bagging*

Tuning `ntree` in a RF

- In Figure 2, the error drops off sharply near the beginning but continues gently downward.
- Increasing the number of trees increases the time it takes to train our model and makes it more likely that we will overfit the model to our data.
- Therefore, picking `ntree` by the smallest overall error doesn't make sense. (It would probably recommend an infinite number of trees!)
- Instead, we want to pick a value near the “elbow”, where we have high accuracy but aren't training more trees than necessary. In practice, most random forest models will perform well with a number of trees somewhere between 50 and 500.

Tuning **mtry** in a RF

- On the other hand, changing the *mtry* parameter will usually lead to a U-shaped error pattern: both very high and very low values will have higher error rates and somewhere in the middle will be a sweet spot where the error rate is lower. This is the type of parameter that it makes sense to tune using *caret*.
- How will we know whether my parameter will respond well to tuning? The *caret* authors (Max Kuhn and the rest of the contributors) have your back. For most models, *caret* will already know which parameters make sense to tune and ignore the others.
- For example, we can't actually tune *ntree* using *caret* and it will automatically tune *mtry* for you. Handy!

Training a random forest in R

- Let's train a basic randomforest model with the default value of *mtry* ($p/3$, for p number of predictors).

```
# train a random forest model
library(randomForest)
rf.model <- randomForest(mpg ~ ., data = training, ntree = 50)
# check out the details
rf.model
```

```
##
## Call:
## randomForest(formula = mpg ~ ., data = training, ntree = 50)
##           Type of random forest: regression
##           Number of trees: 50
## No. of variables tried at each split: 2
##
##           Mean of squared residuals: 9.10753
##           % Var explained: 84.32
```

- The output tells us about the model's qualities and how good it was at capturing the variation in our training data.

Output of *train()* for RF

- Call: It describes what we passed into the *randomForest* function in the first place.
- Type of random forest: This is automatically determined based on your target variable *mpg*. Since our target variable is not a factor, we've trained a regression tree.
- Number of trees: We specified this manually (*ntree* = 50).
- No. of variables tried at each split, by default $mtry = 7/3 = 2.3 = 3$.
- Mean squared residuals (MSE) - fitness error measure
- % variance explained: It is R-squared value $\times 100$ for this model (100 means that our model perfectly fits our data)
 - ▶ Too small values of MSE or values of % Var too close to 100 -> overfitting
 - ▶ Too large values of MSE or value of %Var too close to 0 -> bad model for this dataset

Prediction

- Calculate the untuned model error on testing data using the root mean squared error (RMSE). Besides the formula we saw last week, we can use the `rmse()` function from package *ModelMetrics* to calculate it.
- **Tip:** Because it scales based on the number of points in your dataset, it doesn't usually make sense to compare RMSE for models trained on different datasets.

```
#prediction errors
library(ModelMetrics)
rf.model.pred <- predict(rf.model, testing)
rmse(rf.model.pred, testing$mpg)
```

```
## [1] 2.733357
```

```
mse(rf.model.pred, testing$mpg)
```

```
## [1] 7.471238
```

```
mae(rf.model.pred, testing$mpg)
```

```
## [1] 2.125199
```

- Those numbers by themselves do not mean much. But now that we have a base error, let's see if we can train a more accurate model by tuning `mtry`!

Tuning model with *caret*

Tuning our model is pretty simple, it will just take awhile: It trains a bunch of models with different values of hyparameter(s).

```
# use caret to pick a value for mtry
rf.tuned_model <- caret::train(mpg ~ ., data = training, ntree = 50,
                              tuneGrid = data.frame(mtry = 1:11), method = "rf")
print(rf.tuned_model)
```

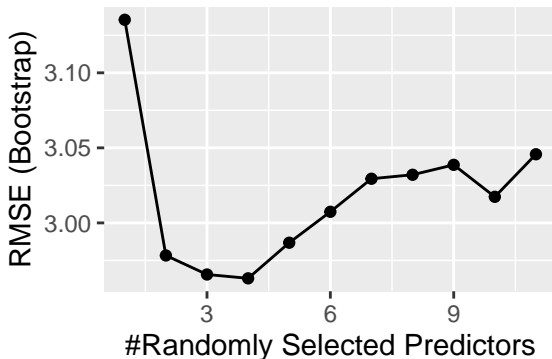
```
## Random Forest
##
## 321 samples
## 7 predictor
##
## No pre-processing
## Resampling: Bootstrapped (25 reps)
## Summary of sample sizes: 321, 321, 321, 321, 321, 321, ...
## Resampling results across tuning parameters:
##
##  mtry  RMSE      Rsquared  MAE
##  1     3.135334  0.8347109  2.166495
##  2     2.978312  0.8461574  2.031661
##  3     2.965640  0.8475586  2.046576
##  4     2.963080  0.8469789  2.042085
##  5     2.986812  0.8445699  2.037476
##  6     3.007427  0.8428050  2.045458
##  7     3.029427  0.8402219  2.068865
##  8     3.032080  0.8396735  2.067650
##  9     3.038679  0.8391012  2.071822
## 10     3.017386  0.8413267  2.057702
## 11     3.045744  0.8381904  2.071641
##
## RMSE was used to select the optimal model using the smallest value.
## The final value used for the model was mtry = 4.
```


Tuning model with *caret*

- The *train()* function picked *mtry=4* as the best choice.
- The rf for this dataset is most accurate when it uses 4 variable in each stump, although the theoretical value is *mtry = 2*
- We can see this clearly in the plot of the rmse for each *mtry* value, passing our tuned model to ggplot.

Tuning model with *caret*

```
library(ggplot2)
ggplot(rf.tuned_model)
```



- RMSE has an U shape with its lowest value for $mtry = 4$.

Activity

- Create an R script with the code above, using the package *caret*
- Run it and understand the output and what each function does.

Prediction

```
# Estimate and predict using the best model  
names(rf.tuned_model)
```

```
## [1] "method"      "modelInfo"    "modelType"    "results"      "pred"  
## [6] "bestTune"     "call"         "dots"         "metric"       "control"  
## [11] "finalModel"   "preProcess"   "trainingData" "ptype"        "resample"  
## [16] "resampledCM" "perfNames"    "maximize"     "yLimits"      "times"  
## [21] "levels"       "terms"        "coefnames"    "xlevels"  
rf.tuned_model.est <- predict(rf.tuned_model$finalModel)  
rf.tuned_model.pred <- predict(rf.tuned_model$finalModel, newdata = testing)
```

Prediction performance

- Prediction performance is very important if prediction is the main objective
- Once we have trained a model, we will use it to predict future data outputs.
- There are different ways of calculating a model prediction performance

Pred. performance: Simple accuracy measure

One simple way is to use an error function like the RMSE or MAE and compare the performance of each model in the training and testing samples.

```
print("base model estimation rmse")
```

```
## [1] "base model estimation rmse"  
print(rmse(predict(rf.model), training$mpg))
```

```
## [1] 3.017869  
print("tuned model estimation rmse:")
```

```
## [1] "tuned model estimation rmse:"  
print(rmse(rf.tuned_model.est, training$mpg))
```

```
## [1] 2.926877  
print("base model prediction rmse:")
```

```
## [1] "base model prediction rmse:"  
print(rmse(rf.model.pred, testing$mpg))
```

```
## [1] 2.733357  
print("tuned model prediction rmse:")
```

```
## [1] "tuned model prediction rmse:"  
print(rmse(rf.tuned_model.pred, testing$mpg))
```

```
## [1] 2.358876
```

This lecture is based on:

- 1 The tutorial [Picking the Best Model with Caret](#) by Rachael Tatman