R Tutorial at the WZB

12 – Introduction to Machine Learning with R I

Markus Konrad

January 31, 2019

Today's schedule

1. Key concepts and terminology
2. General workflow for creating predictive models
3. Practical Machine Learning with `caret`

WZB
Wissenschaftszentrum Berlin für Sozialforschung

Review of last week's tasks

Solution for tasks #11

now online on
https://wzbsocialsciencecenter.github.io/wzb_r_tutorial/

Key concepts and terminology

Machine Learning

source: xkcd.com/1838/

Supervised and unsupervised ML

Machine learning (ML) in general

- algorithms that build mathematical models which allow making predictions/decisions without being explicitly programmed on the task - models are "learnt" from sample data and predict outcomes from unseen data

supervised ML

- labelled data with explanatory variables and outcomes
- examples: classification, regression
  - spam / not spam based on word occurrences
  - hotdog / not hotdog based on features in image pixel data
  - annual income based on individuals working experience, education, etc.

unsupervised ML

- no labelled data; takes the data as is and tries to uncover key structure
- → dimension reduction / simplification
- examples: clustering, principle component analysis, topic modeling

History

Although only popular in the media since about a decade, the term Machine Learning is already around since the late 50s, when "Artificial Intelligence" had had it's first heyday. Most methods come from the field of statistics, probability theory and computer science, some of them several decades old.

Back then, there were two problems to actually apply these methods in large scale: Too few data and too slow computers. Researchers hence focused on "expert systems" (rule-based methods).

Since the advent of the Internet and cheaper, more powerful computers, this has changed and ML is back as highly active research field and an established method applied in many industries.

Terminology and notations

Some terminology used in these slides and popular synonyms:

- response, dependent variable, class: outcome that is being predicted – notation: \(y\)
- predictors, independent variables, features: input data that is used for training or prediction – notation: \(X\)
- model training, model building, model fitting, parameter estimation: use sample of predictors to "learn" patterns (model parameters) that allow outcome prediction
- hyperparameters or tuning parameters: model parameters which cannot be learnt from the predictors; an optimal set of hyperparameters can be estimated by model tuning

Predictive models and interpretable models

Predictive models are supervised ML models primarily built to **most accurately make a prediction** from input data.

Some predictive models are more complex than "traditional" models such as Ordinary Least Squares (OLS) regression, logistic regression, etc. which can make them hard to interprete ("black box models"). This is accepted as trade-off for higher predictive accuracy.

Example:

Given a set of features about a certain neighborhood in a town, a model should **most accurately** answer the question: **Is it worth to do a campaign here in order to win voters for an election?** It would take into account known features of the neighborhood (previous election results, social structure, etc.) and campaigning costs. The interpretation of why a neighborhood is chosen for campaigning or not, is only of secondary interest (e.g. in order to understand why predictions fail).

Predictive models and interpretable models

There is always a tension between predictability and interpretability:

> While the primary interest of predictive modeling is to generate accurate predictions, a secondary interest may be to interpret the model and under- stand why it works. The unfortunate reality is that as we push towards higher accuracy, models become more complex and their interpretability becomes more difficult. This is almost always the trade-off we make when prediction accuracy is the primary goal.
>
> – Kuhn & Johnson 2016

What do you think about that? Are "black box models" a danger? Or can they be justified?

Predictive models and interpretable models

Note that there is currently a lot of research about "interpretable ML models", which tries to reduce the tension between highly accurate predictability and interpretability.

One recent approach is LIME (Ribeiro et al. 2016), which we can have a look at in the next session.

Why should I be interested?

Besides potentially higher predictive accuracy, many models have beneficial properties such as:

- effective handling of problematic predictor characteristics (skew, sparsity, many categories)
- relationship between response and predictors must not be explicitely defined
- implicit modeling of nonlinearities and interactions
- built-in feature selection (model learns "important" predictors, discards others)
- higher flexiblity due to "tuning parameters"

Which of these properties apply depends on the actual model you use.

Why should I be interested?

For some research questions, high prediction accuracy might be of interest (e.g. forecasting, classification).

Other research questions might focus on whether or how accurate a prediction can be made, rather than what influences those predictions (see Taylor Brown's project about the art market).

Predictive ML models might be used as a tool for parts of your research (e.g. using their strong predictive power for imputation or classification).

Impact on society

Even if you don't intend to use these models in your research studying them might be interesting for you, because they are widely employed in many sectors. To name a few:

- targeted advertisement (Google Ads, Amazon, Facebook, etc.)
- political campaigning (e.g. Obama campaign)
- credit scoring
- "predictive policing"
- legal judgement (!)

For a more comprehensive list and the (often unsettling) implications see Weapons of Math Destruction (O'Neil 2016).

General workflow for creating predictive models

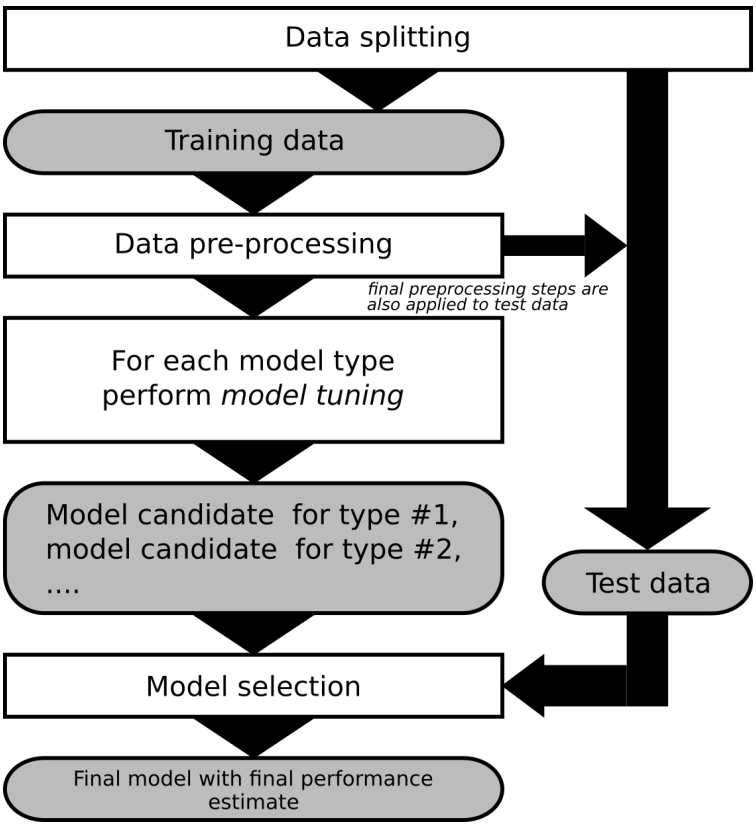
Common failures in predictive modeling

Kuhn & Johnson 2016 identify four main reasons why predictive models fail:

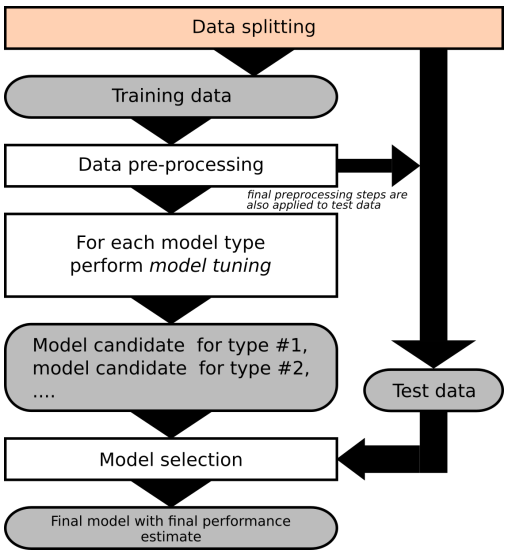
1. Inadequate pre-processing of the data
2. Inadequate model validation
3. Unjustified extrapolation
4. Overfitting the model to the existing data

They propose a workflow for building predictive models to avoid these culprits.

General workflow



Data splitting



- original data is split to produce training and test or validation data sets
- training data is used to build and validate model candidates → produces few best model candidates
- final model performance is estimated using held-out test data set → final best model is selected

→ produces honest model performance estimate because test data was not part of training process

Data splitting

There are several strategies to split the data (depends on data and aim of the model):

- simple random sampling
- stratified sampling (i.e. to make sure class proportions are similar in training and test set)
- sampling based on date (for forecasting)

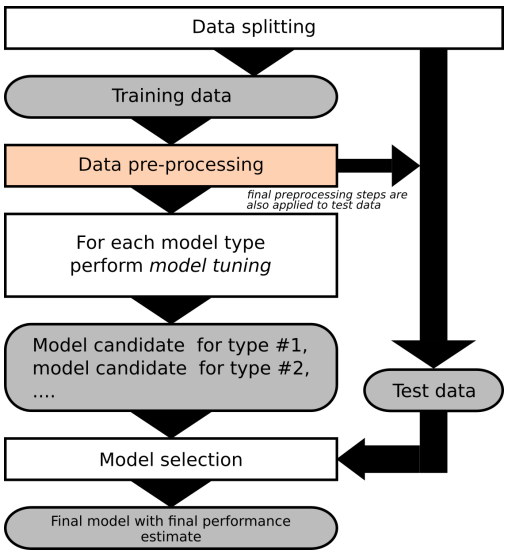
How much data is used for the test set?

→ find right balance, because:

- the larger the training set, the better the model candidates
- the larger the test set, the more robust the model performance estimates

Usually 10 to 30% of the sample is used as test data. If only small data is available, use resampling in the test data set.

Data pre-processing



Data pre-processing **transforms, adds or removes** variables in the **training set data**.[1] This process is also called **feature engineering**.

Depending on the model type, it is important which and how the predictors enter the model, e.g.:

- linear regression models are sensitive to collinearities, skew and sparsity
- tree-based models are not sensitive to skew and sparsity but may have problems with collinearities (esp. when later determining variable importance)

Note that some data transformation techniques reduce (e.g. skewness transformations, PCA) model interpretability.

[1] The final pre-processing steps must also be applied to the test data and final input data in production for proper prediction. However, keep in mind that the test data is left **untouched** until final model validation.

Data transformations

Common techniques for individual predictors include:

- centering (predictor mean becomes zero)
- scaling (predictor std. deviation becomes one)
- resolving skewness (log, inverse or other transformations)

Data reduction

Removing predictors prior modeling has several advantages:

- removing predictors that are highly correlated (collinearity) often improves model performance
- removing predictors with "degenerate distributions" (e.g. extreme skew, near-zero variance, etc.) often improves model performance
- simpler models are easier to interprete
- simpler models are faster to compute

Common techniques include:

- removal of predictors with high pair-wise correlations
- removal of predictors with near-zero variance
- Principle Component Analysis (PCA): find linear combinations of predictors that capture most possible variance

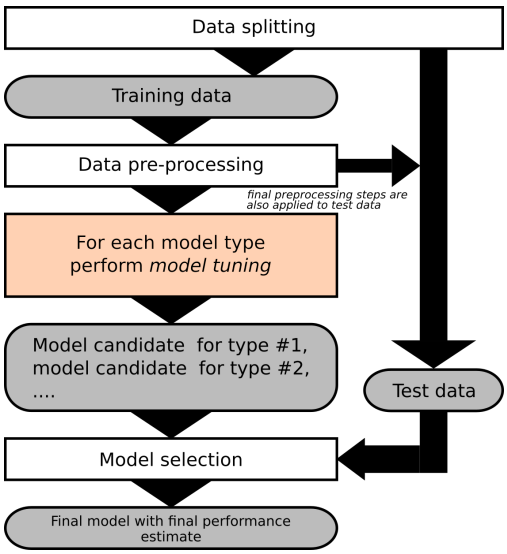
Replacing predictors

Sometimes, it is beneficial to replace one or multiple predictors, e.g.:

- create a ratio of two predictors (e.g. success rate, gender ratio)
- create dummy variables from a categorical predictor
- split date into year, month, day, weekday, etc. (e.g. to capture seasonal effects)

This may improve model performance and/or interpretability.

Model tuning and evaluation

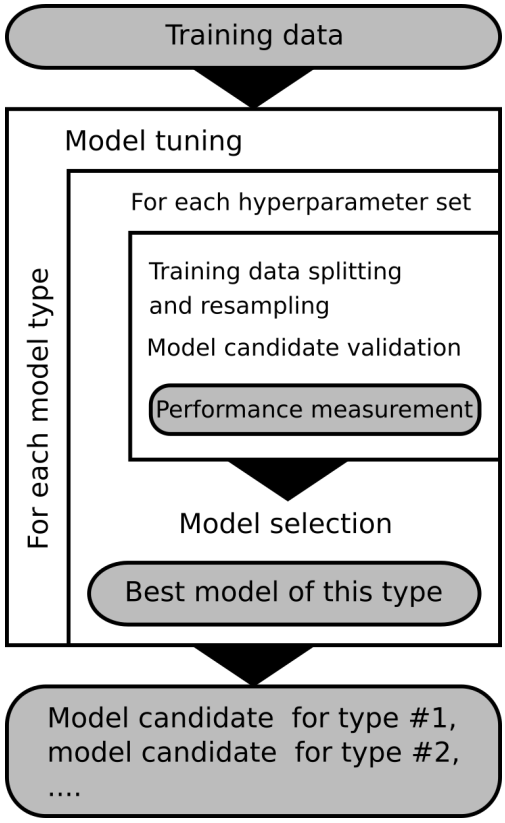


Model tuning is the process of finding optimal model parameters and hyperparameters that both fit the training data well and generalize well to unseen data.

Data splitting, resampling and performance metrics are used to evaluate the model fit and generalizability.

Model tuning and evaluation

A more detailed look at the model tuning process:



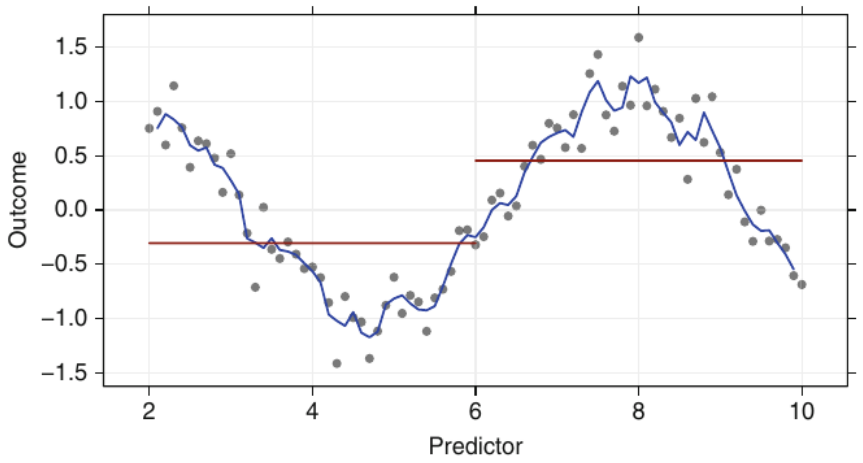
Over-fitting and bias-variance trade-off

A model that was learnt from the training data set may have the ability to perfectly "re-predict" the correct outcome.

Example: A model trained on a **labelled set of emails** could perfectly classify emails as spam / not spam if it used the **same (but unlabelled) emails** as input again.

However, this model also learnt the noise in the data set! It would perform badly on unseen data. This model is said to be **over-fitting**.

Over-fitting and bias-variance trade-off



source: Kuhn & Johnson 2016

- blue line: over-fitting model with low bias (close to the outcome in the training data) but high variance (small changes in the data have high impact on model fit)
- red line: under-fitting model with low variance (small changes in the data don't have high impact on model fit) but high bias (far from the outcome in the training data)

Hyperparameter tuning

Many models have hyperparameters. These parameters allow for greater flexibility of the model but **cannot be determined analytically.**

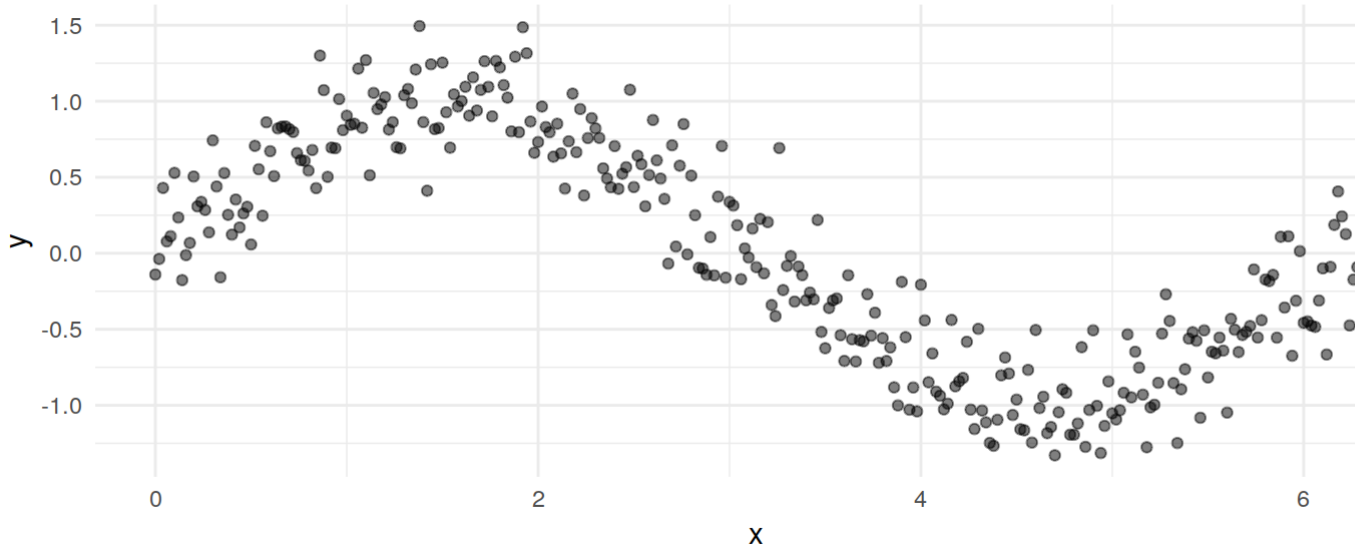
In order to find an optimal set of hyperparameters, we can create a sequence of hyperparameter candidates H. Then for each set of hyperparameters H_i:

1. Perform resampling of the training data to generate data sets for model fitting A and testing B.
2. Fit model m_{H_i} to data A.
3. Using held-out data B and model m_{H_i}, calculate predictions and performance metrics.
4. Repeat steps 1 to 3 several times and calculate a final performance estimate.
5. Model selection: Choose H_{best} as the H_i with the best performance (may take model complexity into account).
6. Fit a final model m_{final} using H_{best} and **all** the training data.

WZB
Wissenschaftszentrum Berlin für Sozialforschung

Hyperparameter tuning

Suppose we have the following data with outcome y and single predictor x:



We want to fit a LOESS model (a local regression model) to the data, which has a single hyperparameter, span. It determines the number of the neighborhood points used to calculate the local regression fit (→ the "smoothness" of the fit).

Model building in R

A model function in R usually provides at least one of the following two "interfaces" to specify a model:

The formula interface:

- model function expects two sided formula like `y ~ x1 + x2` and the data set
- allows to explicitly define the relationship between respone and predictors, including interactions (e.g. `y ~ gender:income`) and transformations
- shortcut for "use all predictors": `y ~ .`
- example: `lm(Hwt ~ Sex:Bwt, data = cats)` – linear model for cat's heart weight as function of body weight per gender (using `cats` data set from MASS)

The matrix interface:

- model function expects a matrix or data frame of predictors and a vector of the response

```
# create numeric matrix with dummy variable from data frame first:
catsX <- as.matrix(mutate(cats, female = Sex == 'F') %>% select(-Hwt, -Sex))
enet(x = catsX, y = cats$Hwt, lambda = 0.001)
```

Model building in R

An example model using the whole `cats` data:

```
library(MASS)  # for cats data
catsmodel <- lm(Hwt ~ Sex:Bwt, data = cats)
summary(catsmodel)
```

```
##
## Call:
## lm(formula = Hwt ~ Sex:Bwt, data = cats)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -3.5686 -0.9631 -0.0937  1.0423  5.1252
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  -0.3465     0.8354  -0.415    0.679
## SexF:Bwt      4.0284     0.3607  11.168   <2e-16 ***
## SexM:Bwt      4.0311     0.2853  14.129   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.458 on 141 degrees of freedom
## Multiple R-squared:  0.6466, Adjusted R-squared:  0.6416
## F-statistic:   129 on 2 and 141 DF,  p-value: < 2.2e-16
```

Making predictions using the model

Create a data frame with new data:

```
(newdata <- data.frame(Sex = c('F', 'M'), Bwt = c(2.3, 3.2)))
```

```
##   Sex Bwt
## 1   F 2.3
## 2   M 3.2
```

Predict the response (heart weight in grams) from it:

```
predict(catsmodel, newdata)
```

```
##         1         2
##  8.918796 12.553010
```

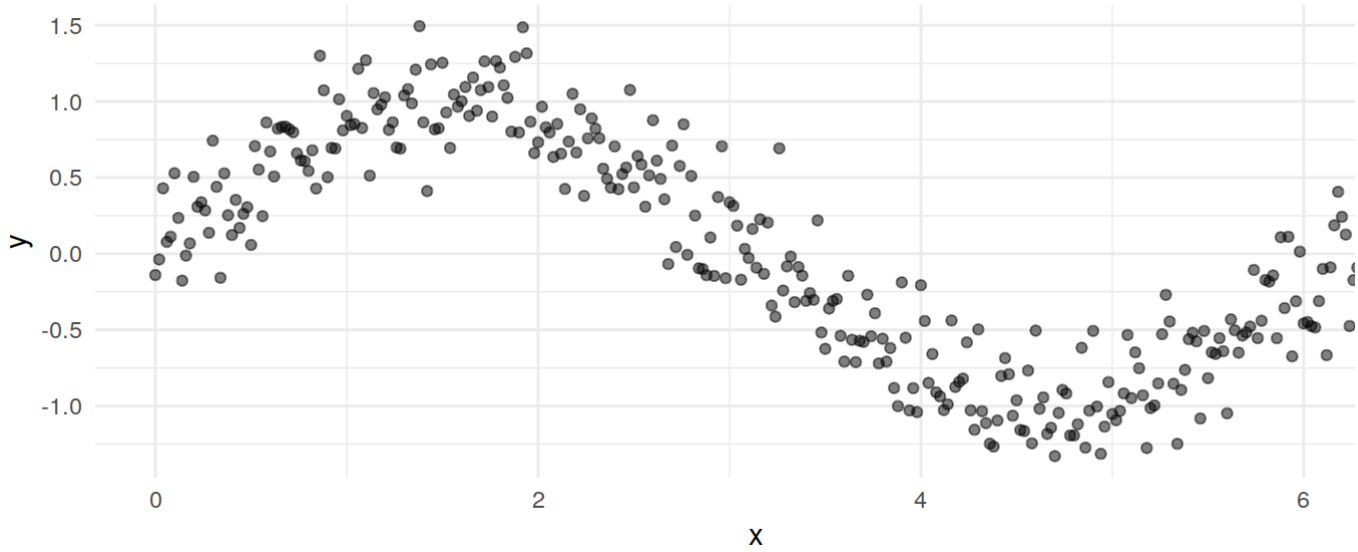
Predictions seem reasonable (of course, this is not a proper model validation):

```
group_by(cats, Sex) %>% summarize(mean(Bwt), mean(Hwt))
```

```
## # A tibble: 2 x 3
##   Sex   `mean(Bwt)` `mean(Hwt)`
##   <fct>       <dbl>       <dbl>
## 1 F            2.36        9.20
## 2 M            2.9        11.3
```

Hyperparameter tuning

Back to our simulated data:



Suppose `sine_data` was our training data. We hold out a random 25% of the data:

```
held_out_indices <- sample(1:nrow(sine_data), size = round(nrow(sine_data) *

data_held_out <- sine_data[held_out_indices,] # hold out the 25% for testing
data_train <- sine_data[-held_out_indices,]   # take the other 75% for train
```

WZB Wissenschaftszentrum Berlin für Sozialforschung

Hyperparameter tuning

We now try different values for span and fit a model to `data_train` and let it predict values from the hold-out data set. Let's start with span = 0.1:

```
loess.1_model <- loess(y ~ x, data = data_train, span = 0.1)
# # predict y from held-out x:
loess.1_pred <- predict(loess.1_model, data_held_out['x'])
```

We can calculate a performance measure (root mean squared error – RMSE) using the predicted values `loess.1_pred` and the true values from the held-out data set:

```
rmse <- function(pred, obs) { sqrt(mean((pred - obs)^2)) }
rmse(loess.1_pred, data_held_out$y)
```

```
## [1] 0.2727023
```

Hyperparameter tuning

We repeat the same as above for more hyperparameter values, here span = 0.5:

```
rmse(loess.5_pred, data_held_out$y)
```

```
## [1] 0.2630971
```

And span = 0.9:

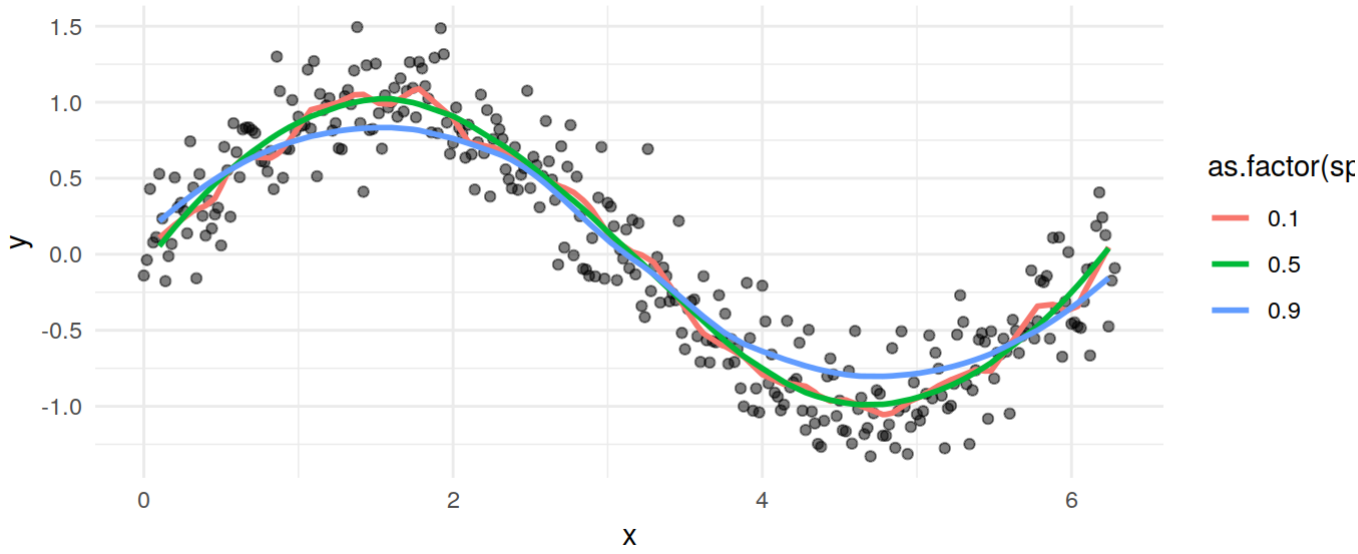
```
rmse(loess.9_pred, data_held_out$y)
```

```
## [1] 0.2755069
```

Hyperparameter tuning

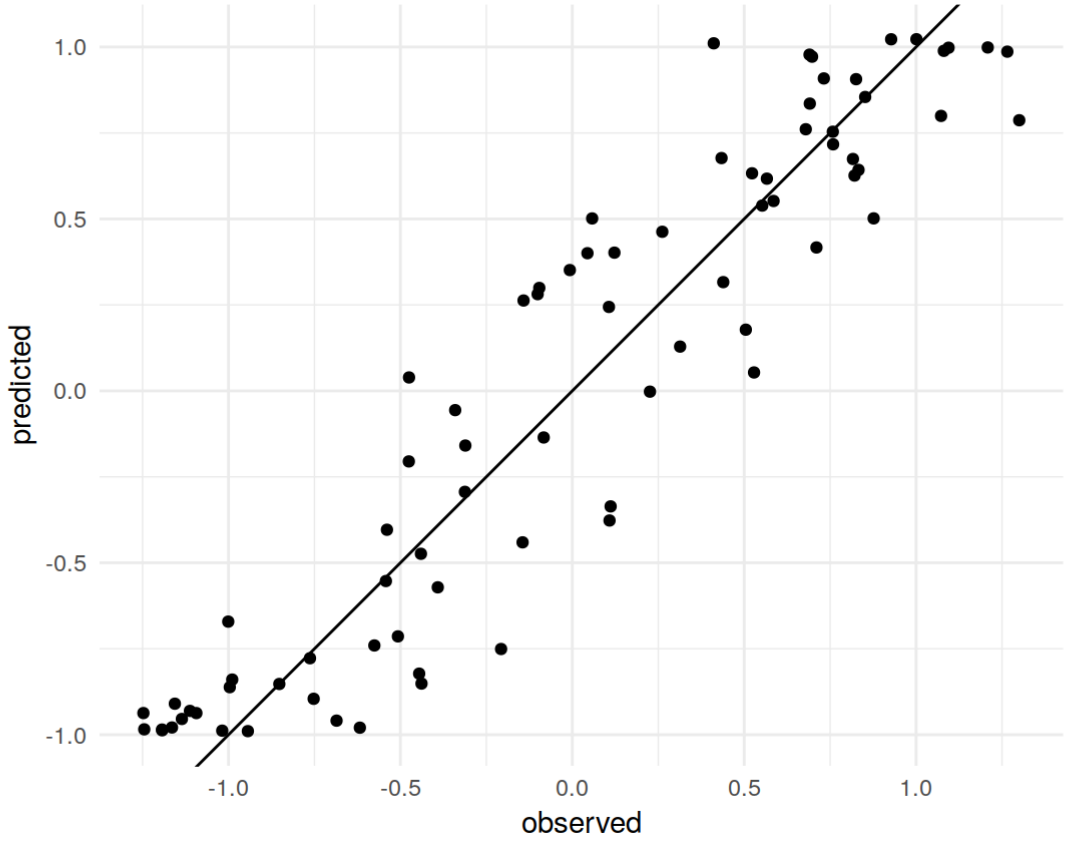
We see that for **this single resampling iteration** span = 0.5 yields the best performance (lowest RMSE). To have more confidence, we would repeat the resampling, model fitting and validation several times.

We also see visually, that `span = 0.5` provides the best model fit:



Hyperparameter tuning

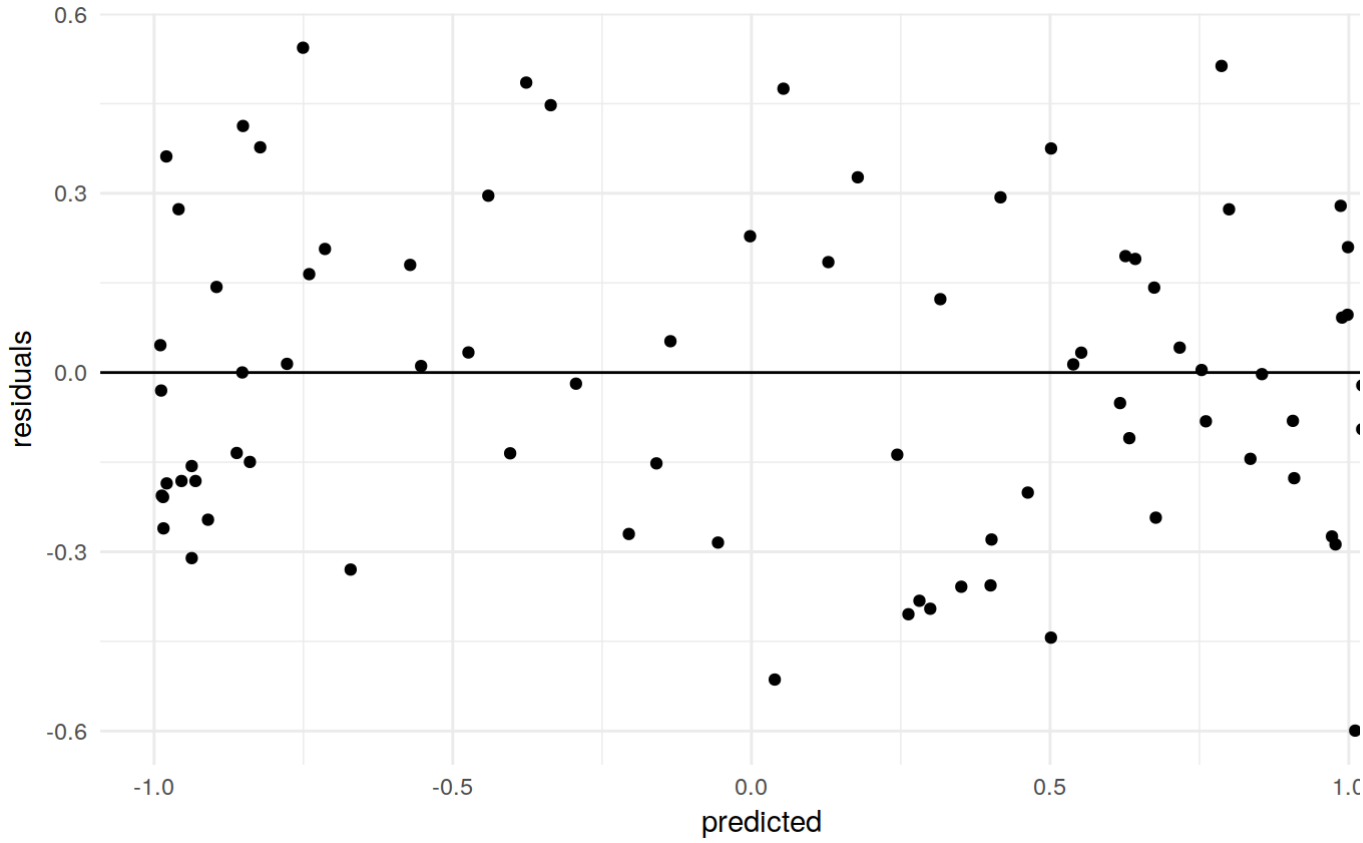
Diagnostic plots for model with span = 0.5:



WZB Wissenschaftszentrum Berlin für Sozialforschung

Hyperparameter tuning

Diagnostic plots for model with span = 0.5:



WZB Wissenschaftszentrum Berlin für Sozialforschung

Data splitting and resampling

In order to get accurate performance measurements and their standard errors / CIs, we need to repeat the resampling and model fitting. There are several common techniques for that:

(repeated) k-fold cross validation

- randomly split data into \(k\) subsamples of roughly equal size
- iterate with \(i = 1, 2, ... k\), use the \(i\)th subsample as test set, all others as training set
- if sample size is low, it is common to repeat the process several times

bootstrapping

- take a random sample of the data with replacement (of the same size as the original data)
- some observations occur multiple times in the bootstrap sample, others are left out → "out-of-bag" samples
- fit model using bootstrap sample, test with "out-of-bag" samples

Performance metrics

Which measures of performance to calculate depend on the prediction problem. Common measures include:

- RMSE, MAE (mean absolute error), R^2 for predicting numeric outcomes (regression)
- Accuracy, ROC, AUC for predicting categorical outcomes (classification – more on that in the next session!)

Cost functions may also account for domain-specific adjustments, e.g. a model for political campaigning may take into account travelling distances (by favoring a model with lower travelling distance to a model with higher travelling distance even if the latter would produce better voter outercome).

Model selection

After repeated model fitting you get a performance estimate for each hyperparameter set. Again, there are several strategies for selecting the final set of hyperparameters:

- simply select the set with the best performance (e.g. lowest RMSE)
- select the simplest model within one standard deviation range from the best performance measure
- selection based on visual inspection (performance plot)

A note on computational complexity

The model tuning process may be **very computationally intensive**. This largely depends on:

- sample size and number of predictors
- number of model types
- number of parameter sets per type
- type and number of data resampling iterations

Example:

- 2 model types (e.g. ridge regression, regression tree)
- 5 parameter sets for each model type
- 10-fold cross-validation with 5 repetitions
- **makes $2 \cdot 5 \cdot 10 \cdot 5 = 500$ model fits**

It may take hours to days to compute. In order to speed up computations, you should use **parallel processing**. This will run the computations in parallel on several CPU cores. Modern laptops have 4 to 8 CPU cores. For larger projects, consider using a dedicated cluster machine (e.g. theia at WZB has 64 CPU cores).

Practical Machine Learning with **caret**

The `caret` package

> The `caret` package (short for Classification and Regression Training) is a set of functions that attempt to streamline the process for creating predictive models. – caret documention manual

→ provides a set of tools for all the steps outlined in the previous slides together with a **unified interface to numerous ML / statistical modeling algorithms**

- developed by Max Kuhn & contributors
- see CRAN page and official documentation
- also introduced in Kuhn & Johnson 2016

Our example data set

Housing Values in Suburbs of Boston: Data set `Boston` from package `MASS`.

- 506 observations with 13 predictors and response `medv` ("median value of owner-occupied homes in $1000s")
- predictors include:
  - `crim`: per capita crime rate
  - `nox`: nitrogen oxides concentration
  - `rm`: average number of rooms per dwelling
  - `tax`: full-value property-tax rate per $10,000s `
  - `ptratio`: pupil-teacher ratio by town
  - `lstat`: lower status of the population (percent)
- see `?Boston` for full list of predictors

Our goal: Predict the median value of a home (response variable `medv`) from the given predictors.

Data splitting

We load the required packages and divide the original data set into training and test data sets:

```
library(MASS)   # for Boston data set
library(dplyr)  # for data transformations
library(caret)

# use ~75% of the data for training; get randomly sampled row indices
in_train_indices <- createDataPartition(Boston$medv, p = 0.75, list = FALSE)

boston_train <- Boston[in_train_indices,]
boston_test <- Boston[-in_train_indices,]
```

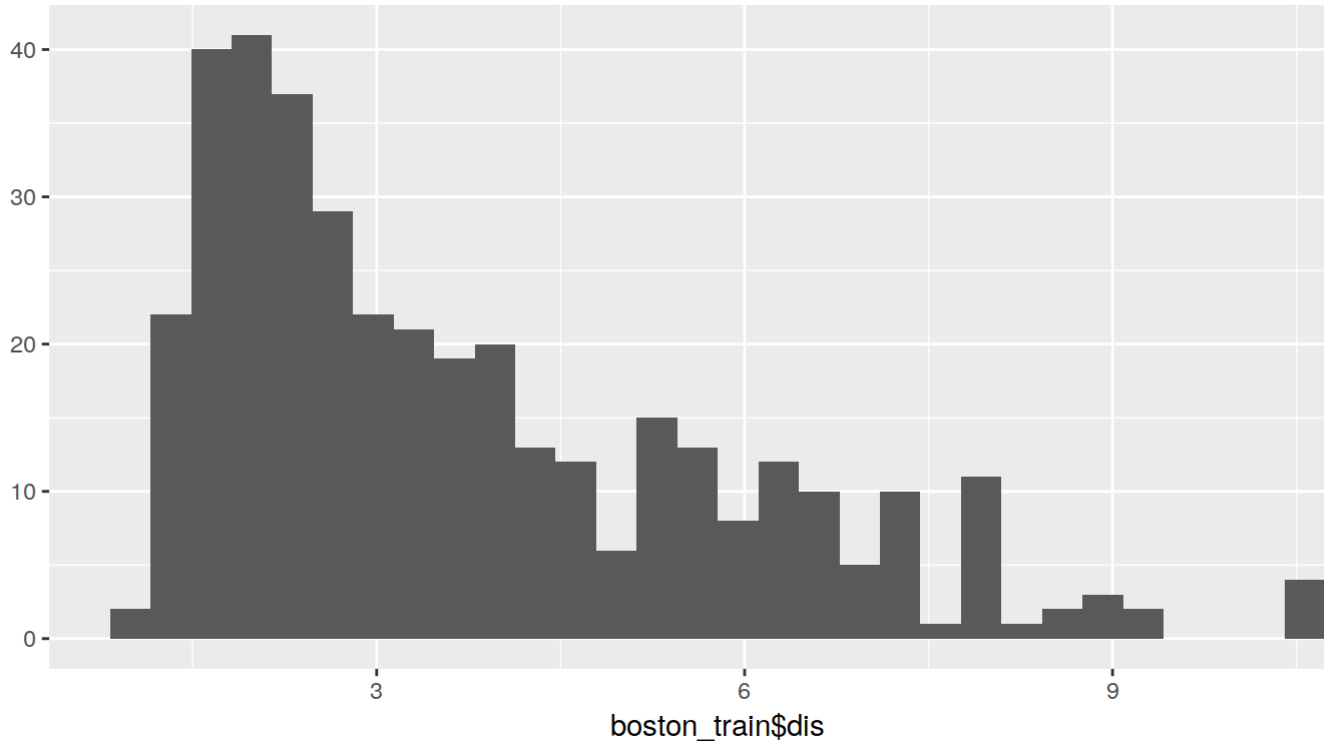
Until final model validation, we'll only work with `boston_train`!

WZB Wissenschaftszentrum Berlin für Sozialforschung

Data pre-processing

One of the first things could be to identify skew and outliers in the predictor's distributions. Histograms can be used for that:

```
# predictor `dis` (weighted mean of distances to five Boston employment cent
qplot(boston_train$dis, bins = 30)
```

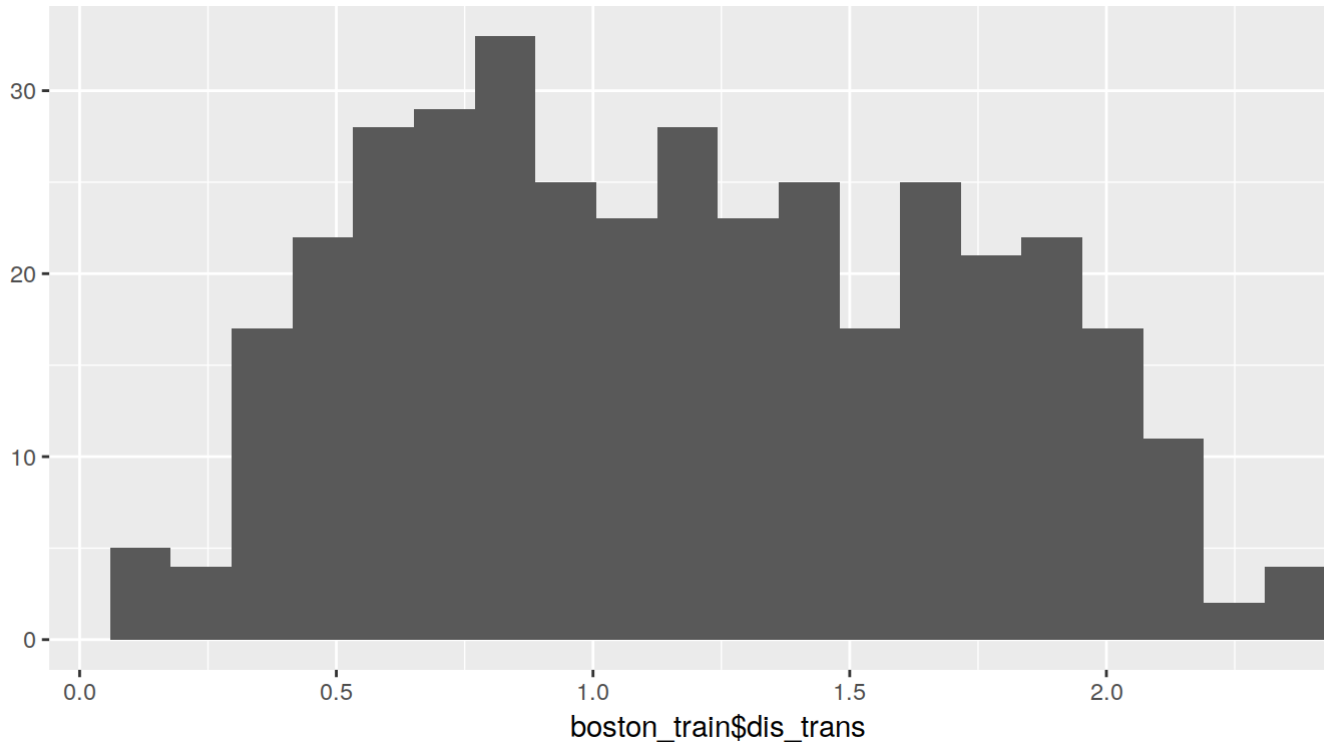


WZB
Wissenschaftszentrum Berlin für Sozialforschung

Data pre-processing

We can apply a logarithmic transformation to reduce the skew:

```
boston_train$dis_trans <- log(boston_train$dis)
boston_train$dis <- NULL
qplot(boston_train$dis_trans, bins = 20)
```



WZB
Wissenschaftszentrum Berlin für Sozialforschung

Data pre-processing

It's also helpfull to identify skewed distributions via the sample skewness statistic (Joanes & Gill 1998):

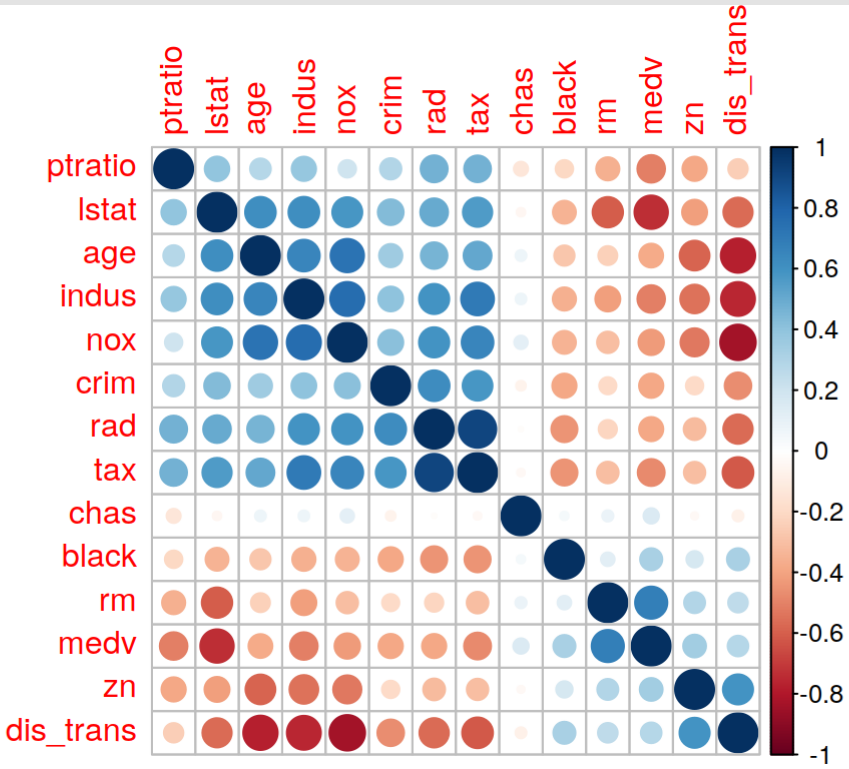
```
library(e1071)
apply(boston_train, 2, skewness)
```

```
##       crim         zn      indus       chas        nox         rm
##  5.1630181  2.2216076  0.2702900  3.1158660  0.7304972  0.3507523
##        age        rad        tax    ptratio      black      lstat
## -0.6512674  0.9688603  0.6532567 -0.7535644 -2.8878938  0.9179482
##       medv  dis_trans
##  1.1047772  0.1611842
```

Removing highly correlated variables

A correlation plot reveals pairwise highly correlated variables:

```
library(corrplot)
correl <- cor(boston_train)
corrplot(correl, order = 'hclust')
```



WZB Wissenschaftszentrum Berlin für Sozialforschung

Removing highly correlated variables

Kuhn & Johnson 2016 suggest a "heuristic approach […] to remove the minimum number of predictors to ensure that all pairwise correlations are below a certain threshold." It is implemented in `findCorrelation`:

```
corr_var_indices <- findCorrelation(correl, cutoff = 0.9)
names(boston_train)[corr_var_indices]
```

```
## [1] "tax"
```

Using a cutoff correlation value of 0.9, this only identified `tax` to be worth removing from the data set:

```
boston_train <- boston_train[, -corr_var_indices]
```

Tuning a ridge regression model

As an example, we'll use ridge regression (Hoerl & Kennard 1970) to built a predictive model from our training data. Ridge regression is an extension to OLS regression which adds a penalty on large coefficients. The hyperparameter $\lambda \in [0,1]$ controls the strength of the penalty.

We can use model tuning on the training data in order to find a λ that corresponds with the lowest RMSE.

Creating hyperparameter candidates

We want to evaluate different values \(0, 0.01, ..., 0.1\) for \(\lambda\). We create a data frame for this with a single column `.lambda` (the `.` before the parameter name is necessary by convention):

```
(ridge_hyperparam_sets <- data.frame(.lambda = seq(0, 0.1, by = 0.01)))
```

```
##    .lambda
## 1     0.00
## 2     0.01
## 3     0.02
## 4     0.03
## 5     0.04
## 6     0.05
## 7     0.06
## 8     0.07
## 9     0.08
## 10    0.09
## 11    0.10
```

Some model algorithms require several parameters for which an optimal combination in sought during model tuning. You can generate combinations using `expand.grid()`.

Specifying a resampling strategy

The resampling strategy for the tuning process can be specified via `trainControl`. The main parameter is `method` which, among others, can be set to:

- `'boot'`: bootstrapping
- `'cv'`: k-fold cross-validation
- `'repeatedcv'` repeated k-fold cross-validation

We'll use 10-fold cross-validation with 5 repeats:

```
train_ctrl <- trainControl(method = 'repeatedcv',
                           number = 10, repeats = 5)
```

Running the model tuning

Now the actual tuning process with repeated cross-validation can be started with `train()`. We pass it:

- predictors `boston_train_X`
- response `boston_train_Y`
- modeling method to use (`'ridge'`)
- hyperparameters to be evaluated (`tuneGrid = ridge_hyperparam_sets`)
- resampling strategy (`trControl = train_ctrl`)

Note that the input data should be centered and scaled (Hastie et al. 2017, p.63) so we additionally set `preProcess` methods.

```
boston_train_X <- dplyr::select(boston_train, -medv)  # predictors
boston_train_Y <- boston_train$medv                   # outcome

tuning_ridge <- train(boston_train_X, boston_train_Y, method = 'ridge',
                      tuneGrid = ridge_hyperparam_sets,
                      trControl = train_ctrl,
                      preProcess = c('center', 'scale'))
```

Inspecting the tuning results

```
tuning_ridge
```

```
Ridge Regression
...
Pre-processing: centered (11), scaled (11)
Resampling: Cross-Validated (10 fold, repeated 5 times)
Summary of sample sizes: 345, 343, 343, 342, 344, 343, ...
Resampling results across tuning parameters:

  lambda  RMSE      Rsquared   MAE
  0.00    4.774223  0.7378075  3.480828
  0.01    4.769590  0.7381880  3.459474
  0.02    4.768482  0.7382246  3.442588
  0.03    4.769962  0.7380142  3.429599
  ...
  0.10    4.815535  0.7335818  3.407671

RMSE was used to select the optimal model using the smallest value.
The final value used for the model was lambda = 0.02.
```

We see a table of performance measures for each evaluated `lambda`. The best model was chosen with `lambda = 0.02` as it achieved the lowest RMSE.

Inspecting the tuning results

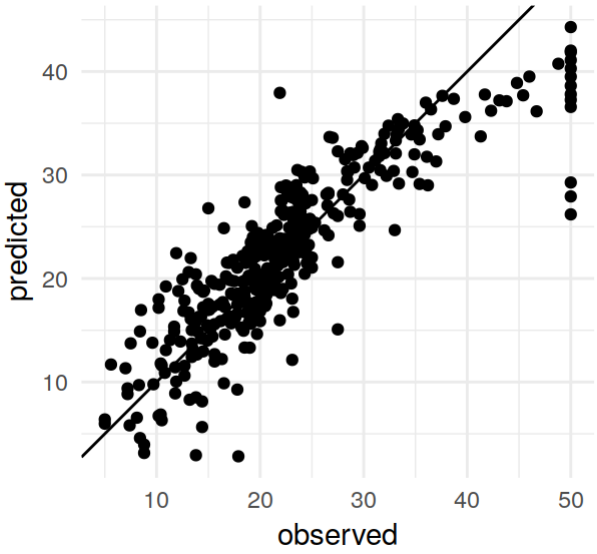
The best model is off by about $4,780 (RMSE) or $3,440 (MAE). We can inspect the fit quickly by re-predicting the training data:

```
# use the same preprocessing as in training first
boston_train_X_scaled <- predict(tuning_ridge$preProcess, newdata = boston_t
# we can access the selected model via `tuning_ridge$finalModel`.
ridge_pred <- predict(tuning_ridge$finalModel, newx = boston_train_X_scaled,
                      s = 1, type = 'fit', mode = 'fraction')$fit
```

WZB Wissenschaftszentrum Berlin für Sozialforschung

Inspecting the tuning results

```
qplot(boston_train_Y, ridge_pred, geom = 'point') +
  geom_abline(intercept = 0, slope = 1) +
  coord_fixed() + xlab('observed') + ylab('predicted') +
  theme_minimal()
```



We can see that there are some problems especially on the upper end of the response scale. Also, there seem to be some strange values of exactly 50.00 for the response which we should investigate further.

Interpreting the model

Ridge regression does not produce a "black box" model. We can have a look at the model's coefficients:

```
predict(tuning_ridge$finalModel, s = 1,
        type = 'coefficients', mode = 'fraction')$coefficients
```

```
##       crim      indus       chas        nox         rm        age
## -1.4718570 -1.1913948  0.7673612 -2.8105930  2.8128855 -0.2938233
##        rad    ptratio      black      lstat  dis_trans
##  1.2628486 -2.0301905  0.7796135 -3.8201206 -4.2611391
```

We can see that the log transformed value of the distance to employment centres (`dis_trans`) has big negative effect on house values, as well as the percentage of "lower status of the population" (`lstat`). The average number of rooms (`rm`) has a positive effect.

What next?

Besides investigating the problem with the large number of outcome values of exactly 50.00 (which probably is a data set issue), the next steps would include:

- tuning different model(s)
- final validation of the best models using the held-out test data
- selecting the final model

In this week's excercise you can complete a script with these steps.

Literature

- Kuhn & Johnson 2016: Applied Predictive Modeling
- O'Neil 2016: Weapons of Math Destruction
- Ribeiro et al. 2016: "Why Should I Trust You?" – Explaining the Predictions of Any Classifier
- Hastie et al. 2017: The Elements of Statistical Learning

Tasks

See dedicated tasks sheet on the tutorial website.