Hands-on Machine Learning with R - Module 1

Hands-on webinar

Katrien Antonio & Roel Henckaerts

hands-on-machine-learning-R-module-1 | December 10 & 17, 2020

Prologue

Introduction

Course

https://github.com/katrienantonio/hands-on-machine-learning-R-module-1

The course repo on GitHub, where you can find the data sets, lecture sheets, R scripts and R markdown files.

Us

https://katrienantonio.github.io/ & https://henckr.github.io/

katrien.antonio@kuleuven.be & roel.henckaerts@kuleuven.be

(Katrien) Professor in insurance data science

(Roel) PhD student in insurance data science

Checklist

☑ Do you have a fairly recent version of R?

```
version$version.string
## [1] "R version 4.0.3 (2020-10-10)"
```

☑ Do you have a fairly recent version of RStudio?

```
RStudio.Version()$version
## Requires an interactive session but should return something like "[1] ‘1.3.1093’"
```

☑ Have you installed the R packages listed in the software requirements?

or

☑ Have you created an account on RStudio Cloud (to avoid any local installation issues)?

Why this course?

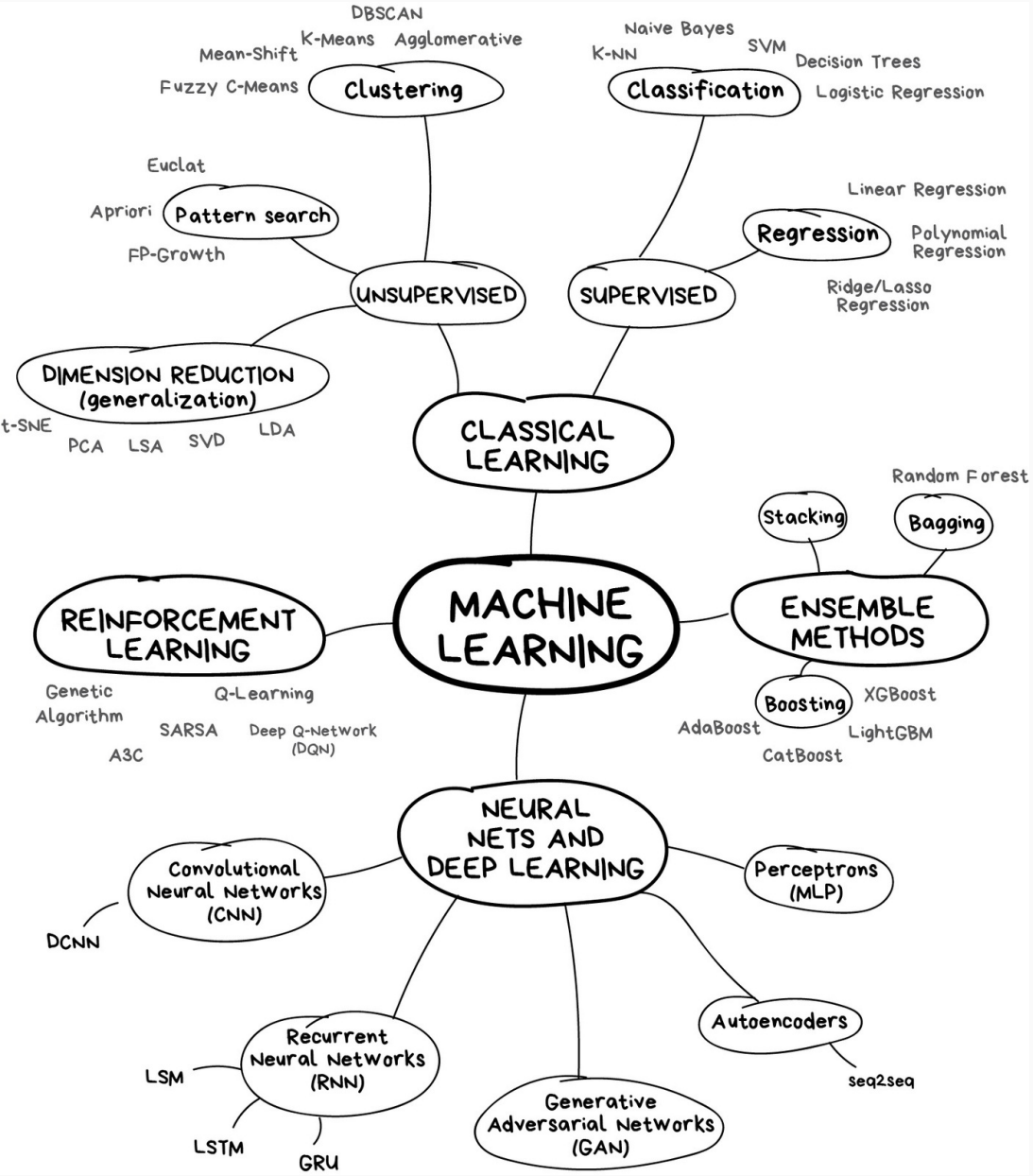
The goals of this course 🚀

- develop practical **machine learning (ML) foundations**
- **fill in the gaps** left by traditional training in actuarial science or econometrics
- focus on the use of ML methods for the **analysis of frequency + severity data**, but also **non-standard data** such as images
- **explore** a substantial range of **methods (and data types)** (from GLMs to deep learning), but - most importantly - **build foundation** so that you can explore other methods (and data types) yourself.

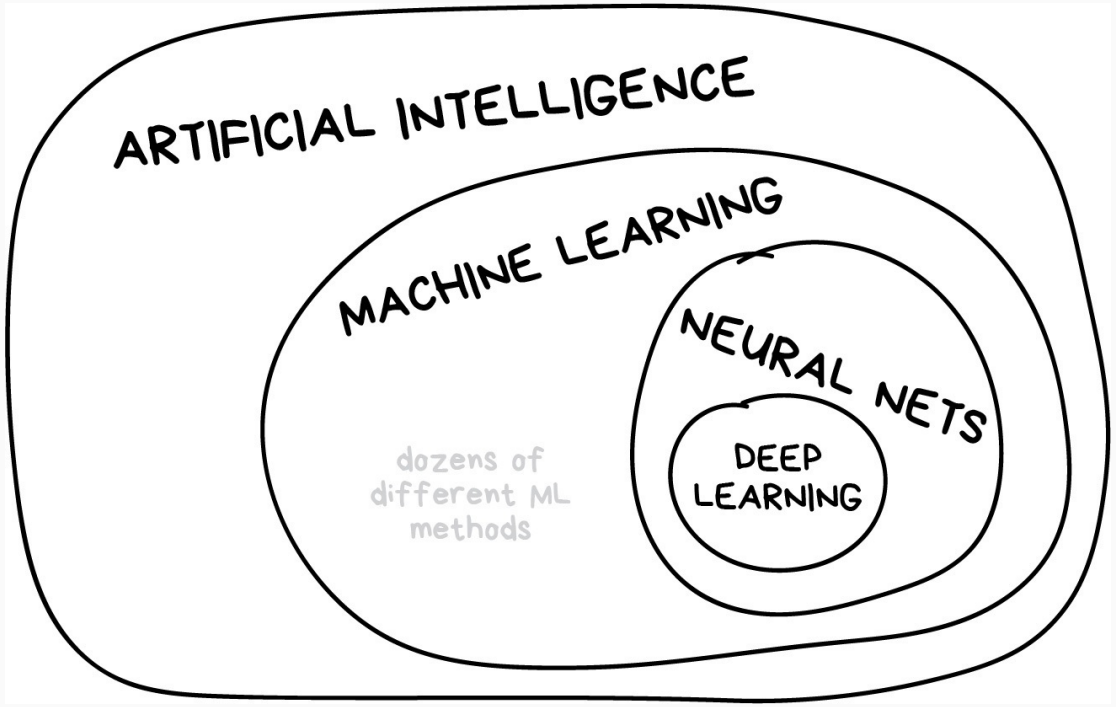
> *"In short, we will cover things that we wish someone had taught us in our undergraduate programs."*
>
> This quote is from the Data science for economists course by Grant McDermott.

Module 1's Outline

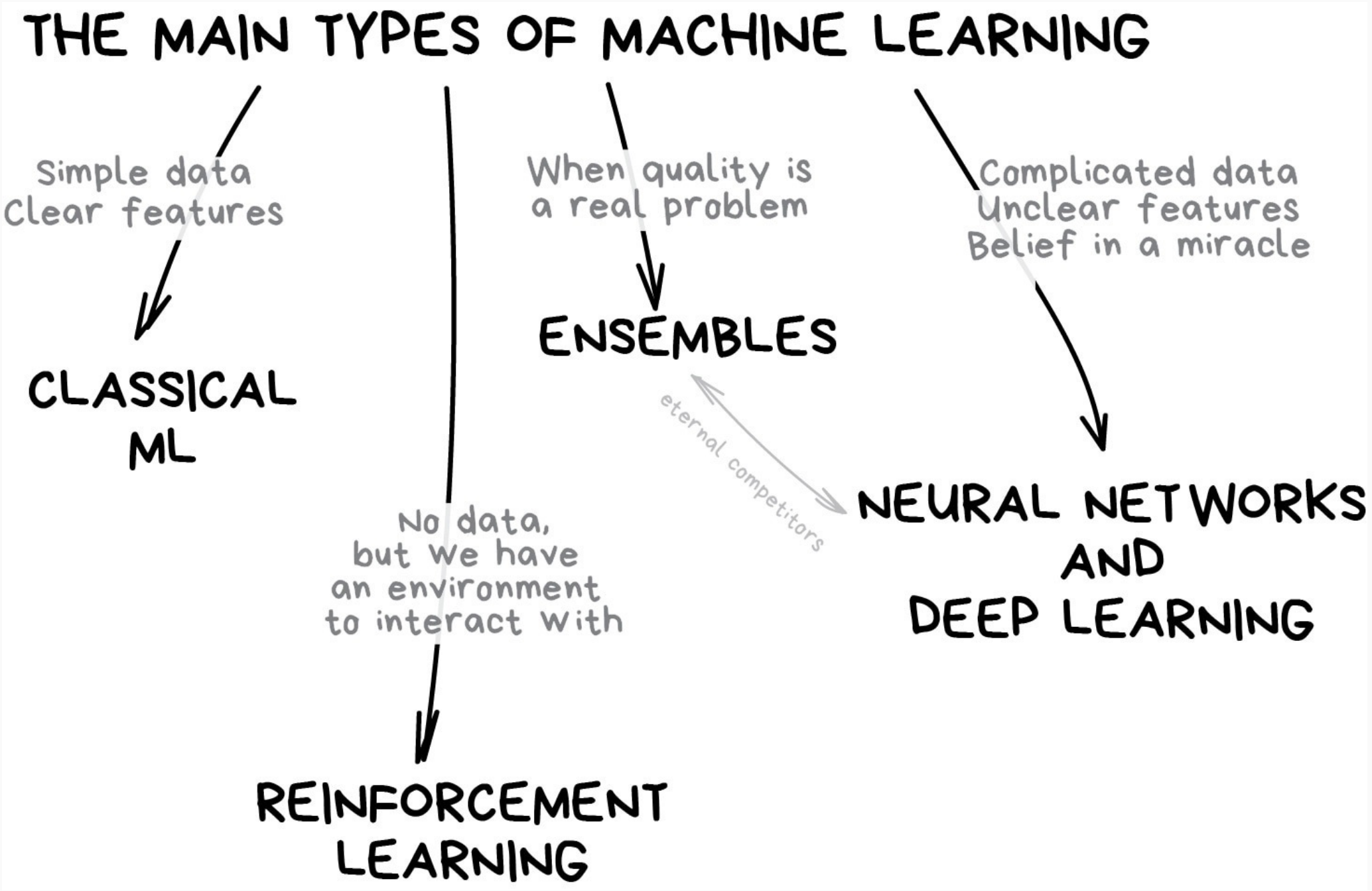
- Prologue
- Knowing me, knowing you: statistical and machine learning
  - Supervised and unsupervised learning
  - Regression and classification
  - Statistical modeling: the two cultures
- Model accuracy and loss functions
- Overfitting and bias-variance tradeoff
- Data splitting, Resampling methods
- Parameter tuning
  - with {caret}, {rsample} and {purrr}
- Target and feature engineering
  - Data leakage
  - Pre-processing steps
  - Specifying blue-prints with {recipes}
  - Putting it all together: {recipes} and {caret}/{rsample}
- Regression models
  - Creating models in R and tidy model output with {broom}
  - GLMs with {glm}
  - GAMs with {mgcv}
  - Regularized (G)LMs with {glmnet}.



Some roadmaps to explore the ML landscape...



Source: Machine Learning for Everyone In simple words. With real-world examples. Yes, again.


THE MAIN TYPES OF MACHINE LEARNING
Simple data
Clear features
CLASSICAL ML
No data,
but we have
an environment
to interact with
REINFORCEMENT LEARNING
When quality is
a real problem
ENSEMBLES
eternal competitors
Complicated data
Unclear features
Belief in a miracle
NEURAL NETWORKS AND DEEP LEARNING

Knowing me, knowing you: statistical and machine learning

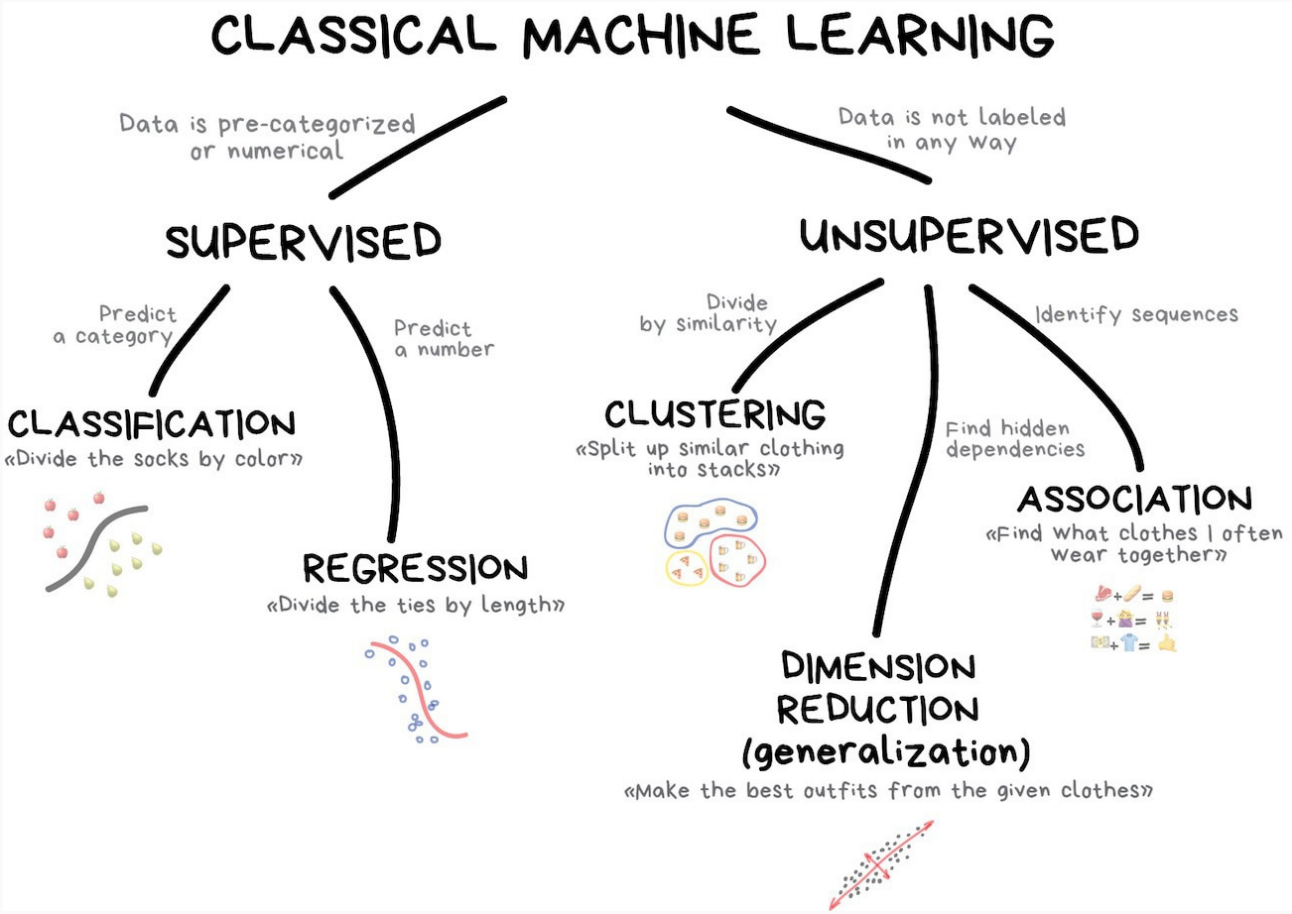
Supervised learning

Supervised learning builds ("learns") a model f (*the Signal*) such that the outcome or target Y can be written as

$$Y = f(x_1, \ldots, x_p) + \epsilon$$

with features $x_1, \ldots, x_p$ and error term ϵ (*the Noise*).

Supervised learners construct **predictive models**.



Picture taken from Machine Learning for Everyone. In simple words. With real-world examples. Yes, again

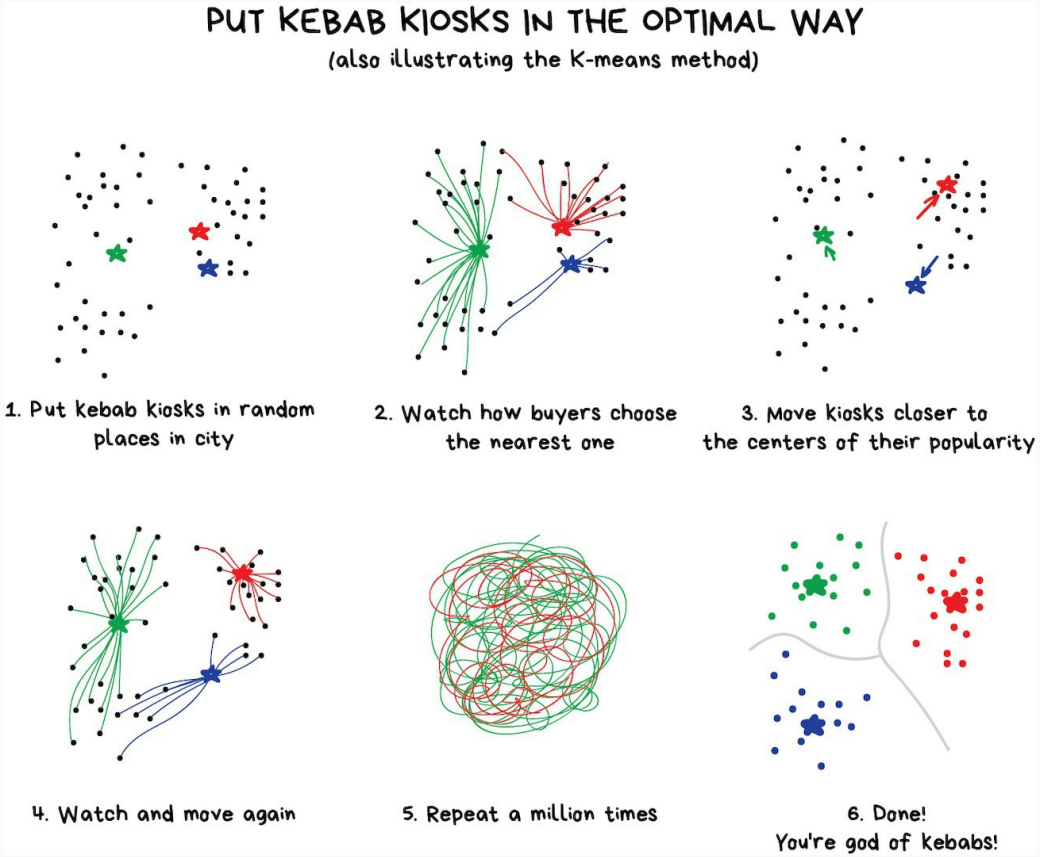
Unsupervised learning

With unsupervised learning there is NO outcome or target Y, only the feature vector $x = (x_1, \ldots, x_p)$.

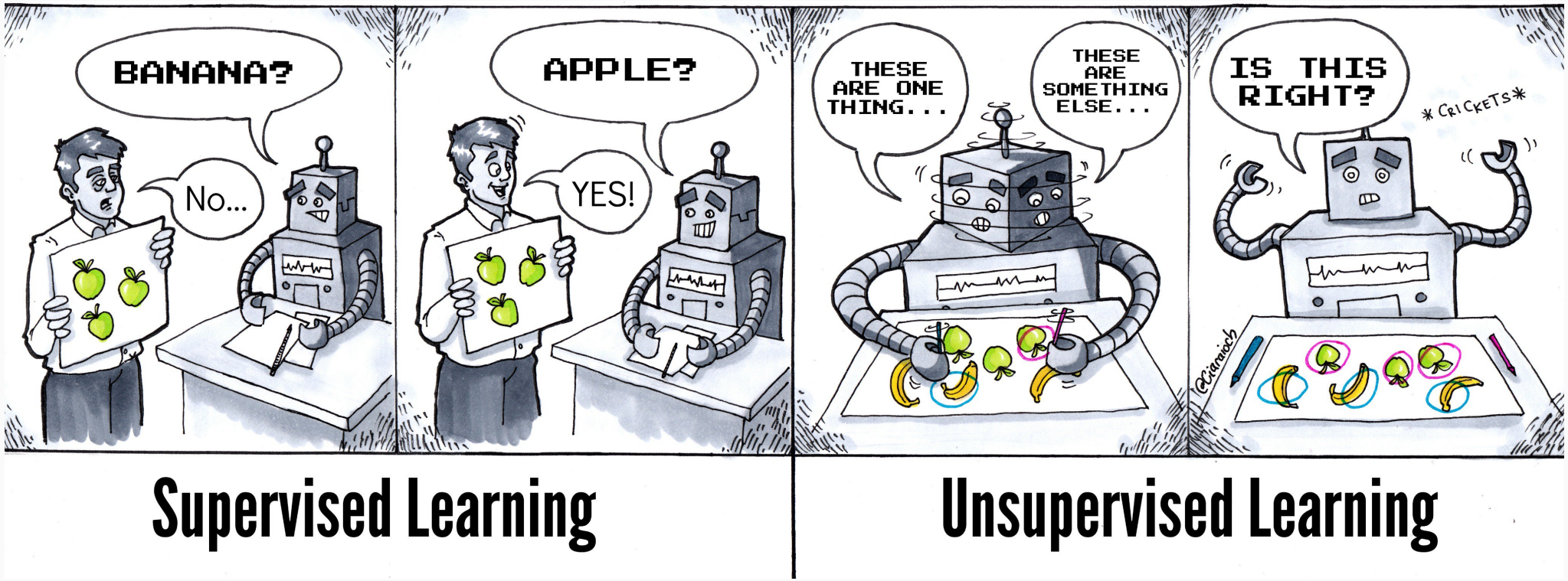
Let n denote the sample size and p the number of features.

Then, X is the $n \times p$ matrix of features, with $x_{i,j}$ observation i on variable or feature j.

Unsupervised learners construct descriptive models, without any *supervising* output, letting the data "speak for itself".



Picture taken from Machine Learning for Everyone. In simple words. With real-world examples. Yes, again



Picture taken from this source.

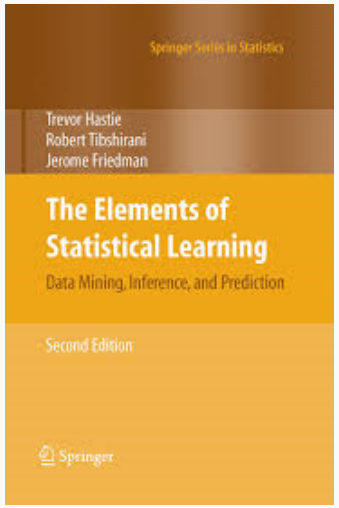
What's in a name?

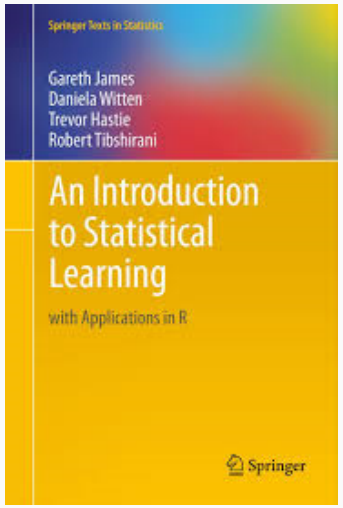
Machine learning constructs algorithms that learn from data.

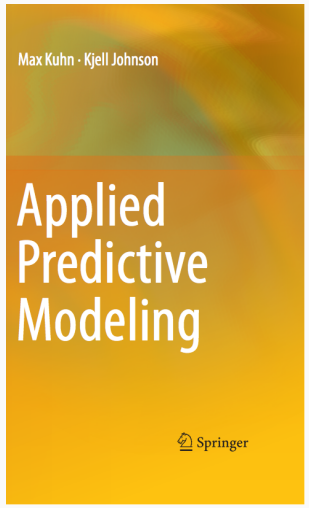
Statistical learning emphasizes statistical models and the assessment of uncertainty.

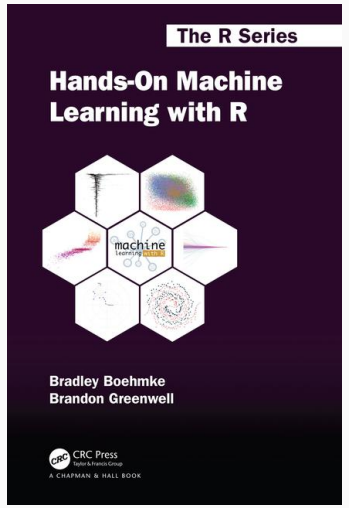
Data science applies mathematics, statistics, machine learning, engineering, etc. to extract knowledge form data.

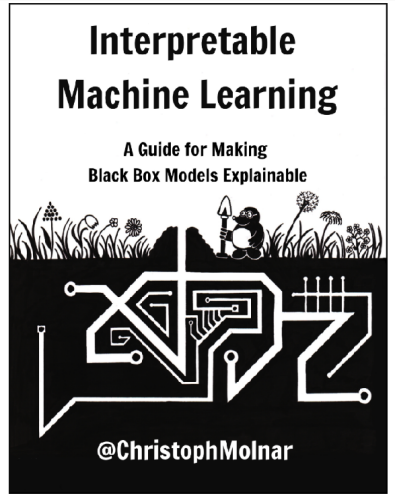
> *"Data Science is statistics on a Mac . "*







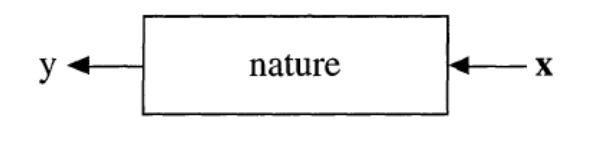




Source: Brandon M. Greenwell on Introduction to Machine Learning in .

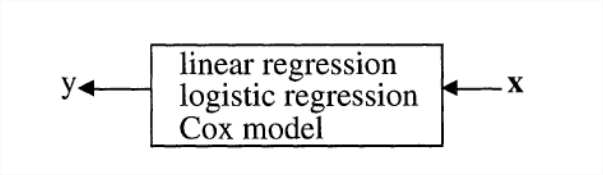
Statistical modeling: the two cultures

Consider a vector of input variables x, being transformed into some vector of response variables y via a black box algorithm.



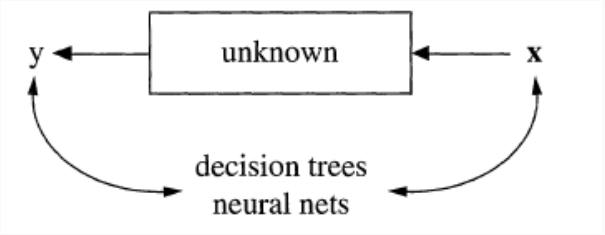
Statistical learning or data modeling culture

- assume statistical model, estimate parameter values
- validate with goodness-of-fit tests and residual inspection



Machine learning or algo modeling culture

- inside of the box is complex and unknown
- find algorithm $f(x)$ to predict y
- measure performance by predictive accuracy



Source: Breiman (2001, Statistical Science) on *Statistical modeling: the two cultures.*

Newspeak from the two cultures

	Statistical learning	**Machine learning**
origin	statistics	computer science
f(x)	model	algorithm
emphasis	interpretability, precision and uncertainty	large scale applicability, prediction accuracy
jargon	parameters, estimation	weights, learning
CI	uncertainty of parameters	no notion of uncertainty
assumptions	explicit a priori assumption	no prior assumption, learn from the data

Source: read the blog Why a mathematician, statistician and machine learner solve the same problem differently

Your turn

As discussed in the lecture, many problems in ML can be approached as a regression, classification or clustering problem.

Q: consider the following **three problem settings** and **label them** as regression, classification or clustering.

1. In disability insurance: how do disability rates depend on the state of the economy (e.g. GDP)?
2. In MTPL insurance: predict whether a claim is attritional or large, *in casu* a claim that exceeds the threshold of 100 000 EUR?
3. How can we group customers based on the insurance products they bought from the company?

Model accuracy and loss functions

Predictive modeling

How to use the observed data to learn or to estimate the unknown $f(.)$?

$$y = f(x_1, x_2, \ldots, x_p) + \epsilon.$$

How do I **estimate** $f(.)$ - one way to phrase *all questions* that underly statistical & machine learning.

Take-aways 📣 - main reasons we want to learn about $f(.)$

prediction

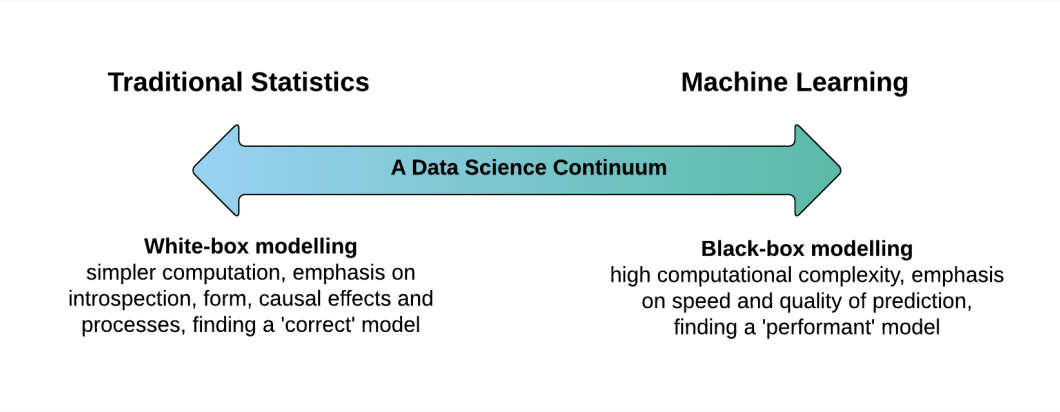
predict the target y as $\hat{f}(x)$

📦 - as black box setting?

inference

how does target y depend on features x?

📭 - as white box setting?



Prediction errors

Why we're stuck with **irreducible error**

assume $\hat{f}$ and x given, then

$$
\begin{aligned}
E\left[\{y - \hat{y}\}^2\right] &= E\left[\left\{f(x) + \epsilon - \hat{f}\,(x)\right\}^2\right] \\
&= \underbrace{\left[f(x) - \hat{f}\,(x)\right]^2}_{\text{Reducible}} + \underbrace{\text{Var}(\epsilon)}_{\text{Irreducible}}
\end{aligned}
$$

In **less math**:

- if ϵ exists, then x cannot perfectly explain y
- so even if $\hat{f} = f$, we still have irreducible error.

Thus, to form our **best predictors**, we will **minimize reducible error**.

Model accuracy

We assess **model** or **predictive accuracy** by evaluating how well predictions actually match observed data.

Use **loss functions**, i.e. metrics that compare predicted values to actual values.

Regression, use e.g. the **Mean Squared Error (MSE)**

$$\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{f}(x_i))^2,$$

Recall: $y_i - \hat{y}_i = y_i - \hat{f}(x_i)$ is the prediction error.

Objective ◎ : minimize!

Classification, use e.g. the **cross-entropy** or **log loss**

$$-\frac{1}{n}\sum_{i=1}^{n}\left(y_i \cdot \log(p_i) + (1 - y_i) \cdot \log(1 - p_i)\right).$$

Objective ◎ : minimize!

Many other useful loss functions (e.g. deviance in regression, Gini index in classification).

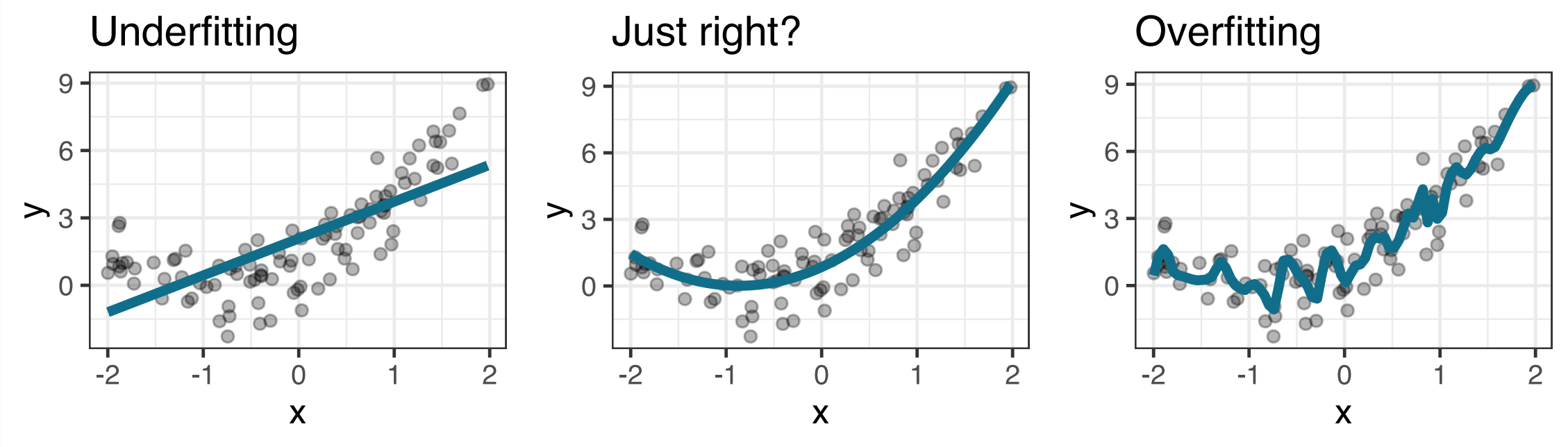
Take-away 📣 - a loss function emphasizes certain types of errors over others → pick a meaningful one!

Overfitting and bias-variance trade off

Overfitting

The Signal and the Noise discussion!

Which of the following three models (in green-blue-ish) will best generalize to new data?



Inspired by Brandon Greenwell's Introduction to Machine Learning in R.

Overfitting (cont.)

With a small training error, but large test error, the model is overfitting or working too hard!

The expected value of the **test MSE**:

$$E\Big(y_0 - \hat{f}\,(x_0)\Big)^2 = \mathrm{Var}(\hat{f}\,(x_0)) + [\mathrm{Bias}(\hat{f}\,(x_0))]^2 + \mathrm{Var}(\epsilon).$$

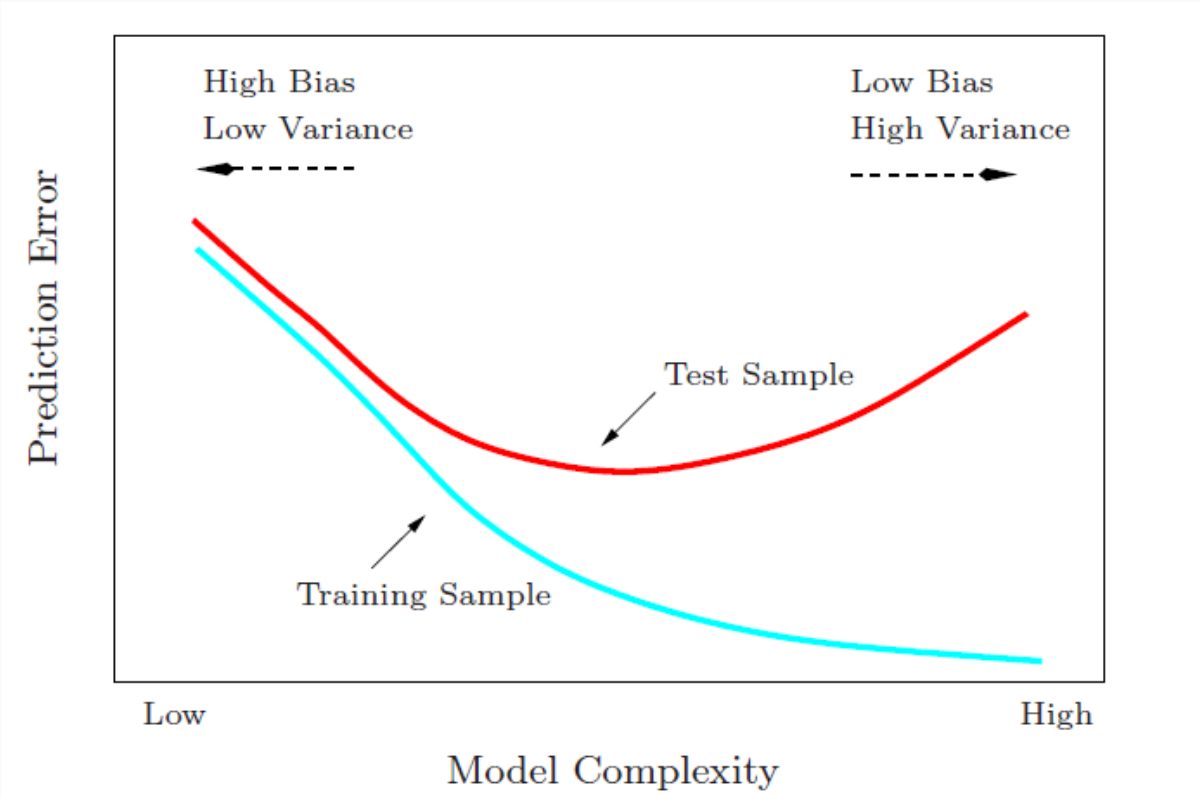
In general - with more flexible methods

- variance ⬆ and bias ⬇
- their relative rate of change determines whether the test error increases or decreases

Take-aways 📢

- U-shape curves of **test MSE** w.r.t model flexibility
- the **bias-variance tradeoff** is central to quality prediction.

Bias-variance trade off

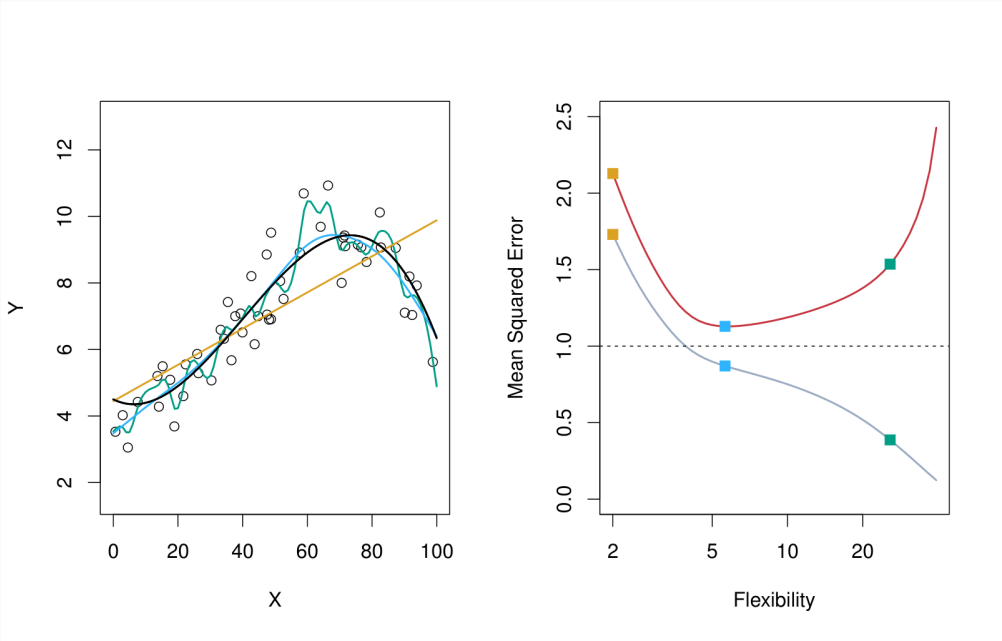


Source: James et al. (2013) on http://faculty.marshall.usc.edu/gareth-james/ISL/.

Your turn

Data are generated from: $y = f(x) + \epsilon$, with the black curve as the true f. The orange (linear regression), blue (smoothing splines) and green (smoothing splines) curves are three estimates for f, with increasing level of complexity.

Q: which model do you prefer (orange, blue, green) for each of the following examples? Why?

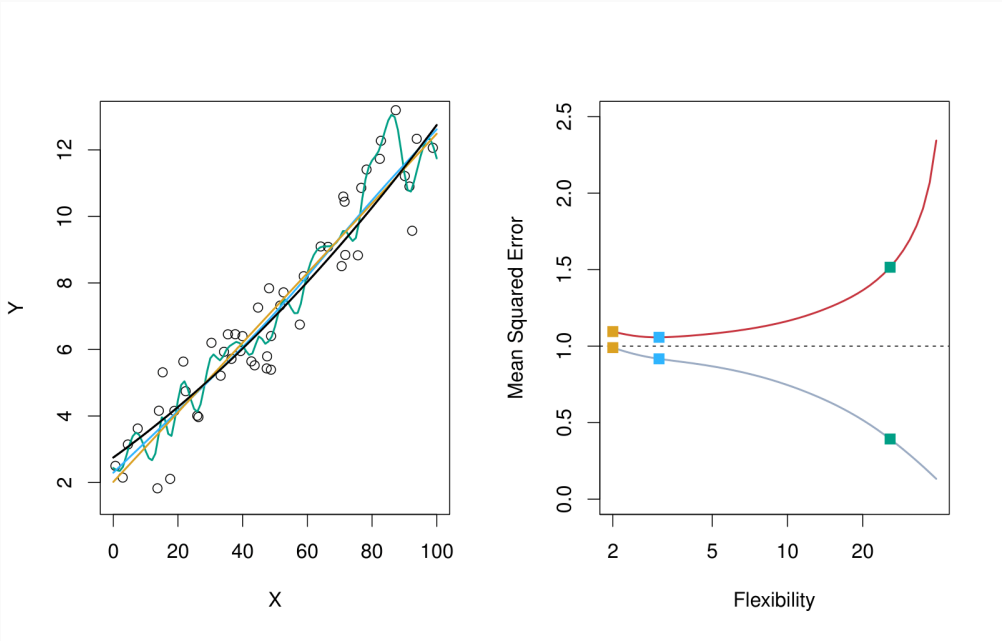


Example from James et al. (2013) on http://faculty.marshall.usc.edu/gareth-james/ISL/.

Your turn

Data are generated from: $y = f(x) + \epsilon$, with the black curve as the true f. The orange (linear regression), blue (smoothing splines) and green (smoothing splines) curves are three estimates for f, with increasing level of complexity.

Q: which model do you prefer (orange, blue, green) for each of the following examples? Why?

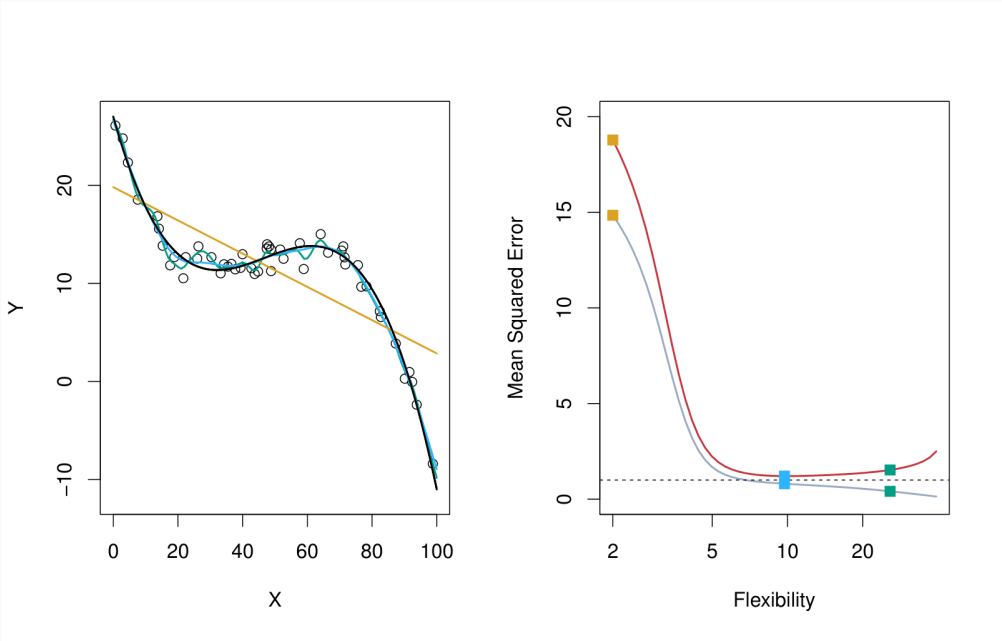


Example from James et al. (2013) on http://faculty.marshall.usc.edu/gareth-james/ISL/.

Your turn

Data are generated from: $y = f(x) + \epsilon$, with the black curve as the true f. The orange (linear regression), blue (smoothing splines) and green (smoothing splines) curves are three estimates for f, with increasing level of complexity.

Q: which model do you prefer (orange, blue, green) for each of the following examples? Why?



Example from James et al. (2013) on http://faculty.marshall.usc.edu/gareth-james/ISL/.

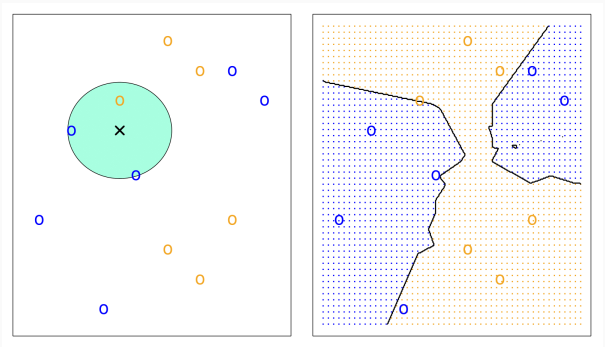
Your turn

The *K*-nearest neighbors (KNN) classifier

- take the *K* observations in the training data set that are 'closest' to test observation x_0, calculate

$$\Pr(Y = j | X = x_0) = \frac{1}{K} \sum_{i \in \mathcal{N}_0} \mathbb{I}(y_i = j).$$

- KNN then assigns the test observation x_0 to the class j with the highest probability, e.g. with *K*=3 (from James et al., 2013)

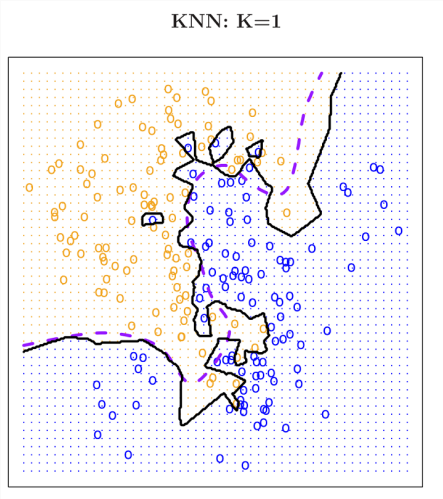


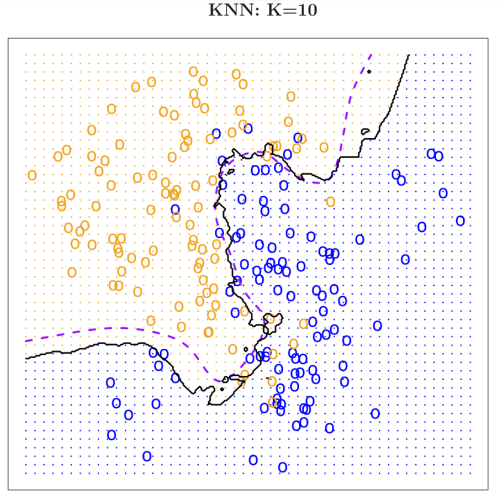
Q: is KNN a supervised learning or unsupervised learning method? Discuss.

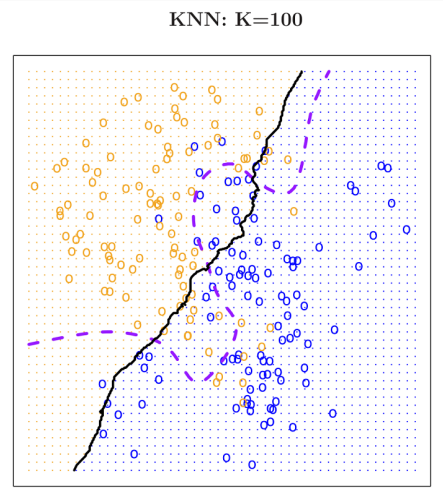
Your turn

The *K*-nearest neighbors (KNN) classifier (cont.)

Now compare KNN with *K* equals 1, 10 and 100.







Q: which classifier do you prefer? Which of these classifiers is under-fitting, which one is over-fitting?

Data splitting and resampling methods with {caret} and {rsample}

Ames Iowa housing data

We will use the Ames Iowa housing data. There are 2,930 properties in the data set.

The `Sale_Price` (target or response) was recorded along with 80 predictors, including:

- location (e.g. neighborhood) and lot information
- house components (garage, fireplace, pool, porch, etc.)
- general assessments such as overall quality and condition
- number of bedrooms, baths, and so on.

More details in De Cock (2011, Journal of Statistics Education).

The raw data are at `http://bit.ly/2whgsQM` but we will use a processed version found in the `AmesHousing` package.

You will load the data with the `make_ames()` function from the `AmesHousing` library, and store the data in the object `ames`:

```
ames ← AmesHousing::make_ames()
```

Data splitting

We fit our model on past data $\{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$ and get $\hat{f}$.

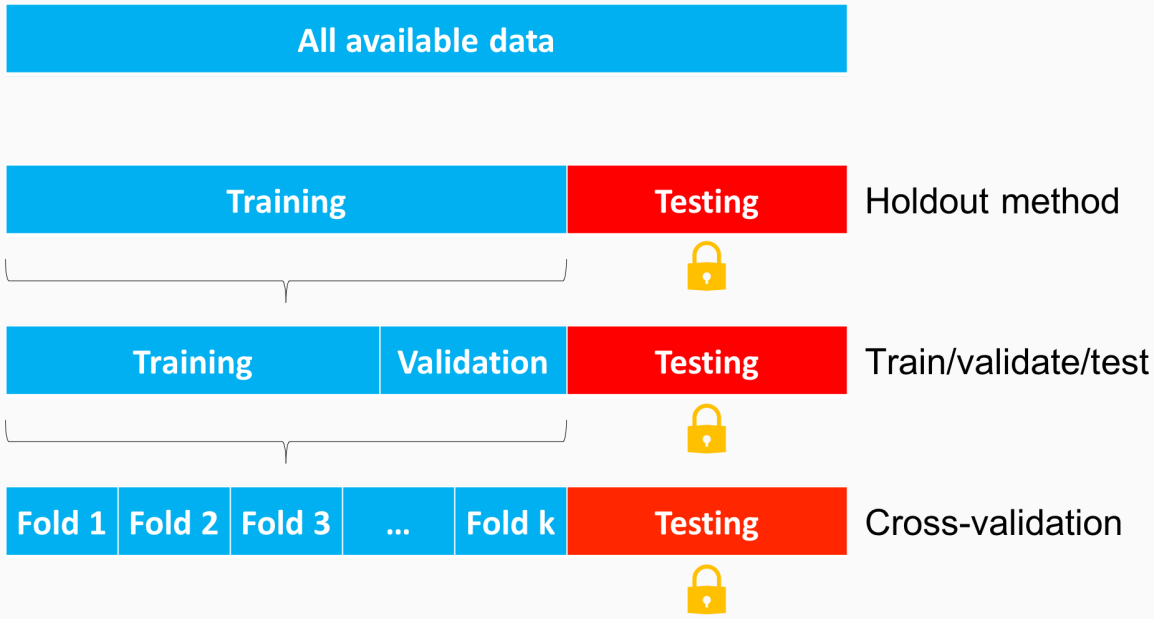
What we want: how does our model generalize to new, unseen data (x_0, y_0), or: is $\hat{f}(x_0)$ close to y_0?

Training set

- to develop, to train, to tune, to compare different settings, ...

Test set

- to obtain unbiased estimate of final model's performance.



Picture taken from Introduction to Machine Learning in R.

Data splitting in base

We first demonstrate the splitting of the `ames` housing data into a training and test set, using `base` R instructions.

```
set.seed(123)
index_1 ← sample(1 : nrow(ames),
                 size = round(nrow(ames) * 0.7))
train_1 ← ames[index_1, ]
test_1  ← ames[-index_1, ]

nrow(train_1)/nrow(ames)
```

Use `set.seed()` for reproducibility.

Data splitting in base

We first demonstrate the splitting of the `ames` housing data into a training and test set, using `base` R instructions.

```
set.seed(123)
index_1 ← sample(1 : nrow(ames),
                 size = round(nrow(ames) * 0.7))
train_1 ← ames[index_1, ]
test_1  ← ames[-index_1, ]

nrow(train_1)/nrow(ames)
```

Sample indices from `1 : nrow(ames)` such that in total 70% of the records is selected.

Vector `index_1` now stores the row numbers of the selected records.

Data splitting in base

We first demonstrate the splitting of the `ames` housing data into a training and test set, using `base` R instructions.

```
set.seed(123)
index_1 ← sample(1 : nrow(ames),
                 size = round(nrow(ames) * 0.7))
train_1 ← ames[index_1, ]
test_1  ← ames[-index_1, ]

nrow(train_1)/nrow(ames)
```

Put the selected records in training set `train_1` by subsetting the original data frame `ames` with the row numbers stored in `index_1`.

Data splitting in base

We first demonstrate the splitting of the `ames` housing data into a training and test set, using `base` R instructions.

```
set.seed(123)
index_1 ← sample(1 : nrow(ames),
                 size = round(nrow(ames) * 0.7))
train_1 ← ames[index_1, ]
test_1  ← ames[-index_1, ]

nrow(train_1)/nrow(ames)
```

Put the not selected records in test set `test_1`.

Data splitting in base

We first demonstrate the splitting of the `ames` housing data into a training and test set, using `base` R instructions.

```
set.seed(123)
index_1 ← sample(1 : nrow(ames),
                 size = round(nrow(ames) * 0.7))
train_1 ← ames[index_1, ]
test_1  ← ames[-index_1, ]

nrow(train_1)/nrow(ames)
## [1] 0.7
```

What is the ratio of the number of records in `train_1` versus original data set `ames`?

Data splitting in {caret}

The {caret} package - short for Classification And REgression Training - contains functions to streamline the model training process for complex regression and classification problems.

With the {caret} package, the function `createDataPartition` will do the job.

```
library(caret)
set.seed(123)
index_2 ← caret::createDataPartition(
                    y = ames$Sale_Price,
                    p = 0.7,
                    list = FALSE)
train_2 ← ames[index_2, ]
test_2  ← ames[-index_2, ]

nrow(train_2)/nrow(ames)
```

Load the library {caret}.

Use `set.seed()` for reproducibility.

Data splitting in {caret}

The {caret} package - short for Classification And REgression Training - contains functions to streamline the model training process for complex regression and classification problems.

With the {caret} package, the function `createDataPartition` will do the job.

```
library(caret)
set.seed(123)
index_2 ← caret::createDataPartition(
                    y = ames$Sale_Price,
                    p = 0.7,
                    list = FALSE)
train_2 ← ames[index_2, ]
test_2  ← ames[-index_2, ]

nrow(train_2)/nrow(ames)
```

`createDataPartition` takes in `y` the vector of outcomes of the data set we wish to split. `createDataPartition` will do stratified sampling based on levels of `y` (for factor) or groups determined by the percentiles of `y` (for numeric).

The percentage of data that goes to training is `p`.

`list = FALSE` tells the function not to store the results in a list, but in a matrix (here: with 1 column)

Data splitting in {rsample}

The {rsample} package, part of the {tidymodels} initiative of RStudio, is home to a wide very variety of resampling functions.

The documentation is at rsample: the basics.

```
library(rsample)
set.seed(123)
split_1  ← rsample::initial_split(ames, prop = 0.7)
train_3  ← training(split_1)
test_3   ← testing(split_1)

nrow(train_3)/nrow(ames)
```

Load the `rsample` package.

Use `set.seed()` for reproducibility.

Data splitting in {rsample}

The {rsample} package, part of the {tidymodels} initiative of RStudio, is home to a wide very variety of resampling functions.

The documentation is at rsample: the basics.

```
library(rsample)
set.seed(123)
split_1  ← rsample::initial_split(ames, prop = 0.7)
train_3  ← training(split_1)
test_3   ← testing(split_1)

nrow(train_3)/nrow(ames)
```

`initial_split` from the {rsample} package.

Split the data `ames` into a training set and testing set.

`prop` is the proportion of data to be retained as training

Data splitting in {rsample}

The {rsample} package, part of the {tidymodels} initiative of RStudio, is home to a wide very variety of resampling functions.

The documentation is at rsample: the basics.

```
library(rsample)
set.seed(123)
split_1  ← rsample::initial_split(ames, prop = 0.7)
train_3  ← training(split_1)
test_3   ← testing(split_1)

nrow(train_3)/nrow(ames)
```

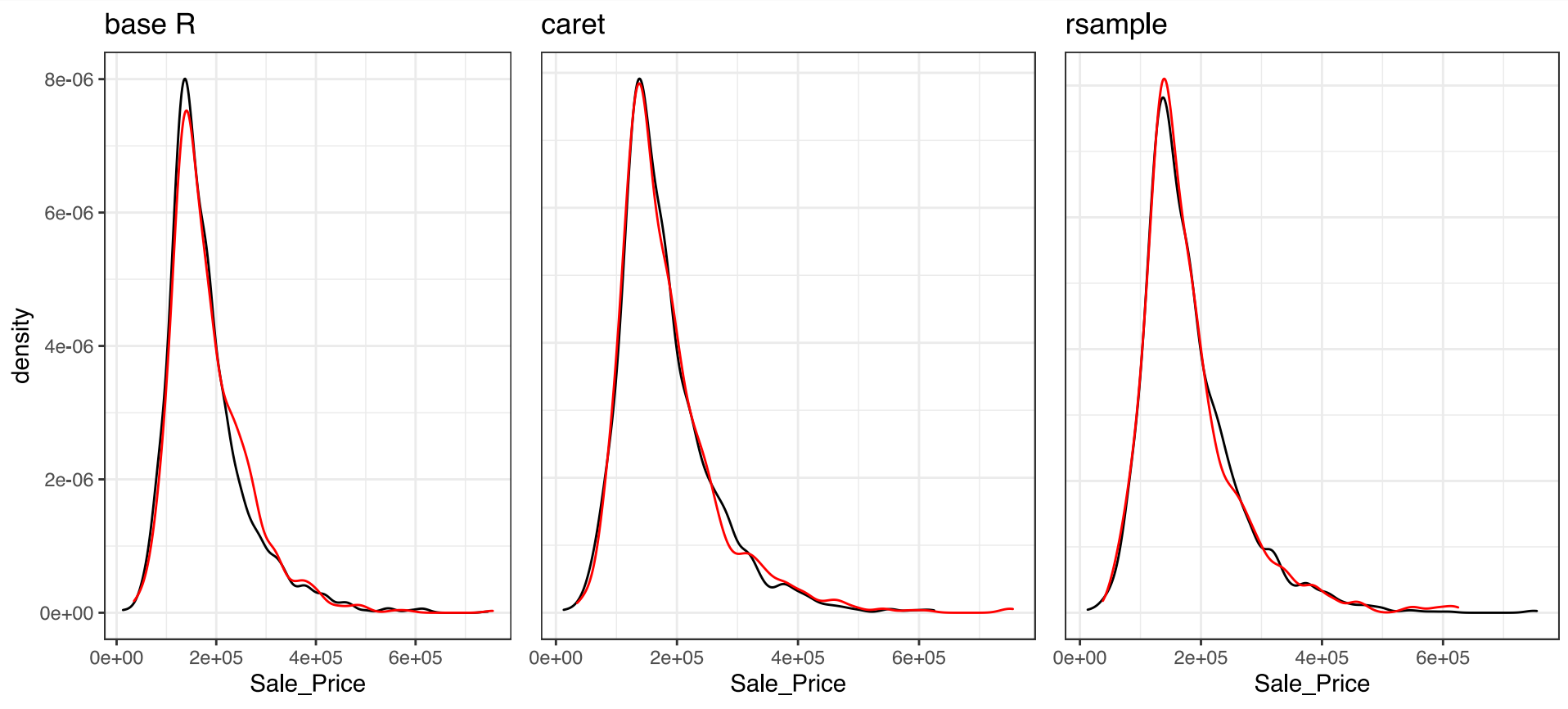
The result of `rsample::initial_split` is an `rset` object.

It is stored in `split_1` and ready for inspection.

Apply the functions `training` and `test` to this object to extract the data in each split.

Data splitting comparison

As a check, we plot the `Sale_Price` as available in the train (in black) vs test (in red) data sets, created by each of the three demonstrated methods.

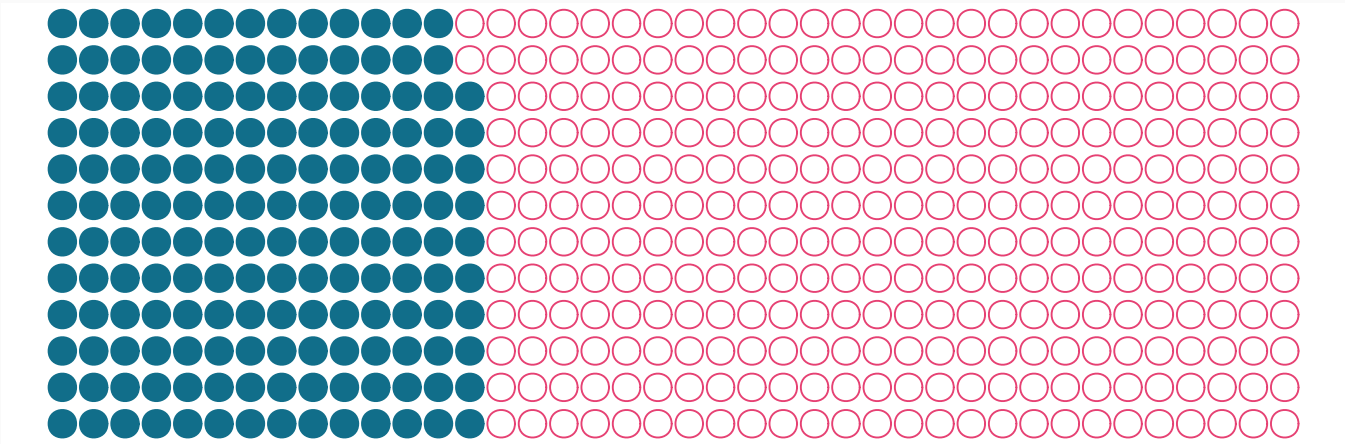


Resampling methods

In Data splitting, we discussed *training* and *test* set. Let's now dive deeper into *resampling* methods.

Validation set (visual inspired by Ed Rubin's course)

- we hold out a subset of the training data (e.g. 30%) and then evaluate the model on this held out validation set
- calculate the loss function on this validation set, as approximation of the true test error
- 👎 high variability + inefficient use of data
- picture **validation set (30%)** and **training set (70%)**

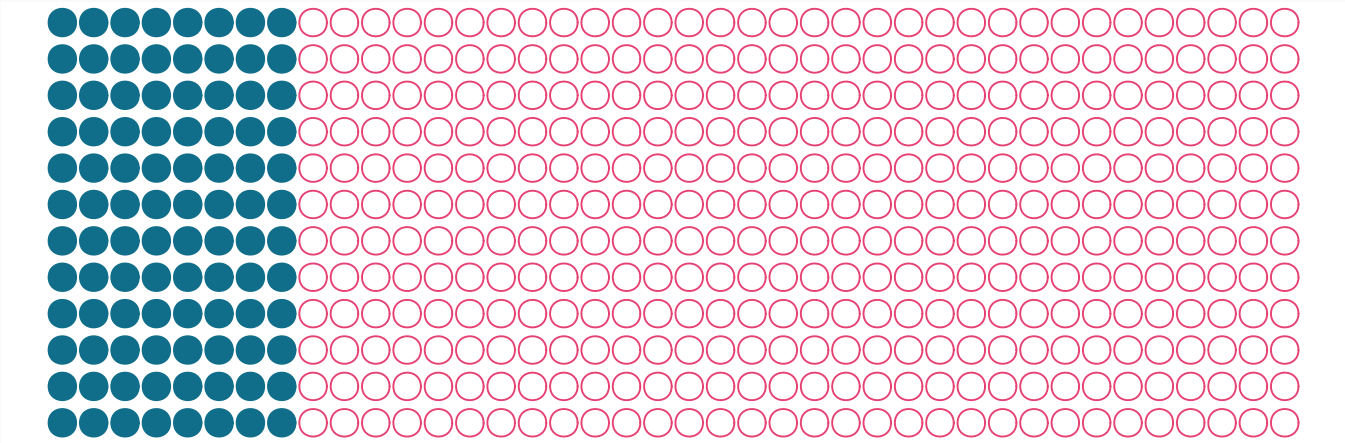


Resampling methods (cont.)

In Data splitting, we discussed *training* and *test* set. Let's now dive deeper into *resampling* methods.

***k* fold cross validation** (visual inspired by Ed Rubin's course)

- divide training data into *k* equally sized groups (e.g. **group 1** on the picture)
- iterate over the *k* groups, treating each as validation set once (and train model on the other *k-1* groups) (e.g. get $\textbf{MSE}_1$ corresponding to fold 1)
- average the folds' loss to estimate the true test error
- 👍 greater accuracy (compared to validation set).

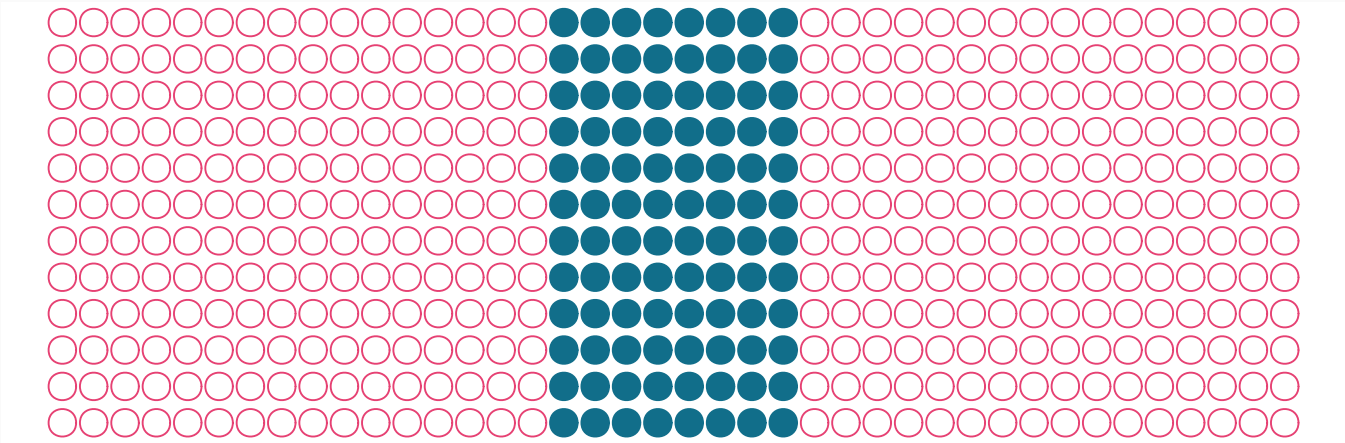


Resampling methods (cont.)

In Data splitting, we discussed *training* and *test* set. Let's now dive deeper into *resampling* methods.

***k* fold cross validation** (visual inspired by Ed Rubin's course)

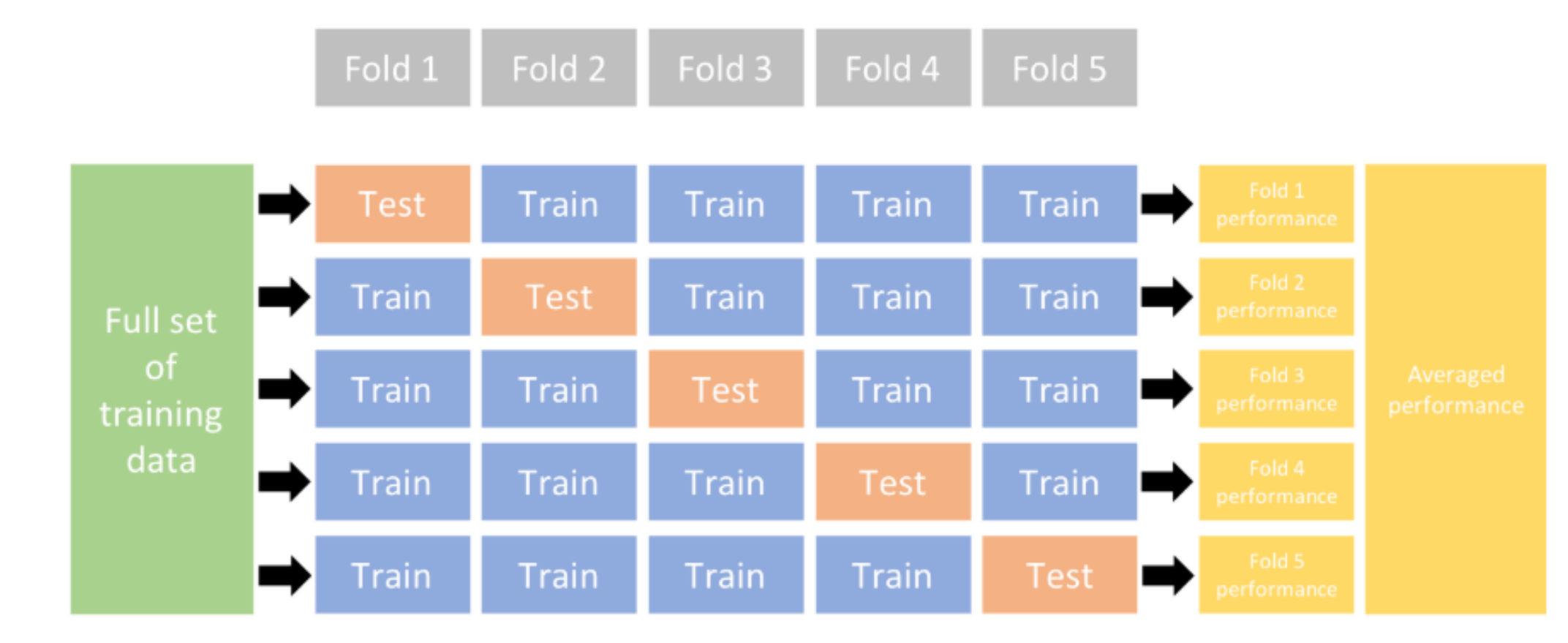
- divide training data into *k* equally sized groups (e.g. **group 1** on the picture)
- iterate over the *k* groups, treating each as validation set once (and train model on the other *k-1* groups) (e.g. get $\textbf{MSE}_1$ corresponding to fold 1)
- average the folds' loss to estimate the true test error
- 👍 greater accuracy (compared to validation set).



Resampling methods (cont.)

In Data splitting, we discussed *training* and *test* set. Let's now dive deeper into *resampling* methods.

***k* fold cross validation** (picture from Boehmke & Greenwell)

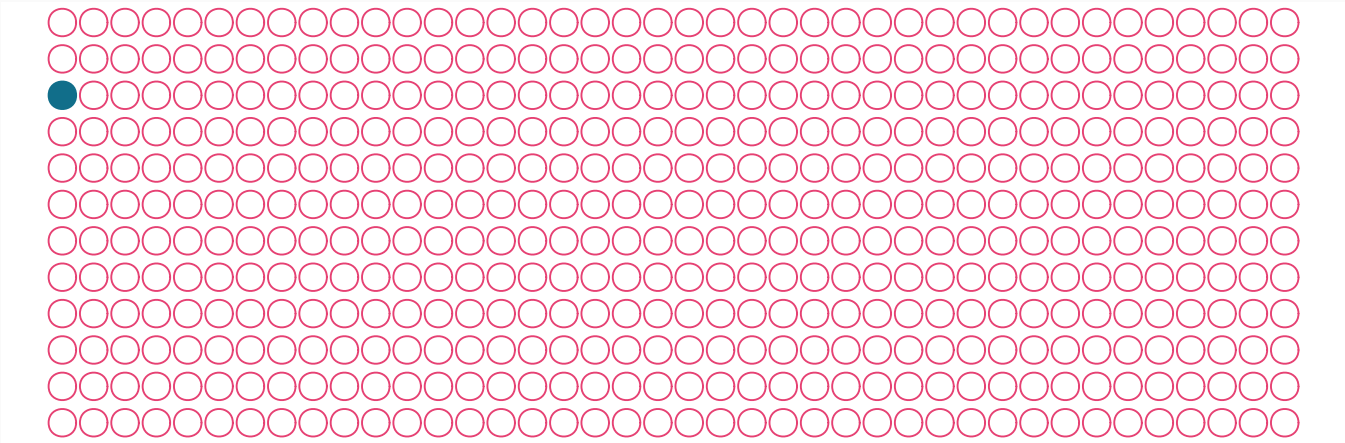


Resampling methods (cont.)

In Data splitting, we discussed *training* and *test* set. Let's now dive deeper into *resampling* methods.

Leave-one-out cross validation (visual inspired by Ed Rubin's course)

- each observation takes a turn as the validation set (e.g. get $\mathbf{MSE_3}$)
- other *n-1* observations are the training set
- average the folds' loss to estimate the true test error
- 👎 very computationally demanding.



Resampling methods in {caret}

We set up 5-fold cross validation using the {caret} package.

```
set.seed(123)
cv_folds ← caret::createFolds(y = ames$Sale_Price,
                              k = 5, list = TRUE,
                              returnTrain = TRUE)
```

```
str(cv_folds)
```

```
## List of 5
##  $ Fold1: int [1:2344] 1 2 3 4 5 6 7 8 9 10 ...
##  $ Fold2: int [1:2343] 2 3 4 6 7 8 9 11 13 14 ...
##  $ Fold3: int [1:2344] 1 2 3 4 5 6 7 8 9 10 ...
##  $ Fold4: int [1:2344] 1 3 5 6 10 11 12 13 14 15 ...
##  $ Fold5: int [1:2345] 1 2 4 5 7 8 9 10 11 12 ...
```

The `createFolds` function from {caret} splits the data into `k` groups.

`list = TRUE` indicates that the results should be stored in a list

`returnTrain = TRUE` indicates that the values returned (and stored) in the elements of the list are - per fold - the row numbers of the observations selected for training.

Resampling methods in {caret}

We set up 5-fold cross validation using the {caret} package.

```
set.seed(123)
cv_folds ← caret::createFolds(y = ames$Sale_Price,
                              k = 5, list = TRUE,
                              returnTrain = TRUE)
```

```
str(cv_folds)
```

```
## List of 5
##  $ Fold1: int [1:2344] 1 2 3 4 5 6 7 8 9 10 ...
##  $ Fold2: int [1:2343] 2 3 4 6 7 8 9 11 13 14 ...
##  $ Fold3: int [1:2344] 1 2 3 4 5 6 7 8 9 10 ...
##  $ Fold4: int [1:2344] 1 3 5 6 10 11 12 13 14 15 ...
##  $ Fold5: int [1:2345] 1 2 4 5 7 8 9 10 11 12 ...
```

Inspect the list `cv_folds` that was returned by `createFolds(.)`.

This list has `k` elements, each storing the row numbers of the observations in the training set of the fold under consideration.

Resampling methods in {caret}


```
mean(ames[cv_folds$Fold1, ]$Sale_Price)
```

```
## [1] 180954
```

```
map_dbl(cv_folds,
        function(x) {
          mean(ames[x, ]$Sale_Price)
        })
```

```
##   Fold1   Fold2   Fold3   Fold4   Fold5
## 180954 180782 180646 180563 181035
```

We calculate the average `Sale_Price` per fold, that is: we average the `Sale_Price` over all observations selected in the training set of a particular fold.

That would go as follows, for `Fold1` in the list `cv_folds`

```
mean(ames[cv_folds$Fold1, ]$Sale_Price)
```

and similarly for `Fold2`, ..., `Fold5`.

Resampling methods in {caret}


```
mean(ames[cv_folds$Fold1, ]$Sale_Price)
```

```
## [1] 180954
```

```
map_dbl(cv_folds,
        function(x) {
          mean(ames[x, ]$Sale_Price)
        })
```

```
##  Fold1  Fold2  Fold3  Fold4  Fold5
## 180954 180782 180646 180563 181035
```

We apply the function `mean(ames[___, ]$Sale_Price)` over all `k` elements of the list `cv_folds`.

`map_dbl(.x, .f)` is one of the `map` functions from the {purrr} package (part of {tidyverse}), used for functional programming in R.

`map_dbl(.x, .f)` applies function `.f` to each element of list `.x`.

The result is a double-precision vector, hence `map_dbl` and not just `map`.

Btw, it is a historical anomaly that R has two names for its floating-point vectors, `double` and `numeric`.

Resampling methods in {rsample}

```
set.seed(123)
cv_rsample ← rsample::vfold_cv(ames, v = 5)
cv_rsample$splits
```

```
## [[1]]
## <Analysis/Assess/Total>
## <2344/586/2930>
##
## [[2]]
## <Analysis/Assess/Total>
## <2344/586/2930>
##
## [[3]]
## <Analysis/Assess/Total>
## <2344/586/2930>
##
## [[4]]
## <Analysis/Assess/Total>
## <2344/586/2930>
##
## [[5]]
## <Analysis/Assess/Total>
```

The function `vfold_cv` splits the data into `v` groups (called folds) of equal size.

Resampling methods in {rsample}


```
set.seed(123)
cv_rsample ← rsample::vfold_cv(ames, v = 5)
cv_rsample$splits
```

```
## [[1]]
## <Analysis/Assess/Total>
## <2344/586/2930>
##
## [[2]]
## <Analysis/Assess/Total>
## <2344/586/2930>
##
## [[3]]
## <Analysis/Assess/Total>
## <2344/586/2930>
##
## [[4]]
## <Analysis/Assess/Total>
## <2344/586/2930>
##
## [[5]]
## <Analysis/Assess/Total>
```

The function `vfold_cv` splits the data into `v` groups (called folds) of equal size.

We store the result of `vfold_cv` in the object `cv_rsample`.

The resulting object stores `v` resamples of the original data set.

Resampling methods in {rsample}


```
set.seed(123)
cv_rsample ← rsample::vfold_cv(ames, v = 5)
```

```
cv_rsample$splits[[1]]
```

```
## <Analysis/Assess/Total>
## <2344/586/2930>
```

```
cv_rsample$splits[[1]] %>% analysis() %>% dim()
```

```
## [1] 2344   81
```

```
cv_rsample$splits[[1]] %>% assessment() %>% dim()
```

```
## [1] 586  81
```

Inspect the composition of the first resample:

2,344 (out of 2,930) observations go to the analysis data (for training, i.e. `v-1` folds),

586 (out of 2,930) observations go to the assessment data (for testing, the final fold).

Resampling methods in {rsample}


```
set.seed(123)
cv_rsample ← rsample::vfold_cv(ames, v = 5)
```

```
cv_rsample$splits[[1]]
```

```
## <Analysis/Assess/Total>
## <2344/586/2930>
```

```
cv_rsample$splits[[1]] %>% analysis() %>% dim()
```

```
## [1] 2344   81
```

```
cv_rsample$splits[[1]] %>% assessment() %>% dim()
```

```
## [1] 586  81
```

Inspect the composition of the first resample:

get the dimensions (`dim()`) of the analysis data (`analysis()`) of the first resample

get the dimensions (`dim()`) of the assessment data (`assessment()`) of the first resample.

Resampling methods in {rsample}


```
map_dbl(cv_rsample$splits,
        function(x) {
          mean(rsample::analysis(x)$Sale_Price)
        })
```

```
## [1] 181311 180991 180840 181269 179570
```

```
map_dbl(cv_rsample$splits,
        function(x) {
          nrow(rsample::analysis(x))
        })
```

```
## [1] 2344 2344 2344 2344 2344
```

As before, use `map_dbl(.x, .f)` to apply a function `.f` over all elements of a list `.x`.

Here the list is stored in `cv_rsample$splits`, with `v = 5` elements.

Your turn

Q: Now you're going to combine data splitting and resampling to create training, validation and test folds in the Ames data.

Use `caret` or `rsample` and make the validation folds of the same size as the test fold.

with `caret`

```
set.seed(5678)
ind_caret ← caret::createDataPartition(
                    y = ames$Sale_Price,
                    p = 5/6, list = FALSE)
train_caret ← ames[ind_caret, ]
test_caret  ← ames[-ind_caret, ]

cv_caret ← caret::createFolds(
            y = train_caret$Sale_Price, k = 5,
            list = TRUE, returnTrain = FALSE)

purrr::map_dbl(cv_caret,
               ~ nrow(ames[., ]))
## Fold1 Fold2 Fold3 Fold4 Fold5
##   488   488   489   489   489
nrow(test_caret)
## [1] 487
```

with `rsample`

```
set.seed(5678)
ind_rsample  ← rsample::initial_split(ames,
                                      prop = 5/6)
train_rsample  ← rsample::training(ind_rsample)
test_rsample   ← rsample::testing(ind_rsample)

cv_rsample ← rsample::vfold_cv(train_rsample, v = 5)

map_dbl(cv_rsample$splits,
        ~ nrow(rsample::assessment(.)))
## [1] 489 489 488 488 488
nrow(test_rsample)
## [1] 488
```

Parameter tuning with {caret}, {rsample} and {purrr}

Tuning parameters

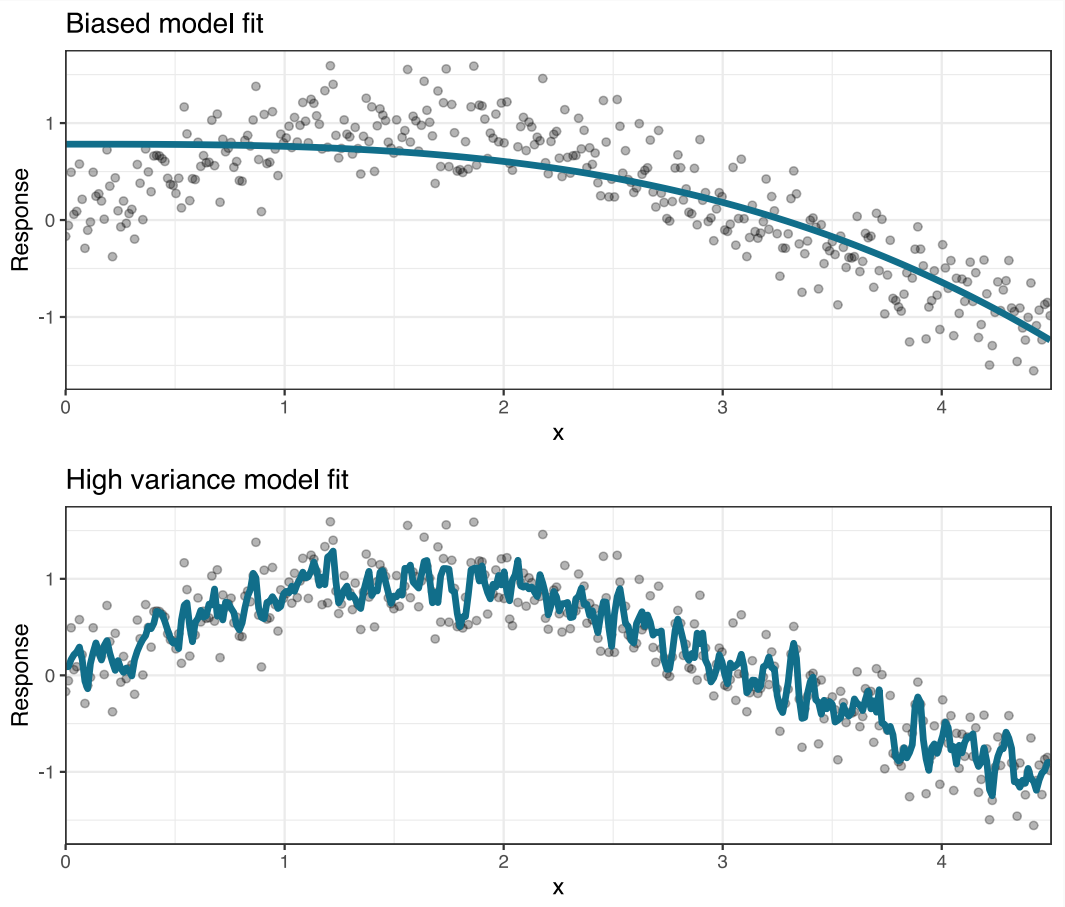
Finding the optimal level of flexibility highlights the bias-variance tradeoff.

Bias : the error that comes from inaccurately estimating f.

Variance : the amount $\hat{f}$ would change with a different training sample.

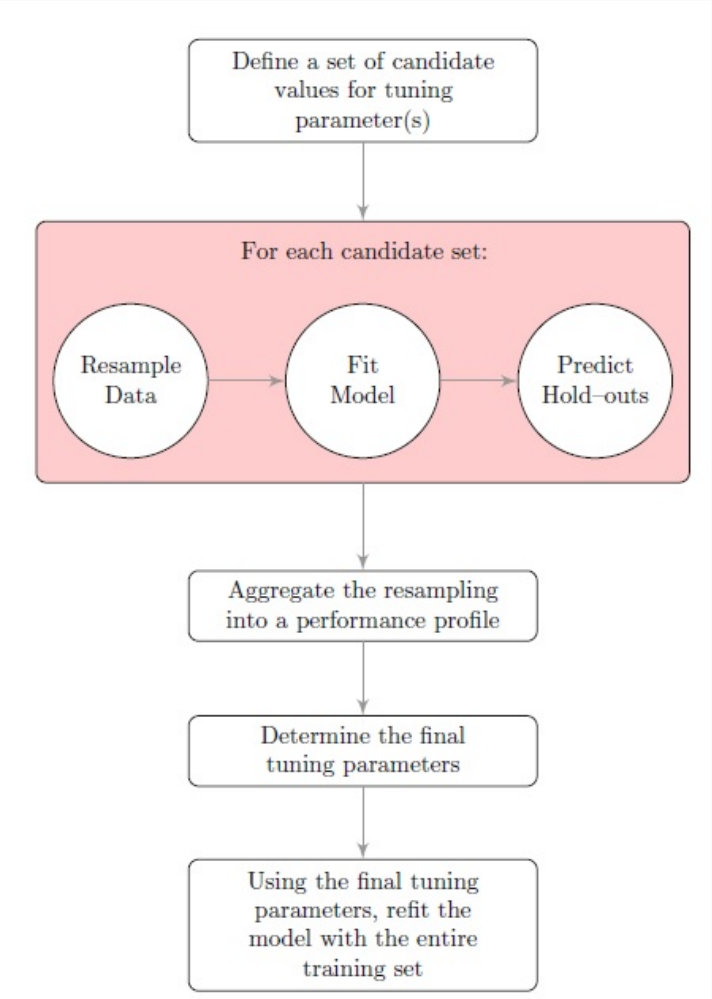
Take-aways 📣 : high variance models more prone to overfitting

- use **resampling methods** to reduce this risk
- hyperparameters (or *tuning parameters*) control complexity, and thus the bias-variance trade-off
- identify their optimal setting, e.g. with a *grid search*
- no analytic expression for these hyperparameters.



Code from Boehmke & Greenwell (2019, Chapter 2) on Hands-on machine learning with R.

Tuning parameters via grid search



Model training & validation phase

- define a set of candidate values (a *grid*)
- assess model utility across the candidates (use clever *resampling*)
- choose the optimal settings (optimize *loss*)
- refit the model on entire training data with final tuning parameters
- evaluate performance of the model on the test data (under 🔒).

Model selection

- repeat the above steps for different models
- compare performance of these models that will generalize to new data (via test data, under 🔒).

Flow chart from Kuhn & Johnson (2013) on Applied predictive modeling.

Training a model with {caret}

```
set.seed(123)
cv ← trainControl(method = "cv", number = 5,
                  returnResamp = "all",
                  selectionFunction = "best")
hyper_grid ← expand.grid(k = seq(2, 150, by = 2))
knn_fit ← train(y ~ x, data = df, method = "knn",
                        trControl = cv,
                        tuneGrid = hyper_grid)
knn_fit$bestTune
```

Use `trainControl` from {caret} to set some control parameters that will be used in the actual `train` function.

Here, we use `method = cv` and `number = 5` for 5-fold cross validation.

Training a model with {caret}

```
set.seed(123)
cv ← trainControl(method = "cv", number = 5,
                  returnResamp = "all",
                  selectionFunction = "best")
hyper_grid ← expand.grid(k = seq(2, 150, by = 2))
knn_fit ← train(y ~ x, data = df, method = "knn",
                        trControl = cv,
                        tuneGrid = hyper_grid)
knn_fit$bestTune
```

In `trainControl` we put `returnResamp = "all"` to store all resampled summary metrics.

`selectionFunction = "best"` specifies how we select the optimal tuning parameter. With `"best"` the value that minimizes the performance (here: RMSE) is selected.

Alternative: `selectionFunction = "oneSE"` applies the one standard error rule.

Training a model with {caret}

```
set.seed(123)
cv ← trainControl(method = "cv", number = 5,
                  returnResamp = "all",
                  selectionFunction = "best")
hyper_grid ← expand.grid(k = seq(2, 150, by = 2))
knn_fit ← train(y ~ x, data = df, method = "knn",
                       trControl = cv,
                       tuneGrid = hyper_grid)
knn_fit$bestTune
```

Set the grid of *K*-values that will be searched.

`expand.grid` creates a data frame with one row for each value of *K* to consider.

Training a model with {caret}

```
set.seed(123)
cv ← trainControl(method = "cv", number = 5,
                  returnResamp = "all",
                  selectionFunction = "best")
hyper_grid ← expand.grid(k = seq(2, 150, by = 2))
knn_fit ← train(y ~ x, data = df, method = "knn",
                        trControl = cv,
                        tuneGrid = hyper_grid)
knn_fit$bestTune
```

{caret} will `train` the method `knn` using the settings in `trControl = cv`, across the values of *K* stored in `tuneGrid = hyper_grid`.

The data `df` and formula `y ~ x` are used.

Training a model with {caret}

```
set.seed(123)
cv ← trainControl(method = "cv", number = 5,
                  returnResamp = "all",
                  selectionFunction = "best")
hyper_grid ← expand.grid(k = seq(2, 150, by = 2))
knn_fit ← train(y ~ x, data = df, method = "knn",
                        trControl = cv,
                        tuneGrid = hyper_grid)
knn_fit$bestTune
```

```
##     k
## 18 36
```

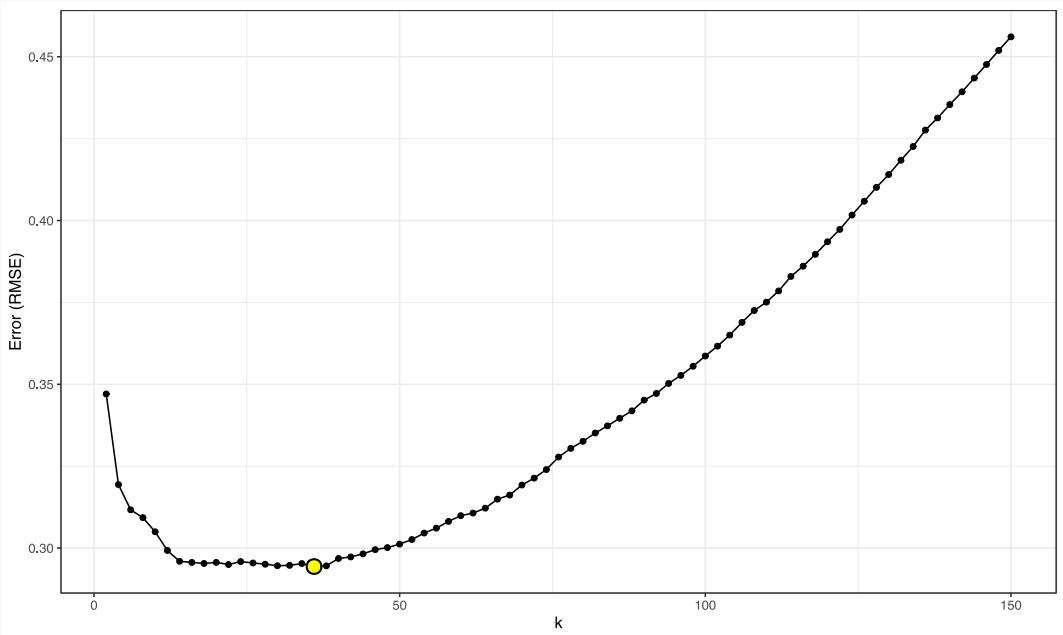
We retrieve the optimal value of the tuning parameter, according to the `selectionFunction`.

For the folds created here and with `selectionFunction = "best"` the optimal *K* value is 36.

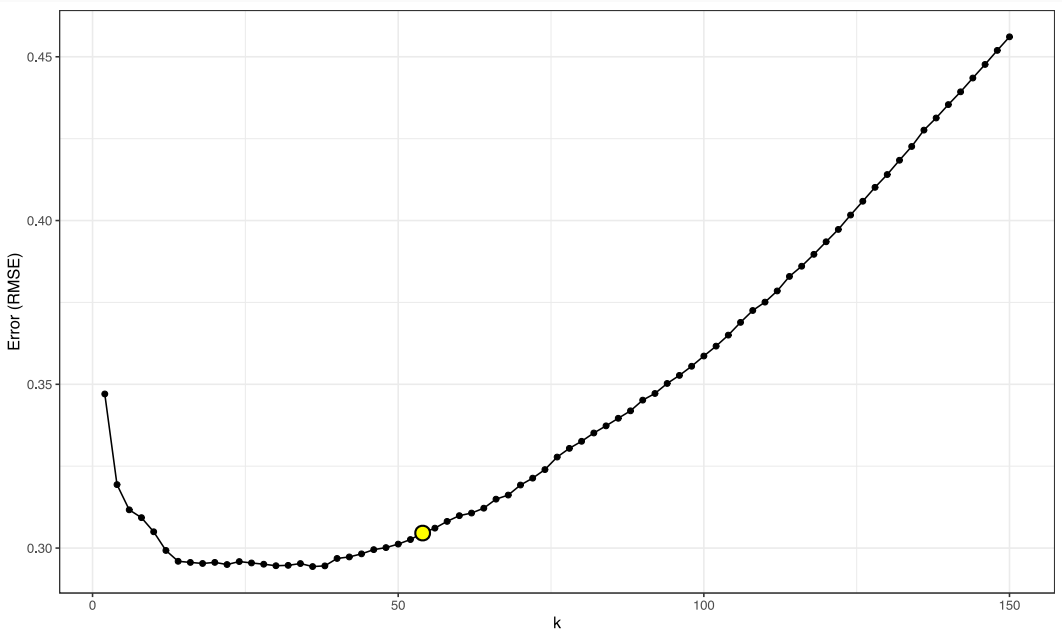
What happens when you change to `selectionFunction = "oneSE"`?

Training a model with {caret}

```
##      k
## 18 36
```




```
##      k
## 27 54
```



Training a model with {rsample}

Our starting point is the simulated data stored in `df`, resampled with 5-fold cross-validation.

```
set.seed(123)  # for reproducibility
cv_rsample ← vfold_cv(df, 5)
cv_rsample$splits[1:3]
```

```
## [[1]]
## <Analysis/Assess/Total>
## <286/72/358>
##
## [[2]]
## <Analysis/Assess/Total>
## <286/72/358>
##
## [[3]]
## <Analysis/Assess/Total>
## <286/72/358>
```

We fit the *KNN* on the holdout data in split *s*, using a given *K* value.

```
holdout_results ← function(s, k_val) {
  # Fit the model to the analysis data in split s
  df_train ← analysis(s)
  mod ← knnreg(y ~ x, k = k_val, data = df_train)
  # Get the remaining group
  holdout ← assessment(s)
  # Get predictions with the holdout data set
  res ← predict(mod, newdata = holdout)
  # Return observed and predicted values
  #                                on holdout set
  res ← tibble(obs = holdout$y, pred = res)
  res
}
```

Your turn

Now you're going to combine the **resampling and model fitting instructions** and set up a first example of **tuning a parameter** over a grid of possible values: the *K* in a **KNN regression model**.

Q: use the function `holdout_results(.s, .k)` as defined on the previous sheet. You will use this function to calculate the RMSE_k of fold *k*.

1. Specify a grid of values of *K*, store it in `hyper_grid`. Use `expand.grid(.)`
2. Pick one of the resamples stored in `cv_rsample$splits` and pick a value from the grid. Calculate the RMSE on the holdout data of this split.
3. For all values in the tuning grid, calculate the RMSE averaged over all folds, and the corresponding standard error.
4. Use the results from **Q.3** to pick the value of *K* via minimal RMSE.
5. Pick the largest value of *K* such that the corresponding RMSE is below the minimal RMSE from **Q.4** plus its corresponding SE.

Q.1 We set up the grid

```
hyper_grid ← expand.grid(k = seq(2, 150, by = 2))
hyper_grid %>% slice(1:3)
```

k
2
4
6

Q.2 We apply the function `holdout_results(.s, .k)` on the third resample, with the first value for *K* in the grid.

```
res ← holdout_results(cv_rsample$splits[[3]],
                      hyper_grid[1, ])
sqrt(sum((res$obs - res$pred)^2)/nrow(res))
```

```
## [1] 0.3609
```

Q.3 Mean RMSE over the 5 folds and corresponding SE.

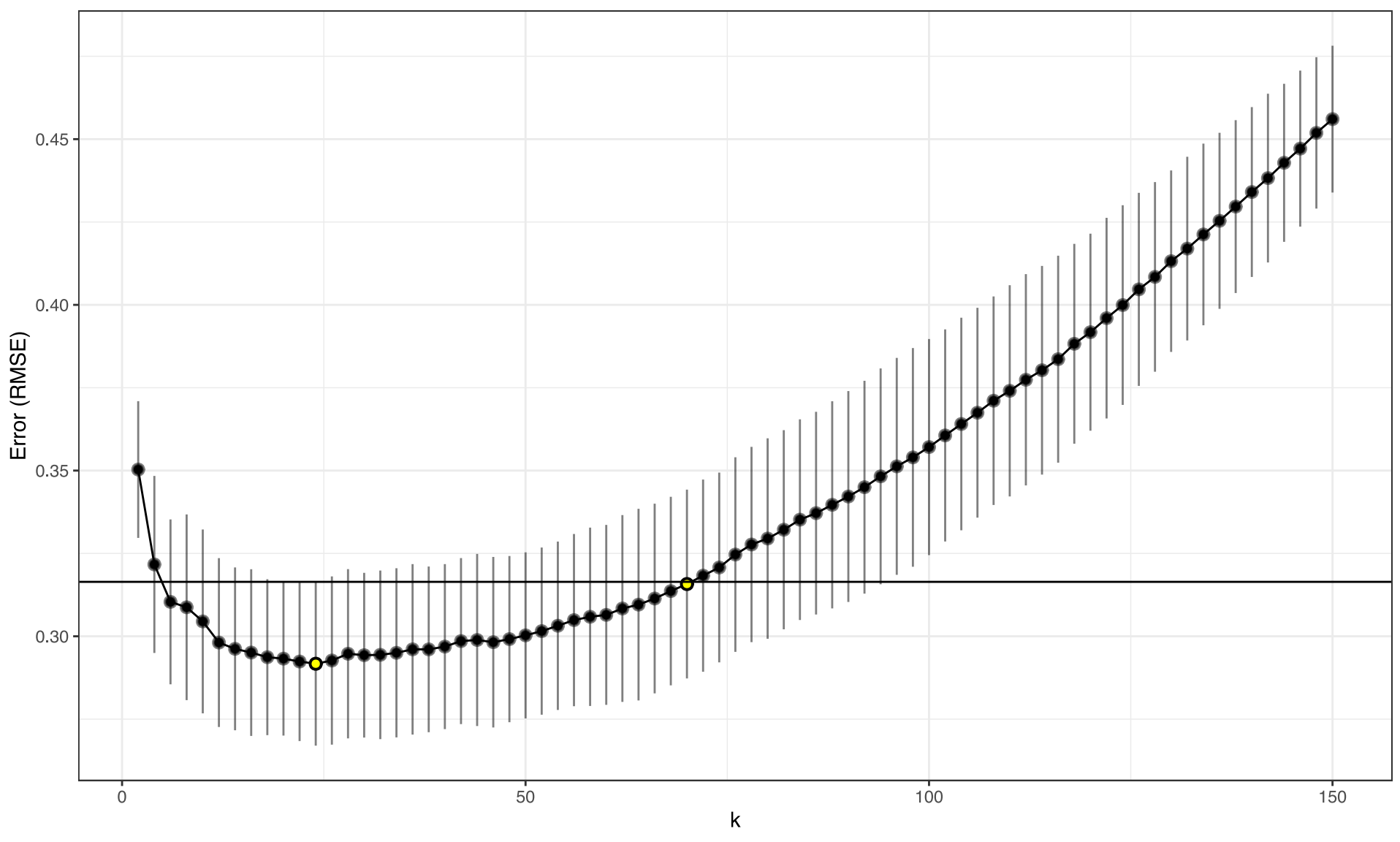
```
RMSE ← numeric(nrow(hyper_grid))
SE ← numeric(nrow(hyper_grid))
for(i in 1:nrow(hyper_grid)){
  cv_rsample$results ← map(cv_rsample$splits,
                           holdout_results,
                           hyper_grid[i, ])
  res ← map_dbl(cv_rsample$results,
                function(x) mean((x$obs - x$pred)^2))
  RMSE[i] ← mean(sqrt(res)) ; SE[i] ← sd(sqrt(res))
}
```

Q.4 Choose *K* via minimal RMSE

RMSE	SE	k	lower	upper
0.2917	0.0247	24	0.267	0.3164

Q.5 Choose *K* via the one-standard-error rule

RMSE	SE	k	lower	upper
0.3158	0.0285	70	0.2873	0.3442


0.45
0.40
0.35
0.30
Error (RMSE)
0
50
100
150
k

Putting it all together

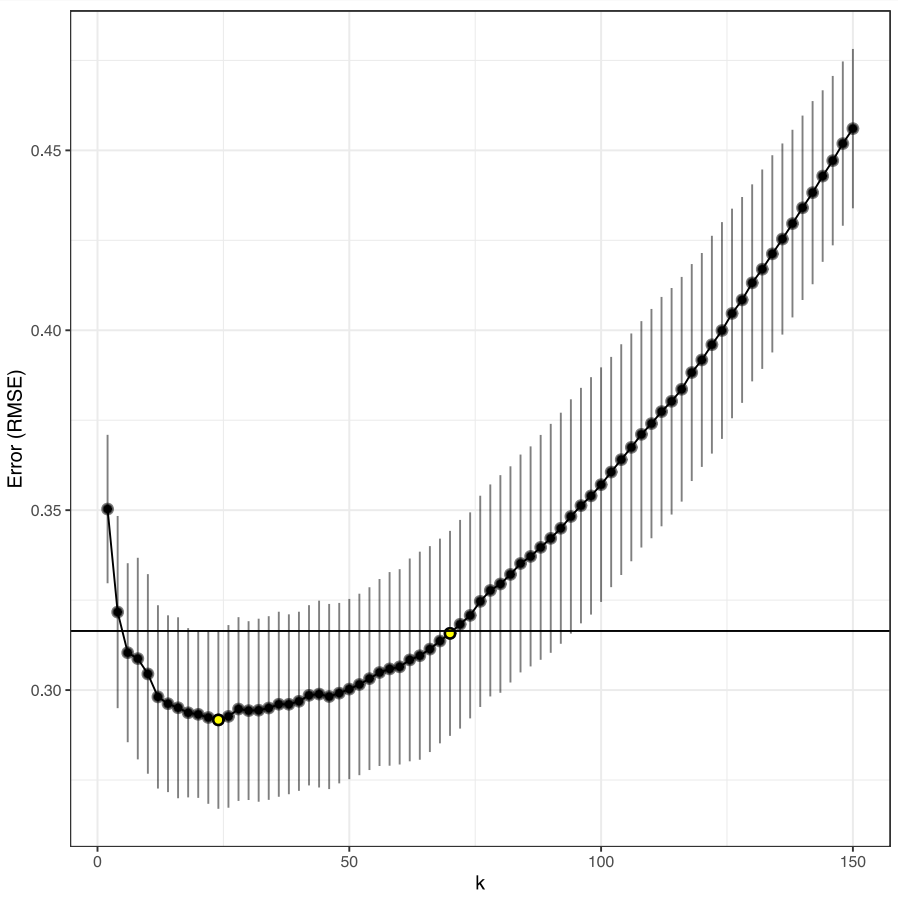
During the tuning process we inspect plots like the one on the right.

Take-aways 📣

Less is more:

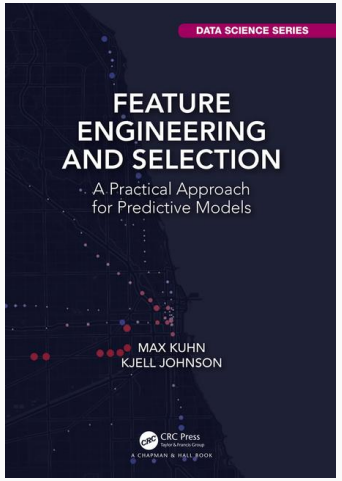
- we prefer simple over more complex
- choose tuning parameters based on the numerically optimal value OR
- choose a simpler model that is within a certain tolerance of the numerically best value
- use the **'one-standard-error' rule**.

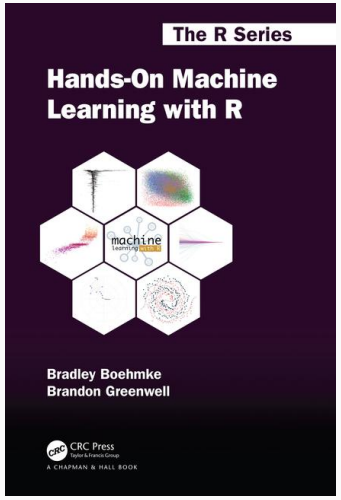
With the selected tuning parameters, we refit the model on the complete training set and use it to predict the test set (under 🔒).



Target and feature engineering: data pre-processing steps

What is feature engineering?





Feature engineering:

- applies **pre-processing steps** to predictor (features) variables
- **creates new input features** from your existing ones (e.g. network features derived from a social network in a fraud detection model).

Target engineering:

- transforms the response variable (or target) to improve the performance of a predictive model.

The goal is to make models more effective.

See Kuhn & Johnson (2019) on Feature Engineering and Selection: A Practical Approach for Predictive Models for a detailed discussion.

Take-aways 📢: *different models* have *different sensitivities* to the type of target and feature values in the model.

Table A.1: A summary of models and some of their characteristics

Model	Allows $n < p$	Pre-processing	Interpretable	Automatic feature selection	# Tuning parameters	Robust to predictor noise	Computation time
Linear regression†	×	CS, NZV, Corr	✓	×	0	×	✓
Partial least squares	✓	CS	✓	○	1	×	✓
Ridge regression	×	CS, NZV	✓	×	1	×	✓
Elastic net/lasso	×	CS, NZV	✓	✓	1–2	×	✓
Neural networks	✓	CS, NZV, Corr	×	×	2	×	×
Support vector machines	✓	CS	×	×	1–3	×	×
MARS/FDA	✓		○	✓	1–2	○	○
K-nearest neighbors	✓	CS, NZV	×	×	1	○	✓
Single trees	✓		○	✓	1	✓	✓
Model trees/rules†	✓		○	✓	1–2	✓	✓
Bagged trees	✓		×	✓	0	✓	○
Random forest	✓		×	○	0–1	✓	×
Boosted trees	✓		×	✓	3	✓	×
Cubist†	✓		×	○	2	✓	×
Logistic regression*	×	CS, NZV, Corr	✓	×	0	×	✓
{LQRM}DA*	×	NZV	○	×	0–2	×	✓
Nearest shrunken centroids*	✓	NZV	○	✓	1	×	✓
Naïve Bayes*	✓	NZV	×	×	0–1	○	○
C5.0*	✓		○	✓	0–3	✓	×

†regression only *classification only

Symbols represent affirmative (✓), negative (×), and somewhere in between (○)

Source: Kuhn & Johnson (2013) on Applied predictive modeling.

Target engineering

We load the `ames` data set from the {AmesHousing} package and apply a **stratified split** of the data into a training (70%) and test (30%) set.

We stratify on the distribution of the target variable `Sale_Price` using the `strata` argument in `rsample::initial_split`.

```
ames ← AmesHousing::make_ames()
set.seed(123)
split  ← rsample::initial_split(ames, prop = 0.7,
                                strata = "Sale_Price")
ames_train  ← rsample::training(split)
ames_test   ← rsample::testing(split)
```

We check the distribution of `Sale_Price` in both `ames_train` and `ames_test`.

```
summary(ames_train$Sale_Price)
summary(ames_test$Sale_Price)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   13100  129500  160000  180996  213500  755000
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   12789  129500  160000  180327  213500  625000
```

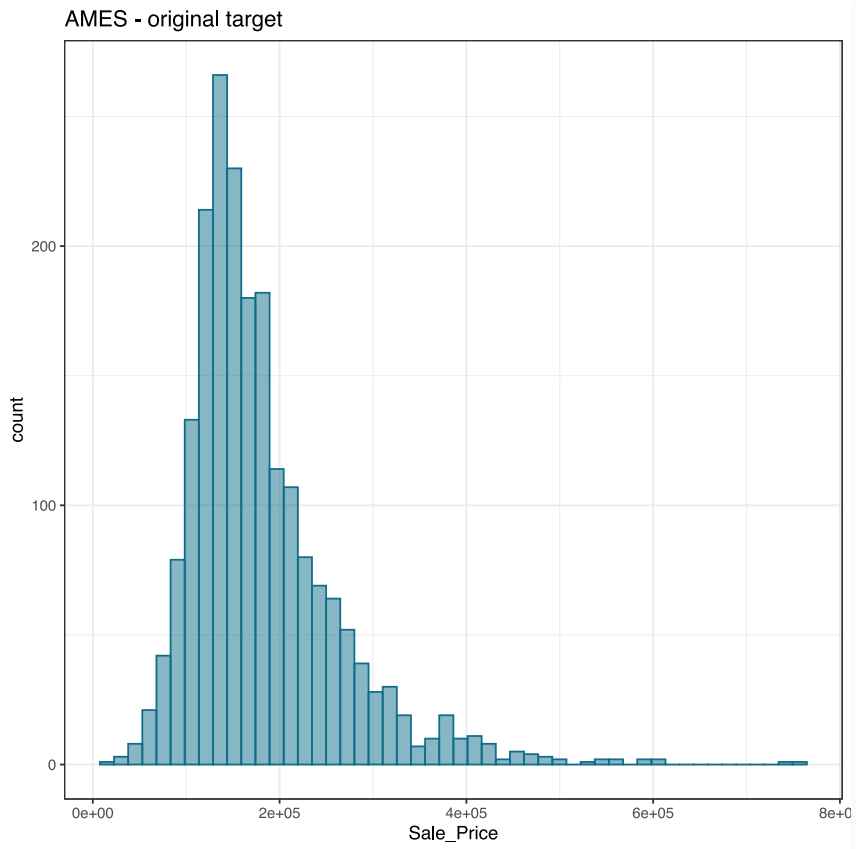
Your turn

Inference with linear models often assumes that the target is generated from a normal distribution.

Q: let's examine whether the `Sale_Price` target satisfies this assumption.

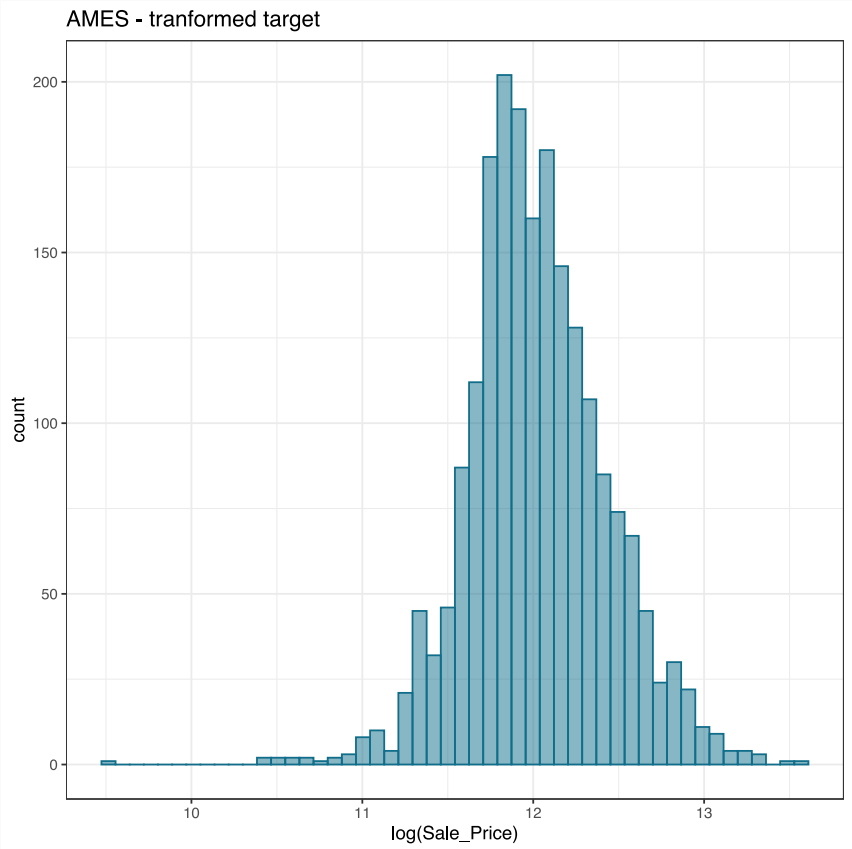
1. Plot a histogram of `Sale_Price`. Is normality a meaningful assumption?
2. Try some transformation functions such that the transformed target approaches a normal distribution.

Q.1 original target




```
summary(ames_train$Sale_Price)
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   13100  129500  160000  180996  213500  755000
```

Q.2 log-transformed target




```
summary(log(ames_train$Sale_Price))
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    9.48   11.77   11.98   12.02   12.27   13.53
```

Feature engineering steps

Examples of common pre-processing steps:

- Some models (e.g. KNN, Lasso, neural networks) require that the predictor variables are on the same scale. **Centering (C)** and **scaling (S)** the predictors can be used for this purpose.
- Other models are very sensitive to correlations between the predictors and filters or PCA signal extraction can improve the model.
- Some models find **(near) zero-variance (NZV)** predictors problematic, and these should be removed before fitting the model.
- In other cases, the data should be **encoded** in a specific way to make sure all predictors are numeric (e.g. one-hot encoding of factor variables in neural networks).
- Many models cannot cope with **missing data** so **imputation strategies** might be necessary.
- Development of new features that represent something important to the outcome.
- (add your own example here!)

This list is inspired by Max Kuhn (2019) on Applied Machine Learning.

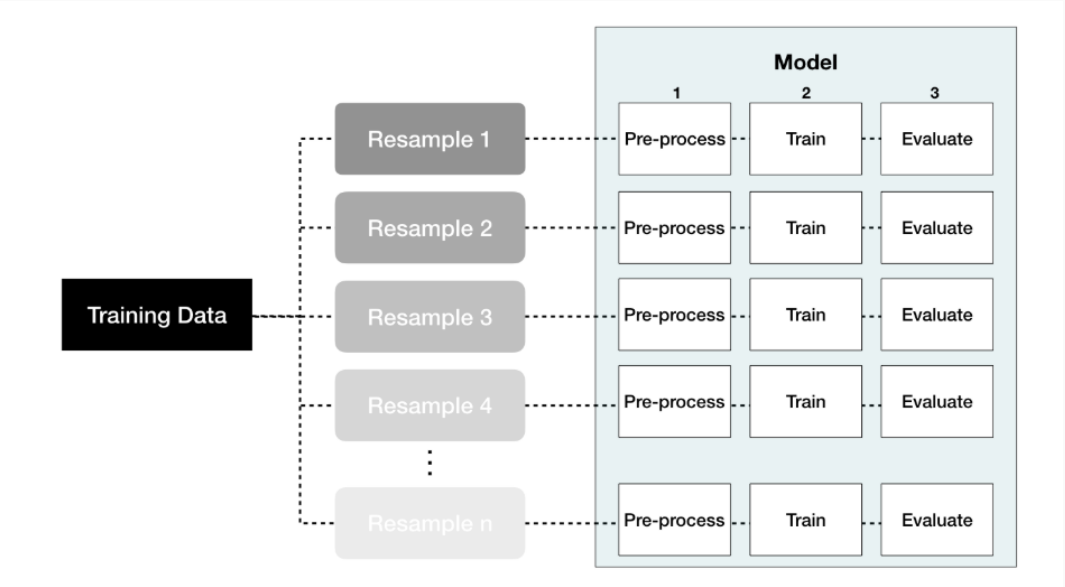
A blueprint for feature engineering

Take-aways 📣 : *a proper implementation*

- draft a **blueprint** of the necessary pre-processing steps, and their order
- Boehme & Greenwell (2019) suggest

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.

- avoid **data leakage** in the pre-processing steps when applied to resampled data sets!



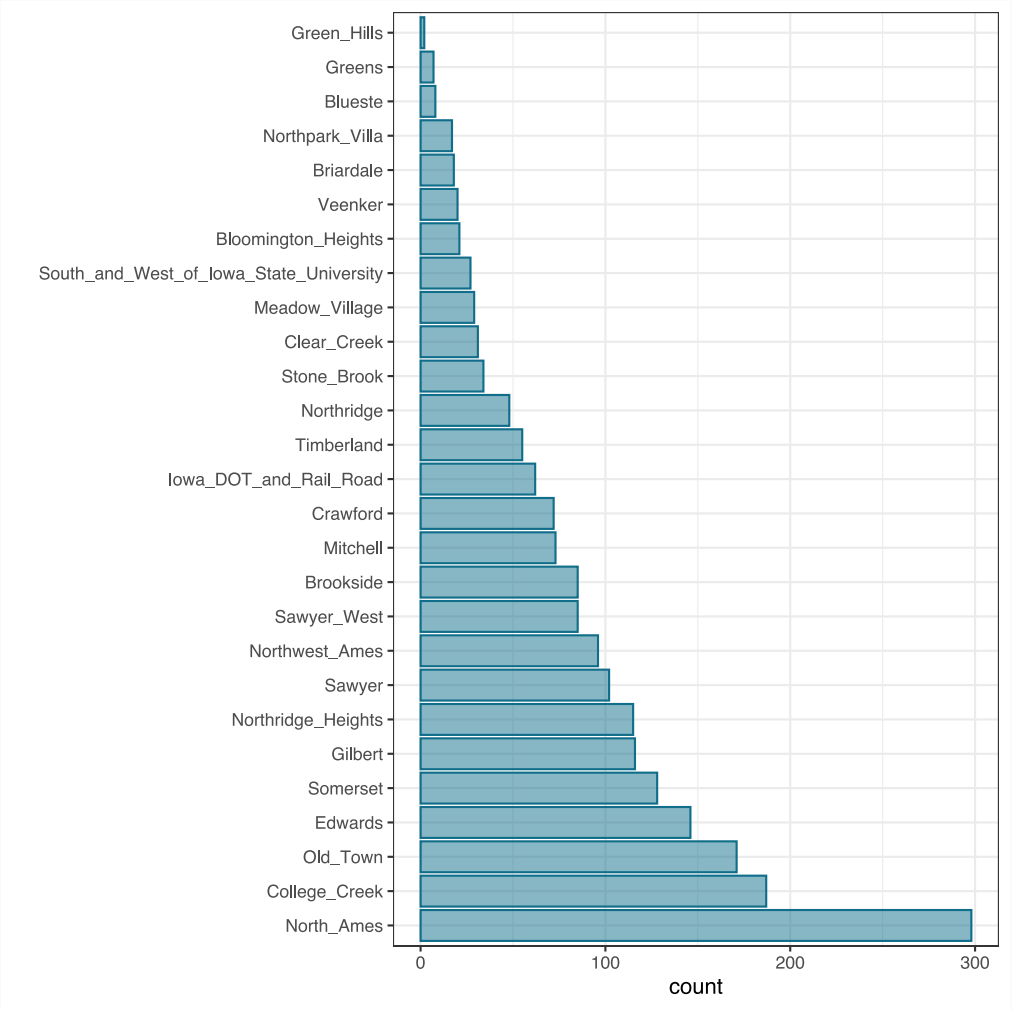
Feature engineering with {recipes}

We already detected the necessity of log-transforming `Sale_Price` when building linear models.

We add another pre-processing step, inspired by the **high cardinality** feature `Neighborhood`.

```
ames_train %>% group_by(Neighborhood) %>%
  summarize(n_obs = n()) %>%
  arrange(n_obs) %>% slice(1:4)
```

Neighborhood	n_obs
Green_Hills	2
Greens	7
Blueste	8
Northpark_Villa	17



Feature engineering with {recipes}

We'll use `recipe()` from the {recipes} package.

The main idea is to **preprocess multiple datasets** using a single `recipe()`.

Before we start, keep the following fundamentals of {recipes} in mind!

Creating a `recipe` takes the following steps:

- get the *ingredients* (`recipe()`): specify the response and predictor variables
- write the recipe (`step_zzz()`): define the *pre-processing steps*, such as imputation, creating dummy variables, scaling, and more
- *prepare* the recipe (`prep()`): provide a dataset to base each step on (e.g. *calculate* constants to do centering and scaling)
- *bake* the recipe (`bake()`): *apply* the pre-processing steps to your datasets.

Source: Rebecca Barter's blog

Feature engineering with {recipes}

Use `recipe()` to create the preprocessing blueprint (to be applied later)

```
library(recipes)
mod_rec ← recipe(Sale_Price ~ ., data = ames_train)
mod_rec
```

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

Now, `mod_rec` knows the role of each variable (`predictor` or `outcome`).

We can use selectors such as `all_predictors()`, `all_outcomes()` or `all_nominal()`.

Extend `mod_rec` with two pre-processing steps:

`step_log(all_outcomes())`

`step_other(Neighborhood, threshold = 0.05)` to lump the levels that occur in less than 5% of data as "other".

```
mod_rec ← mod_rec %>% step_log(all_outcomes()) %>%
            step_other(Neighborhood, threshold = 0.05)
mod_rec
## Data Recipe
##
## Inputs:
##
##       role #variables
##    outcome           1
##  predictor          80
##
## Operations:
##
## Log transformation on all_outcomes()
## Collapsing factor levels for Neighborhood
```

Feature engineering with {recipes}


```
recipe  -->     prep    --> bake/juice

(define) --> (calculate) -->  (apply)
```

Now that we have a preprocessing *specification*, we run on it on the `ames_train` to *prepare* (or `prep()`) the recipe.

```
mod_rec_trained ← prep(mod_rec, training = ames_train, verbose = TRUE, retain = TRUE)
```

```
mod_rec_trained ← prep(mod_rec, training = ames_train, verbose = TRUE, retain = TRUE)
## oper 1 step log [training]
## oper 2 step other [training]
## The retained training set is ~ 0.82 Mb  in memory.
```

The `retain = TRUE` indicates that the preprocessed training set should be saved.

Source Max Kuhn (2019) on Applied Machine Learning.

Feature engineering with {recipes}


```
mod_rec_trained
```

```
## Data Recipe
##
## Inputs:
##
##       role #variables
##    outcome           1
##  predictor          80
##
## Training data contained 2053 data points and no missing data.
##
## Operations:
##
## Log transformation on Sale_Price [trained]
## Collapsing factor levels for Neighborhood [trained]
```

Once the recipe is prepared, it can be applied to any data set using `bake()`. There is no need to `bake()` the data used in the `prep()` step; you get the processed training set with `juice()`.

```
ames_test_prep ← bake(mod_rec_trained, new_data = ames_test)
```

Feature engineering with {recipes}

```
ames_test_prep %>% group_by(Neighborhood) %>%
  summarize(n_obs = n()) %>%
  arrange(n_obs)
```

Neighborhood	n_obs
Edwards	48
Gilbert	49
Northridge_Heights	51
Somerset	54
Old_Town	68
College_Creek	80
North_Ames	145
other	382

```
juice(mod_rec_trained) %>% group_by(Neighborhood) %>%
  summarize(n_obs = n()) %>%
  arrange(n_obs)
```

Neighborhood	n_obs
Northridge_Heights	115
Gilbert	116
Somerset	128
Edwards	146
Old_Town	171
College_Creek	187
North_Ames	298
other	892

Your turn

Now you will extend the existing recipe in `mod_rec`, prepare and bake it again!

Q: consult the {recipes} manual and specify a recipe for the housing data that includes the following pre-processing steps (in this order)

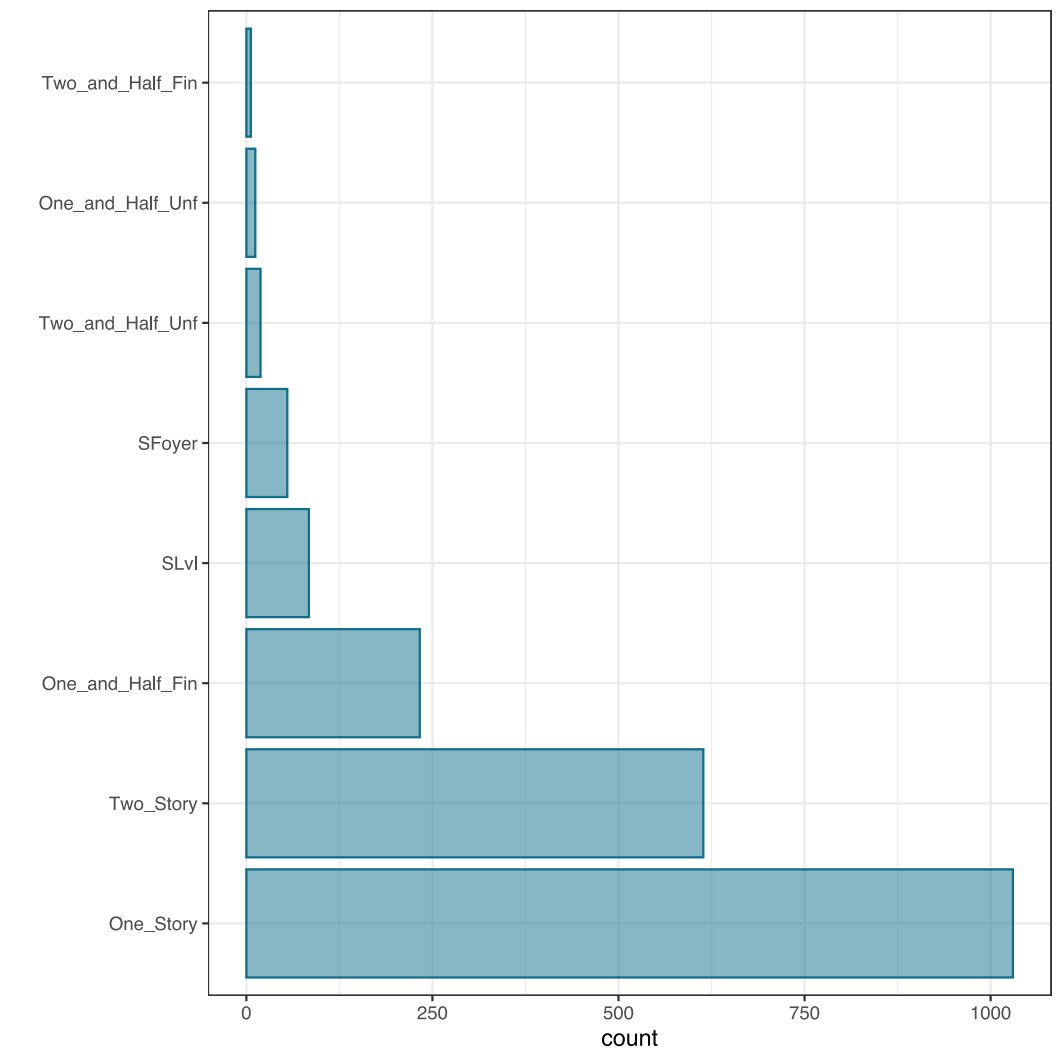
- log-transform the outcome variable
- remove any zero-variance predictors
- lump factor levels that occur in <= 5% of data as "other" for both `Neighborhood` as well as `House_Style`
- center and scale all numeric features.

1. Specify the above recipe on the training set and store it in the object `mod_rec`.
2. Inspect the object `mod_rec` using `summary(mod_rec)`. What can you learn from this summary?
3. Prepare the recipe on the training data and then apply it to the test set.

First, let's try to get a grasp of the `House_Style` feature as well as the presence of zero-variance predictors.

```
ames_train %>% group_by(House_Style) %>%
               summarize(n_obs = n()) %>%
                arrange(n_obs)
```

House_Style	n_obs
Two_and_Half_Fin	6
One_and_Half_Unf	12
Two_and_Half_Unf	19
SFoyer	55
SLvl	84
One_and_Half_Fin	233
Two_Story	614
One_Story	1030



To detect the presence of zero-variance and near-zero-variance features the `caret` library has the function `nearZeroVar`

```
library(caret)
nzv ← caret::nearZeroVar(ames_train, saveMetrics = TRUE)
```

```
names(ames_train)[nzv$zeroVar]
```

```
## character(0)
```

```
names(ames_train)[nzv$nzv]
```

```
##  [1] "Street"            "Alley"             "Land_Contour"      "Utilities"         "Land_Slope"        "Con
##  [7] "Roof_Matl"         "Bsmt_Cond"         "BsmtFin_Type_2"    "Heating"           "Low_Qual_Fin_SF"   "Kit
## [13] "Functional"        "Enclosed_Porch"    "Three_season_porch" "Screen_Porch"     "Pool_Area"         "Poo
## [19] "Misc_Feature"      "Misc_Val"
```

So, no features have zero- variance, but 20 features have near-zero-variance.

We put the recipe together with the following steps

```
mod_rec ← recipe(Sale_Price ~ ., data = ames_train) %>
        step_log(all_outcomes()) %>%
        step_other(Neighborhood, threshold = 0.05) %>%
        step_other(House_Style, threshold = 0.05) %>%
        step_zv(all_predictors()) %>%
        step_nzv(all_predictors()) %>%
        step_center(all_numeric(), -all_outcomes()) %>%
        step_scale(all_numeric(), -all_outcomes())
summary(mod_rec) %>% slice(1:6)
```

variable	type	role	source
MS_SubClass	nominal	predictor	original
MS_Zoning	nominal	predictor	original
Lot_Frontage	numeric	predictor	original
Lot_Area	numeric	predictor	original
Street	nominal	predictor	original
Alley	nominal	predictor	original

```
mod_rec
```

```
## Data Recipe
##
## Inputs:
##
##       role #variables
##    outcome          1
##  predictor         80
##
## Operations:
##
## Log transformation on all_outcomes()
## Collapsing factor levels for Neighborhood
## Collapsing factor levels for House_Style
## Zero variance filter on all_predictors()
## Sparse, unbalanced variable filter on all_predictors
## Centering for all_numeric(), -all_outcomes()
## Scaling for all_numeric(), -all_outcomes()
```

We prep the recipe on `ames_train`

```
mod_rec_trained ← prep(mod_rec,
                       training = ames_train,
                       verbose = TRUE, retain = TRUE)
## oper 1 step log [training]
## oper 2 step other [training]
## oper 3 step other [training]
## oper 4 step zv [training]
## oper 5 step nzv [training]
## oper 6 step center [training]
## oper 7 step scale [training]
## The retained training set is ~ 0.77 Mb  in memory.
```

and bake it on the `ames_test` data

```
ames_test_prep ← bake(mod_rec_trained,
                              new_data = ames_test)
```

We inspect the processed training and test set

```
dim(juice(mod_rec_trained))
```

```
## [1] 2053   61
```

Verify that `Sale_Price` is log-transformed (but not centred and scaled)

```
head(juice(mod_rec_trained)$Sale_Price)
head(ames_train$Sale_Price)
head(ames_test_prep$Sale_Price)
head(ames_test$Sale_Price)
```

```
## [1] 12.28 11.56 12.06 12.18 12.27 12.37
```

```
## [1] 215000 105000 172000 195500 213500 236500
```

```
## [1] 12.40 12.15 12.16 12.13 12.26 11.57
```

```
## [1] 244000 189900 191500 185000 212000 105500
```

```
levels(juice(mod_rec_trained)$House_Style)
```

```
levels(ames_test_prep$House_Style)
```

```
## [1] "One_and_Half_Fin" "One_Story"
## [1] "Two_Story" "other"
```

Putting it all together {rsample} and {recipes}

Let's redo the KNN example, with centering and scaling of the x-feature, by combining {rsample}/{caret} with a recipe.

```
# get the simulated data
set.seed(123)  # for reproducibility
x ← seq(from = 0, to = 2 * pi, length = 500)
y ← sin(x) + rnorm(length(x), sd = 0.3)
df ← data.frame(x, y) %>% filter(x < 4.5)
```

```
# specify the recipe
library(recipes)
rec ← recipe(y ~ x, data = df)
rec ← rec %>% step_center(all_predictors()) %>%
              step_scale(all_predictors())
```

```
# doing this on complete data set df
rec_df ← prep(rec, training = df)
mean(juice(rec_df)$x) # centered!
## [1] 1.473e-16
sd(juice(rec_df)$x)   # scaled!
## [1] 1
```

```
# now we combine the recipe with rsample steps
library(rsample)
set.seed(123)  # for reproducibility
cv_rsample ← vfold_cv(df, 5)
```

```
# we apply the steps in the recipe to each fold
library(purrr)
cv_rsample$recipes ← map(cv_rsample$splits, prepper,
                         recipe = rec)
# check ?prepper
```

Putting it all together {rsample} and {recipes}

Let's redo the KNN example, with centering and scaling of the x-feature, by combining {rsample}/{caret} with a recipe.

Now you can inspect `cv_rsample` as follows

```
cv_rsample$recipes[[1]]
juice(cv_rsample$recipes[[1]])
bake(cv_rsample$recipes[[1]],
     new_data = assessment(cv_rsample$splits[[1]]))
```

```
holdout_results ← function(s, rec, k_val) {
  # Fit the model to the analysis data in split s
  df_train ← juice(rec)
  mod ← knnreg(y ~ x, k = k_val, data = df_train)
  # Get the remaining group
  holdout ← bake(rec, new_data = assessment(s))
  # Get predictions with the holdout data set
  res ← predict(mod, newdata = holdout)
  # Return observed and predicted values
  #                              on holdout set
  res ← tibble(obs = holdout$y, pred = res)
  res
}
```

```
res ← holdout_results(cv_rsample$splits[[2]],
                      cv_rsample$recipes[[2]],
                      k_val = 58)
sqrt(sum((res$obs - res$pred)^2)/nrow(res))
## [1] 0.3505
```

Putting it all together {rsample} and {recipes}

Let's redo the KNN example, with centering and scaling of the x-feature, by combining {rsample}/{caret} with a recipe.

```
RMSE ← numeric(nrow(hyper_grid))
SE ← numeric(nrow(hyper_grid))
for(i in 1:nrow(hyper_grid)){
  cv_rsample$results ← map2(cv_rsample$splits, cv_rsample$recipes,
                            holdout_results,
                            hyper_grid[i, ])
  res ← map_dbl(cv_rsample$results,
                function(x) mean((x$obs - x$pred)^2))
  RMSE[i] ← mean(sqrt(res)) ; SE[i] ← sd(sqrt(res))
}
```

Regression models in R and tidy model output with {broom}

Creating models in R

The **formula** interface using R's formula rules to specify a *symbolic* representation of the terms:

- response ~ variable, with `model_fn` referring to the specific model function you want to use, e.g. `lm` for linear regression

```
model_fn(Sale_Price ~ Gr_Liv_Area, data = ames)
```

- response ~ variable_1 + variable_2

```
model_fn(Sale_Price ~ Gr_Liv_Area + Neighborhood, data = ames)
```

- response ~ variable_1 + variable_2 + their interaction

```
model_fn(Sale_Price ~ Gr_Liv_Area + Neighborhood + Neighborhood:Gr_Liv_Area, data = ames)
```

- shorthand for all predictors

```
model_fn(Sale_Price ~ ., data = ames)
```

Your turn

You will now fit some linear regression models on the `ames` housing data.

You will explore the model fits with `base` R instructions as well as the functionalities offered by the `broom` package.

Q: load the `ames` housing data set via `ames ← AmesHousing::make_ames()`

1. Fit a linear regression model with `Sale_Price` as response and `Gr_Liv_Area` as covariate. Store the resulting object as `model_1`.
2. Repeat your instruction, but now put it between brackets. What happens?
3. Inspect `model_1` with the following set of instructions

- `summary(___)`
- extract the fitted coefficients, using `___$coefficients`
- what happens with `summary(___)$coefficients`?
- extract fitted values, using `___$fitted.values`
- now try to extract the R^2 of this model.

Q.1 Linear model with `Sale_Price` as a function of `Gr_Live_Area`

```
model_1 ← lm(Sale_Price ~ Gr_Liv_Area, data = ames)
```

Q.3 Check `model_1` - What happens - do you *like* this display?

```
summary(model_1)
```

Now let's extract some meaningful information from `model_1` (using `base` R instructions)

```
model_1$coefficients
```

```
## (Intercept) Gr_Liv_Area
##     13289.6       111.7
```

```
summary(model_1)$coefficients
```

```
##              Estimate Std. Error t value  Pr(>|t|)
## (Intercept)  13289.6   3269.703   4.064 4.941e-05
## Gr_Liv_Area    111.7      2.066  54.061 0.000e+00
```

```
head(model_1$fitted.values)
```

```
##      1      2      3      4      5      6
## 198255 113367 161731 248964 195239 192447
```

```
summary(model_1)$r.squared
```

```
## [1] 0.4995
```

Tidy model output

The package {broom} allows to summarize key information about statistical objects (e.g. a linear regression model) in so-called tidy tibbles.

This makes it easy to report results, create plots and consistently work with large numbers of models at once.

We briefly illustrate the three essential verbs of `broom`: `tidy()`, `glance()` and `augment()`.

```
model_1 %>% broom::tidy()
```

term	estimate	std.error	statistic	p.value
(Intercept)	13289.6	3269.703	4.064	0
Gr_Liv_Area	111.7	2.066	54.061	0

```
model_1 %>% broom::glance()
```

r.squared	adj.r.squared	sigma	statistic	p.value	df	logLik	AIC	BIC	deviance	df.residual	nobs
0.4995	0.4994	56524	2923	0	1	-36218	72442	72460	9.355e+12	2928	2930

Tidy model output

The package {broom} allows to summarize key information about statistical objects (e.g. a linear regression model) in so-called tidy tibbles.

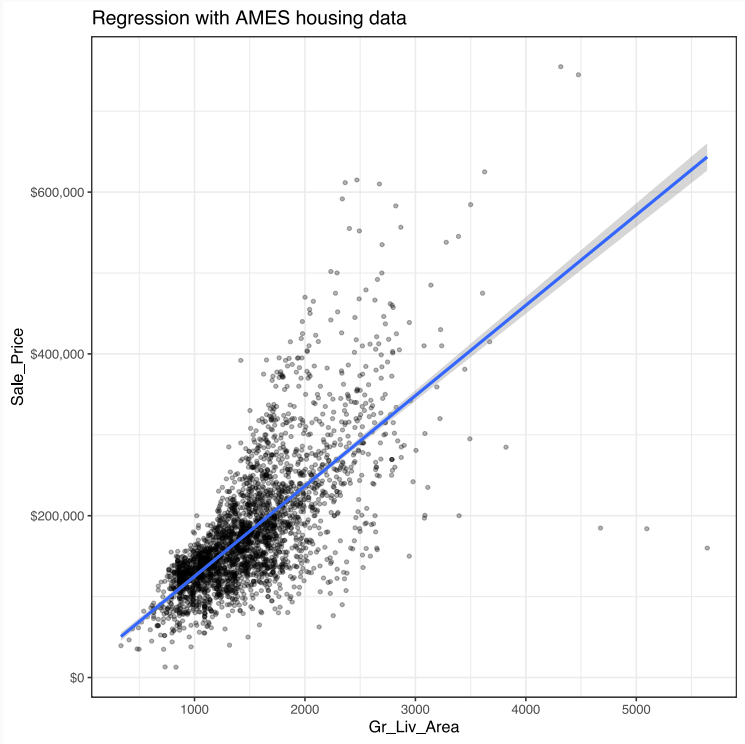
This makes it easy to report results, create plots and consistently work with large numbers of models at once.

We briefly illustrate the three essential verbs of `broom`: `tidy()`, `glance()` and `augment()`.

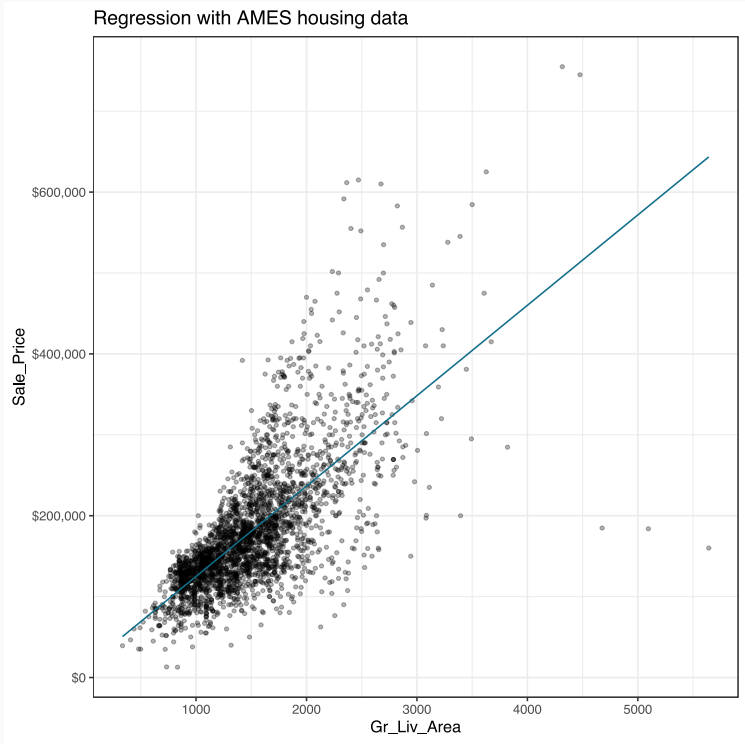
```
model_1 %>% broom::augment() %>% slice(1:5)
```

Sale_Price	Gr_Liv_Area	.fitted	.resid	.std.resid	.hat	.sigma	.cooksd
215000	1656	198255	16745	0.2963	4e-04	56533	0
105000	896	113367	-8367	-0.1481	8e-04	56534	0
172000	1329	161731	10269	0.1817	4e-04	56534	0
244000	2110	248964	-4964	-0.0879	8e-04	56534	0
189900	1629	195239	-5339	-0.0945	4e-04	56534	0

```
g_lm_1 ← ggplot(data = ames,
                aes(Gr_Liv_Area, Sale_Price)) +
  theme_bw() +
  geom_point(size = 1, alpha = 0.3) +
  geom_smooth(se = TRUE, method = "lm") +
  scale_y_continuous(labels = scales::dollar) +
  ggtitle("Regression with AMES housing data")
g_lm_1
```

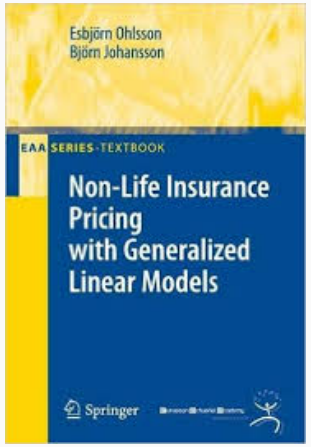


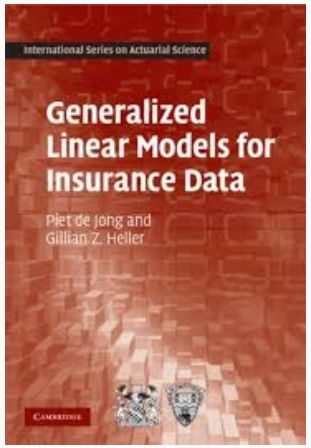

```
g_lm_2 ← model_1 %>% broom::augment() %>%
ggplot(aes(Gr_Liv_Area, Sale_Price)) +
    theme_bw() +
    geom_point(size = 1, alpha = 0.3) +
    geom_line(aes(y = .fitted), col = KULbg) +
    scale_y_continuous(labels = scales::dollar) +
    ggtitle("Regression with AMES housing data")
g_lm_2
```



Generalized Linear Models

Linear and Generalized Linear Models





With **linear regression models** `lm(.)`

- model specification

$$Y = x^{'}\beta + \epsilon.$$

- ϵ is normally distributed with mean 0 and common variance σ^2, thus: Y is normal with mean $x^{'}\beta$ and variance σ^2

With **generalized linear regression models** `glm(.)`

- model specification

$$g(E[Y]) = x^{'}\beta.$$

- $g(.)$ is the link function
- Y follows a distribution from the exponential family.

Motor Third Party Liability data

We will use the Motor Third Party Liability data set. There are 163,231 policyholders in this data set.

The frequency of claiming (`nclaims`) and corresponding severity (`avg`, the amount paid on average per claim reported by a policyholder) are the **target variables** in this data set.

Predictor variables are:

- the exposure-to-risk, the duration of the insurance coverage (max. 1 year)
- factor variables, e.g. gender, coverage, fuel
- continuous, numeric variables, e.g. age of the policyholder, age of the car
- spatial information: postal code (in Belgium) of the municipality where the policyholder resides.

More details in Henckaerts et al. (2018, Scandinavian Actuarial Journal) and Henckaerts et al. (2020, North American Actuarial Journal).

Motor Third Party Liability data

You can load the data from the `data` folder as follows:

```
# install.packages("rstudioapi")
dir ← dirname(rstudioapi::getActiveDocumentContext()$path)
setwd(dir)
mtpl_orig ← read.table('../data/PC_data.txt',
                       header = TRUE,
                       stringsAsFactors = TRUE)
mtpl_orig ← as_tibble(mtpl_orig)
```

Alternatively, you can also go for:

```
# install.packages("here")
dir ← here::here()
setwd(dir)
mtpl_orig ← read.table('../data/PC_data.txt',
                       header = TRUE,
                       stringsAsFactors = TRUE)
mtpl_orig ← as_tibble(mtpl_orig)
```

Some basic exploratory steps with this data follow on the next sheet.

Motor Third Party Liability data

Note that the data `mtpl_orig` uses capitals for the variable names

```
mtpl_orig %>% slice(1:3) %>% select(-LONG, -LAT) %>% kable(format = 'html')
```

ID	NCLAIMS	AMOUNT	AVG	EXP	COVERAGE	FUEL	USE	FLEET	SEX	AGEPH	BM	AGEC	POWER	PC	TOWN
1	1	1618	1618	1	TPL	gasoline	private	N	male	50	5	12	77	1000	BRUSSEL
2	0	0	NA	1	PO	gasoline	private	N	female	64	5	3	66	1000	BRUSSEL
3	0	0	NA	1	TPL	diesel	private	N	male	60	0	10	70	1000	BRUSSEL

We change this to lower case variables, and rename `exp` to `expo`.

```
mtpl ← mtpl_orig %>% rename_all(tolower) %>% rename(expo = exp)
names(mtpl)
##  [1] "id"       "nclaims"  "amount"   "avg"      "expo"     "coverage" "fuel"     "use"      "fleet"    "sex"
## [13] "agec"     "power"    "pc"       "town"     "long"     "lat"
```

```
dim(mtpl)
```

```
## [1] 163231     18
```

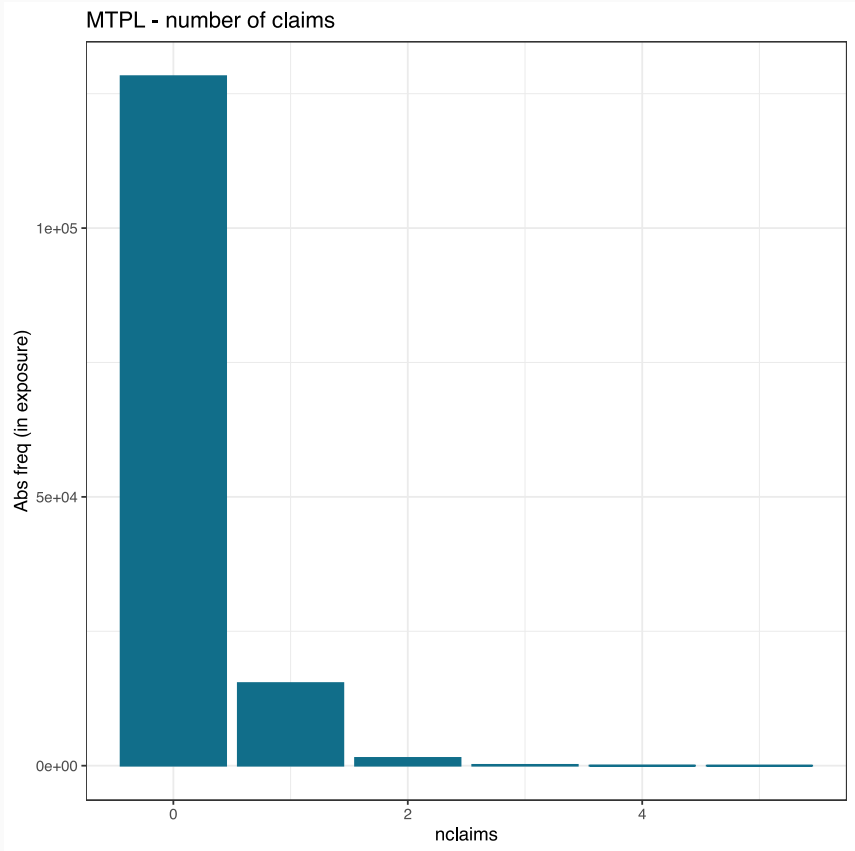
```
mtpl %>%
  summarize(emp_freq = sum(nclaims) / sum(expo))
```

emp_freq
0.1393

```
mtpl %>%
  group_by(sex) %>%
  summarize(emp_freq = sum(nclaims) / sum(expo))
```

sex	emp_freq
female	0.1484
male	0.1361

```
g ← ggplot(mtpl, aes(nclaims)) + theme_bw() +
    geom_bar(aes(weight = expo),
             col = KULbg, fill = KULbg) +
    labs(y = "Abs freq (in exposure)") +
    ggtitle("MTPL - number of claims")
g
```



Your turn

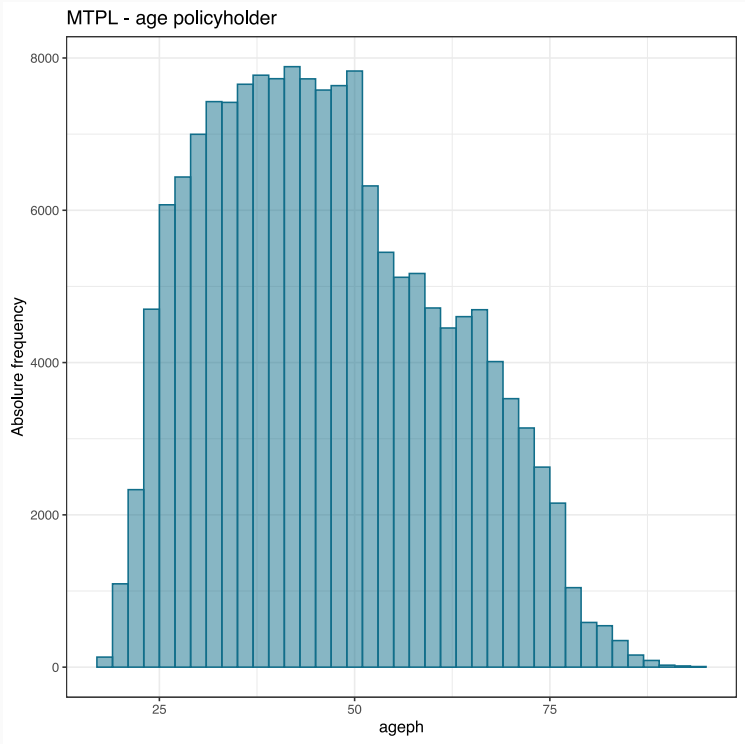
To get warmed up, let's load the `mtpl` data and do some basic investigations into the variables. The idea is to get a feel for the data.

Q: you will work through the following exploratory steps.

1. Visualize the distribution of the `ageph` with a histogram.
2. For each age recorded in the data set `mtpl`: what is the total number of observations, the total exposure, and the corresponding total number of claims reported?
3. Calculate the empirical claim frequency, per unit of exposure, for each age and picture it. Discuss this figure.
4. Repeat the above for `bm`, the level occupied by the policyholder in the Belgian bonus-malus scale.

Q.1 a histogram of `ageph`

```
ggplot(data = mtpl, aes(ageph)) + theme_bw() +
  geom_histogram(binwidth = 2, alpha = .5,
                 col = KULbg, fill = KULbg) +
  labs(y = "Absolure frequency") +
  ggtitle("MTPL - age policyholder")
```



Q.2 for each `ageph` recorded

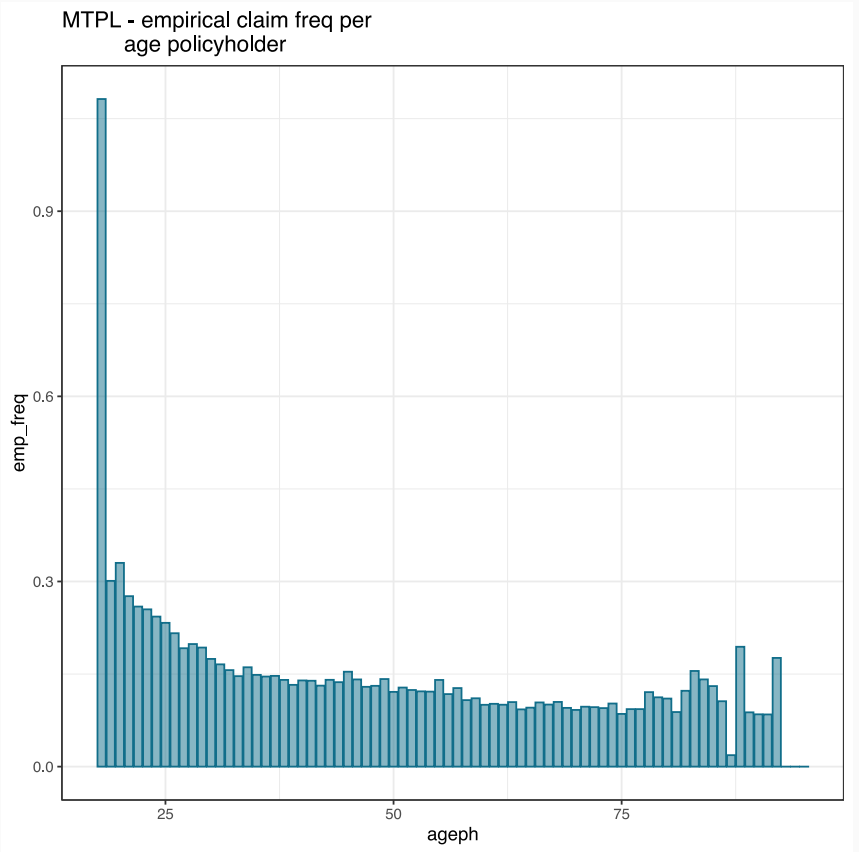
```
mtpl %>%
  group_by(ageph) %>%
  summarize(tot_claims = sum(nclaims),
            tot_expo = sum(expo),
            tot_obs = n())
```

ageph	tot_claims	tot_expo	tot_obs
18	5	4.621918	16
19	28	93.021918	116
20	113	342.284932	393
21	165	597.389041	701
22	202	778.827397	952
23	297	1165.358904	1379
24	426	1752.249315	2028

Q.3 for each `ageph` recorded

```
freq_by_age ← mtpl %>%
  group_by(ageph) %>%
  summarize(emp_freq = sum(nclaims) / sum(expo))

ggplot(data = freq_by_age,
       aes(x = ageph, y = emp_freq)) + theme_bw() +
  geom_bar(stat = 'identity', alpha = .5,
           color = KULbg, fill = KULbg) +
  ggtitle('MTPL - empirical claim freq per
          age policyholder')
```

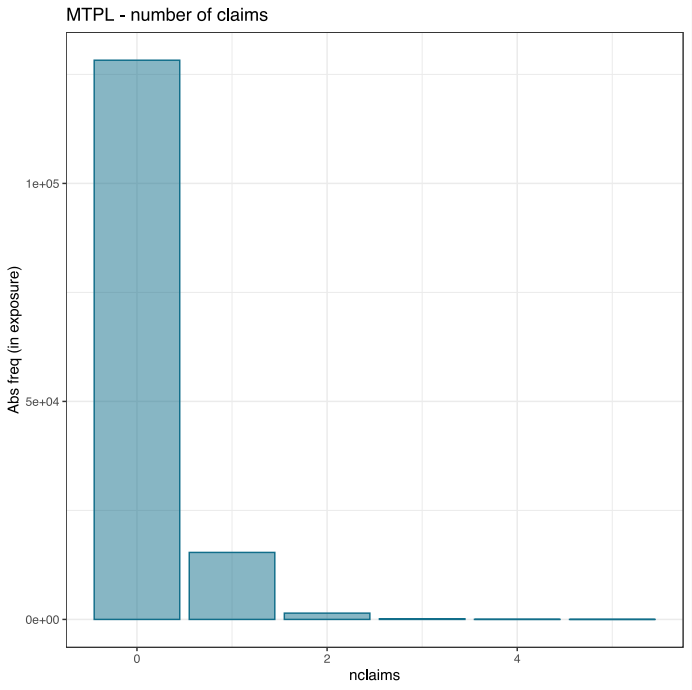


Q.4 recycle the above instructions and replace `ageph` with `bm`

Generalized Linear Models (GLMs)

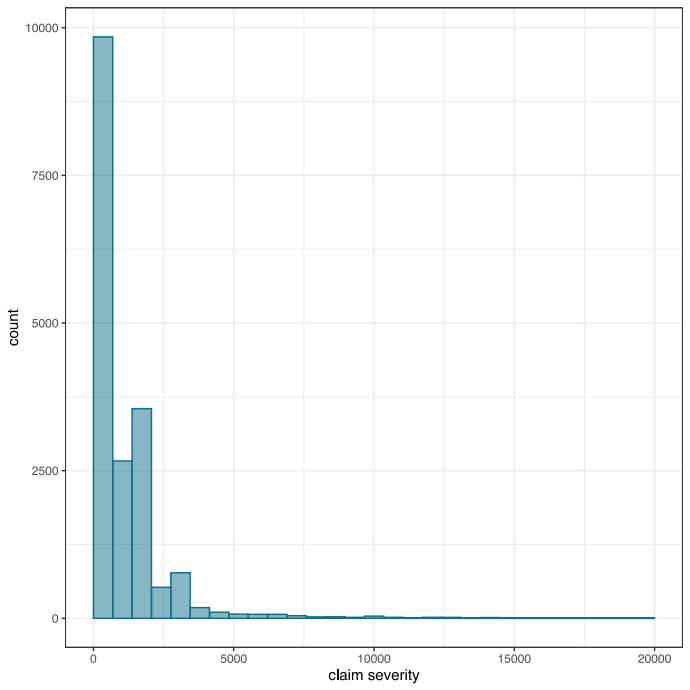
Modeling claim **frequency** and **severity** in the `mtpl` data set.

Target variable `nclaims` (frequency)



Suitable distributions: Poisson, Negative Binomial.

... and `avg` (severity).



Suitable distributions: log-normal, gamma.

A Poisson GLM

```
freq_glm_1 ← glm(nclaims ~ sex, offset = log(expo),
                 family = poisson(link = "log"),
                 data = mtpl)
```

Fit a Poisson GLM, with logarithmic link function.

This implies:

Y ~ Poisson, with

$$\log(E[Y]) = x^{'}\beta,$$

or,

$$E[Y] = \exp\left(x^{'}\beta\right).$$

Fit this model on `data = mtpl`.

A Poisson GLM (cont.)

```
freq_glm_1 ← glm(nclaims ~ sex, offset = log(expo),
                 family = poisson(link = "log"),
                 data = mtpl)
```

Use `nclaims` as Y.

Use `gender` as the only (factor) variable in the linear predictor.

Include `log(expo)` as an offset term in the linear predictor.

Then,

$$x'\beta = \log(\mathbf{expo}) + \beta_0 + \beta_1 \mathbb{I}(\mathbf{male}).$$

Put otherwise,

$$E[Y] = \mathbf{expo} \cdot \exp\left(\beta_0 + \beta_1 \mathbb{I}(\mathbf{male})\right),$$

where **expo** refers to `expo` the exposure variable.

```
freq_glm_1 ← glm(nclaims ~ sex, offset = log(expo),
                 family = poisson(link = "log"),
                 data = mtpl)
```

```
freq_glm_1 %>% broom::tidy()
```

term	estimate	std.error	statistic	p.value
(Intercept)	-1.9076251	0.0133227	-143.186324	0
sexmale	-0.0866198	0.0156837	-5.522931	0

Mind the specification of `type.predict` when using `augment` with a GLM!

```
freq_glm_1 %>% broom::augment(type.predict = "response"
```

nclaims	sex	.fitted
1	male	0.1361164
0	female	0.1484325

The `predict` function of a GLM object offers 3 options: `"link"`, `"response"` or `"terms"`.

The same options hold when `augment()` is applied to a GLM object.

Let's see how the fitted values at `"response"` level are constructed:

```
exp(coef(freq_glm_1)[1])
## (Intercept)
##   0.1484325
exp(coef(freq_glm_1)[1] + coef(freq_glm_1)[2])
## (Intercept)
##   0.1361164
```

Do you recognize these numbers?

Last step:

try `freq_glm_1 %>% glance()` or `summary(freq_glm_1)` for deviances.

Your turn

You will further explore GLMs in R with the `glm(.)` function.

Q: continue with the `freq_glm_1` object that was created, you will now explicitly call the `predict()` function on this object.

1. Verify the arguments of `predict.glm` using `? predict.glm`.

2. The help reveals the following structure `predict(.object, .newdata, type = (" ... "))` where `.object` is the fitted GLM object, `.newdata` is (optionally) a data frame to look for the features used in the model, and `type` is `"link"`, `"response"` or `"terms"`.
   Use `predict` with `freq_glm_1` and a newly created data frame.
   Explore the different options for `type`, and their connections.

3. Fit a gamma GLM for `avg` (the claim severity) with log link.
   Use `sex` as the only variable in the model. What do you conclude?

Q.1 You can access the documentation via `? predict.glm`.

Q.2 You create new data frames (or tibbles) as follows

```
male_driver ← data.frame(expo = 1, sex = "male")
female_driver ← data.frame(expo = 1, sex = "female")
```

Next, you apply `predict` with the GLM object `freq_glm_1` and one of these data frames, e.g.

```
predict(freq_glm_1, newdata = male_driver,
                    type = "response")
```

```
##         1
## 0.1361164
```

Q.2 Next, you apply `predict` with the GLM object `freq_glm_1` and one of these data frames, e.g.

```
predict(freq_glm_1, newdata = male_driver,
                    type = "response")
```

```
##         1
## 0.1361164
```

At the level of the linear predictor:

```
predict(freq_glm_1, newdata = male_driver,
                    type = "link")
```

```
##         1
## -1.994245
```

```
exp(predict(freq_glm_1, newdata = male_driver,
                        type = "link"))
```

```
##         1
## 0.1361164
```

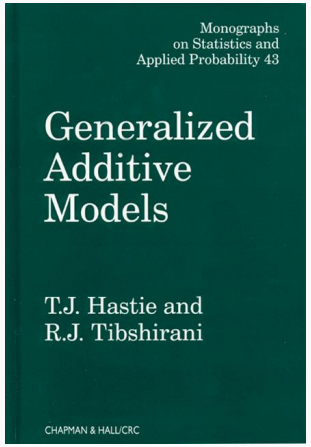
Q.3 For the gamma regression model

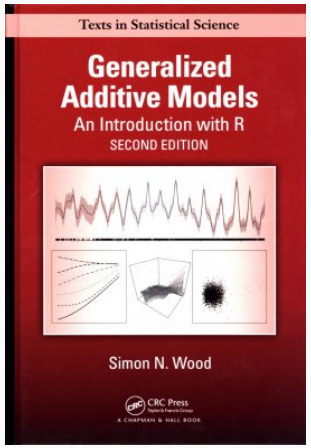
```
sev_glm_1 ← glm(avg ~ sex, family = Gamma(link = "log"), data = mtpl)
sev_glm_1
```

```
##
## Call:  glm(formula = avg ~ sex, family = Gamma(link = "log"), data = mtpl)
##
## Coefficients:
## (Intercept)      sexmale
##      7.5730      -0.2581
##
## Degrees of Freedom: 18294 Total (i.e. Null);  18293 Residual
##   (144936 observations deleted due to missingness)
## Null Deviance:        46690
## Residual Deviance: 46440     AIC: 299700
```

Generalized Additive Models with {mgcv}

Generalized Additive Models (GAMs)





With **GLMs** `glm(.)`

- transformation of the mean modelled with a linear predictor

$$x'\beta$$

- not well suited for continuous risk factors that relate to the response in a non-linear way.

With **Generalized Additive Models (GAMs)**

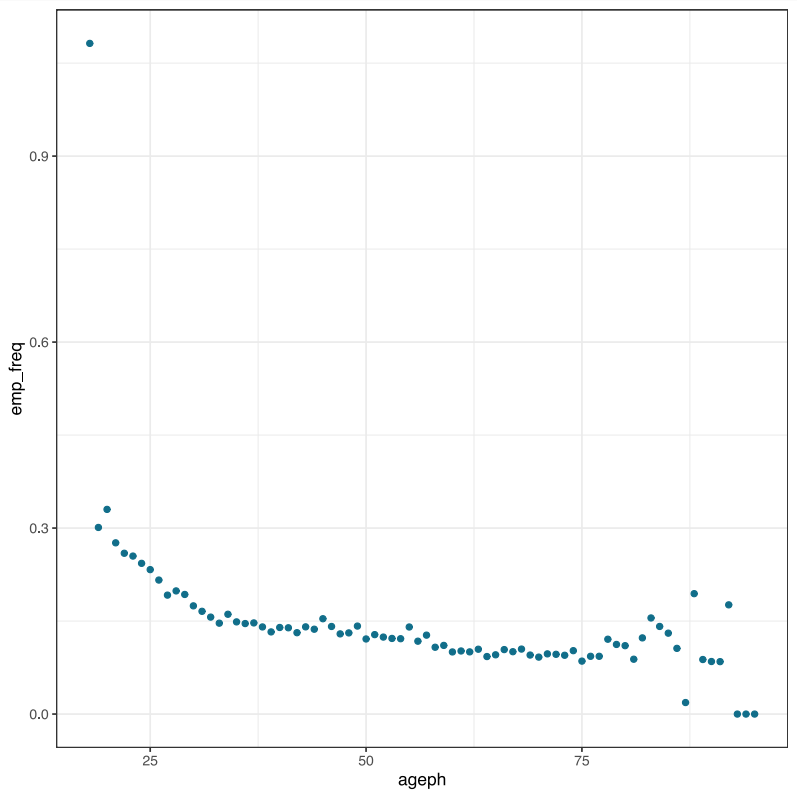
- the predictor allows for smooth effects of continuous risk factors and spatial covariates, next to the linear terms, e.g.

$$x'\beta + \sum_j f_j(x_j) + f(\texttt{lat}, \texttt{long})$$

- predictor is still additive
- preferred R package is {mgcv} by Simon Wood.

A Poisson GAM

We continue working with `mtpl` and now focus on `ageph`.



We will now explore **four different model specifications**:

1. `ageph` as linear effect in `glm`
2. `ageph` as factor variable in `glm`
3. `ageph` split manually into bins using `cut`, then used as factor in `glm`
4. a smooth effect of `ageph` in `mgcv::gam`.

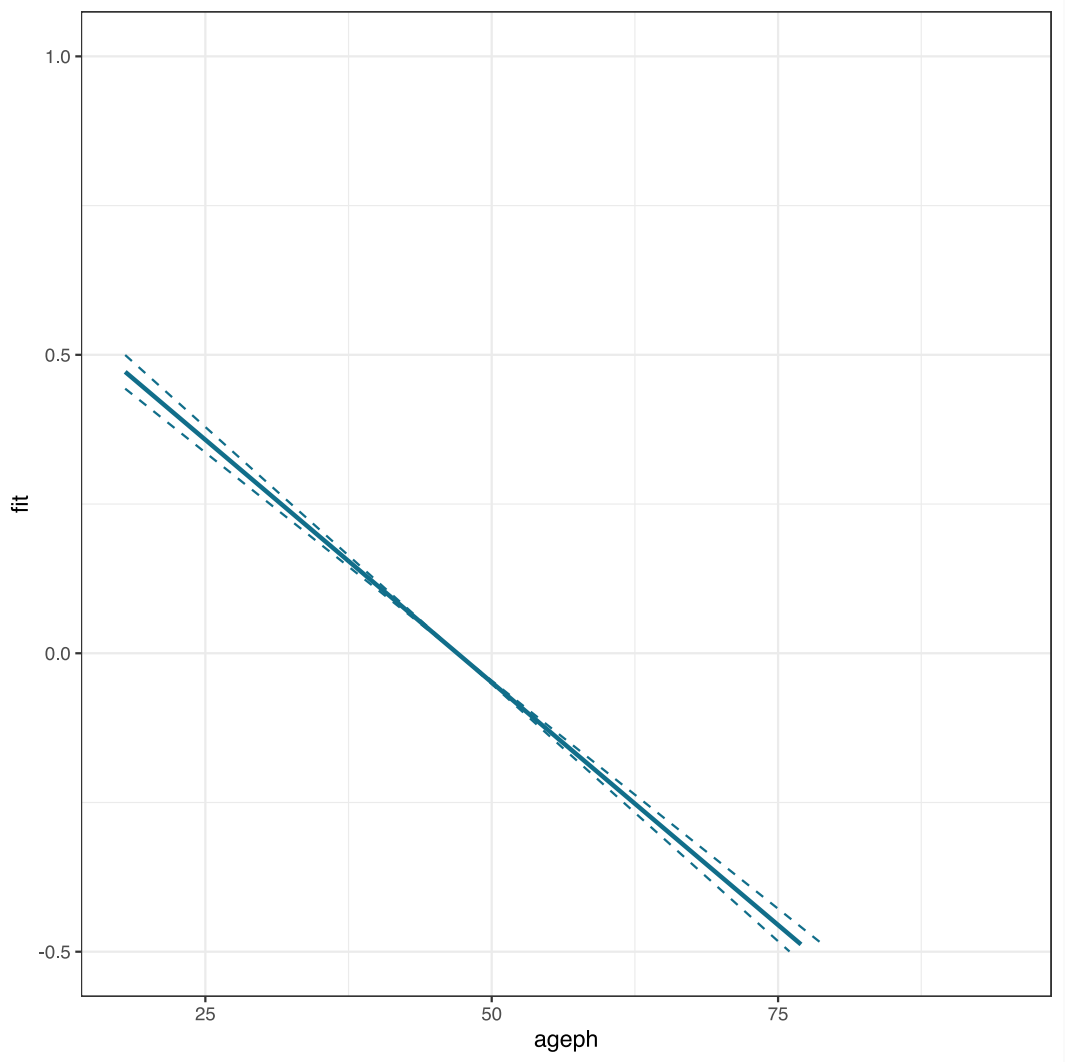
Let's go!

Grid of observed `ageph` values

```
a ← min(mtpl$ageph):max(mtpl$ageph)
```

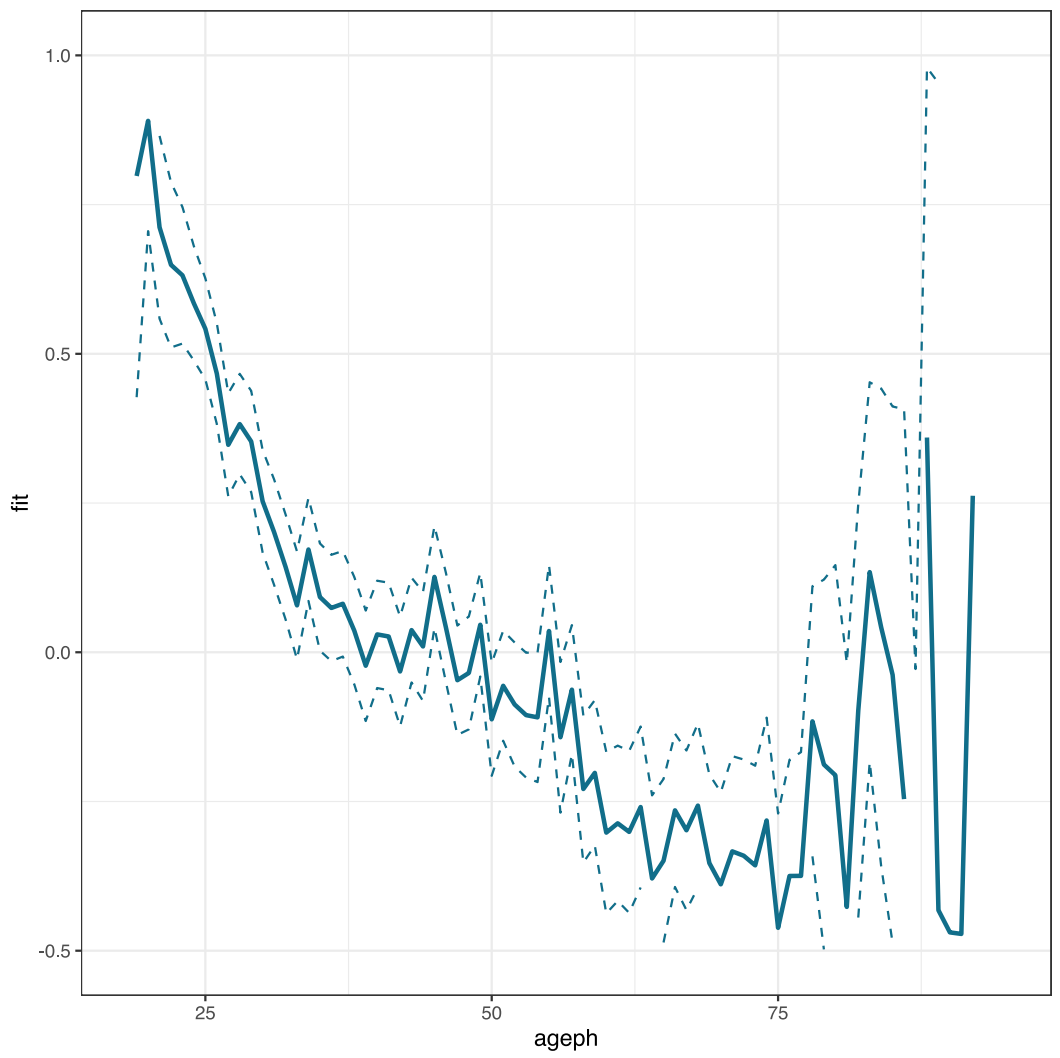
Model 1: linear effect of `ageph`

```
freq_glm_age ← glm(nclaims ~ ageph,
                   offset = log(expo),
                   data = mtpl,
                   family = poisson(link = "log"))
pred_glm_age ← predict(freq_glm_age,
          newdata = data.frame(ageph = a, expo = 1),
          type = "terms", se.fit = TRUE)
b_glm_age ← pred_glm_age$fit
l_glm_age ← pred_glm_age$fit
                 - qnorm(0.975)*pred_glm_age$se.fit
u_glm_age ← pred_glm_age$fit
                 + qnorm(0.975)*pred_glm_age$se.fit
df ← data.frame(a, b_glm_age, l_glm_age, u_glm_age)
```



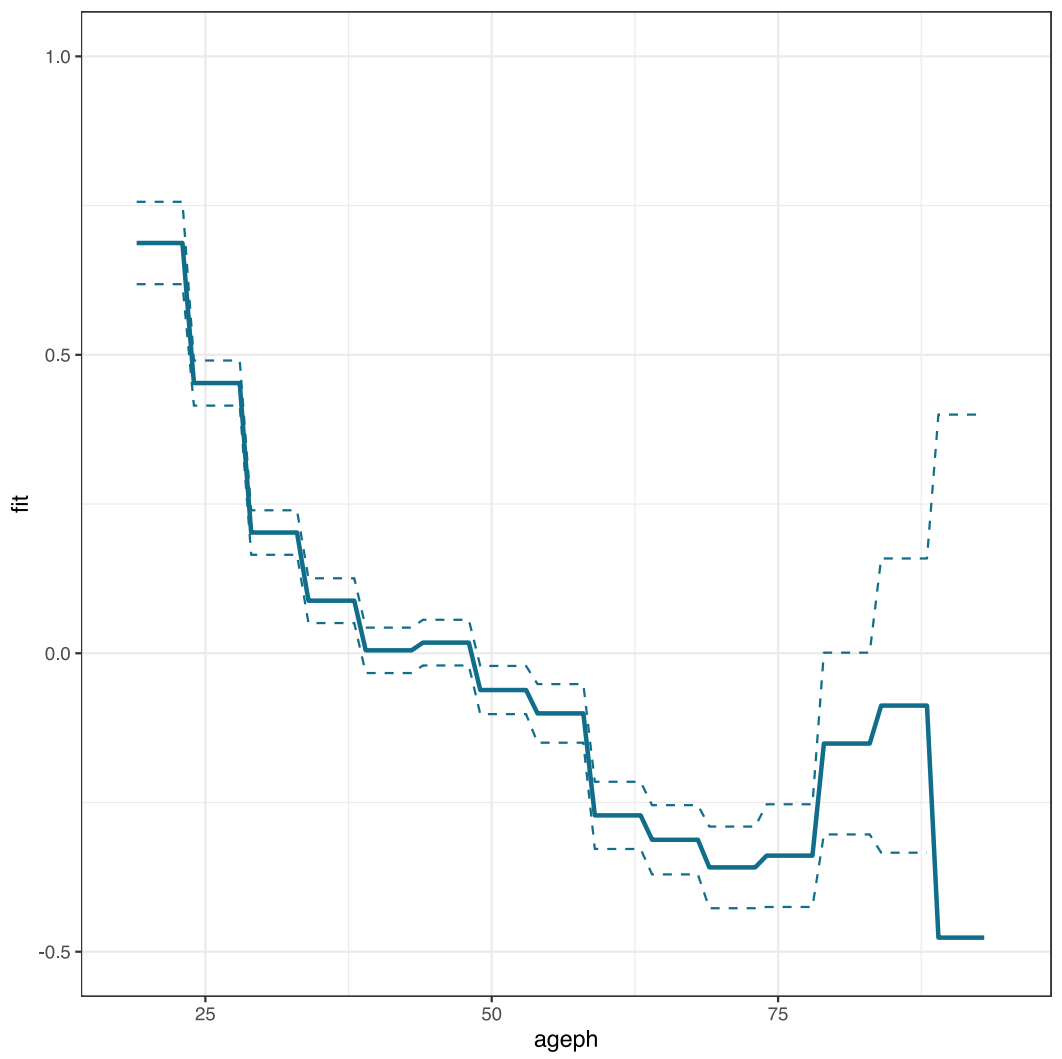
Model 2: `ageph` as factor variable in `glm`

```
freq_glm_age_f ← glm(nclaims ~ as.factor(ageph),
                     offset = log(expo),
                     data = mtpl,
                     family = poisson(link = "log"))
pred_glm_age_f ← predict(freq_glm_age_f,
          newdata = data.frame(ageph = a, expo = 1),
          type = "terms", se.fit = TRUE)
b_glm_age_f ← pred_glm_age_f$fit
l_glm_age_f ← pred_glm_age_f$fit
                  - qnorm(0.975)*pred_glm_age_f$se.fit
u_glm_age_f ← pred_glm_age_f$fit
                  + qnorm(0.975)*pred_glm_age_f$se.fit
df ← data.frame(a, b_glm_age_f,
                     l_glm_age_f, u_glm_age_f)
```



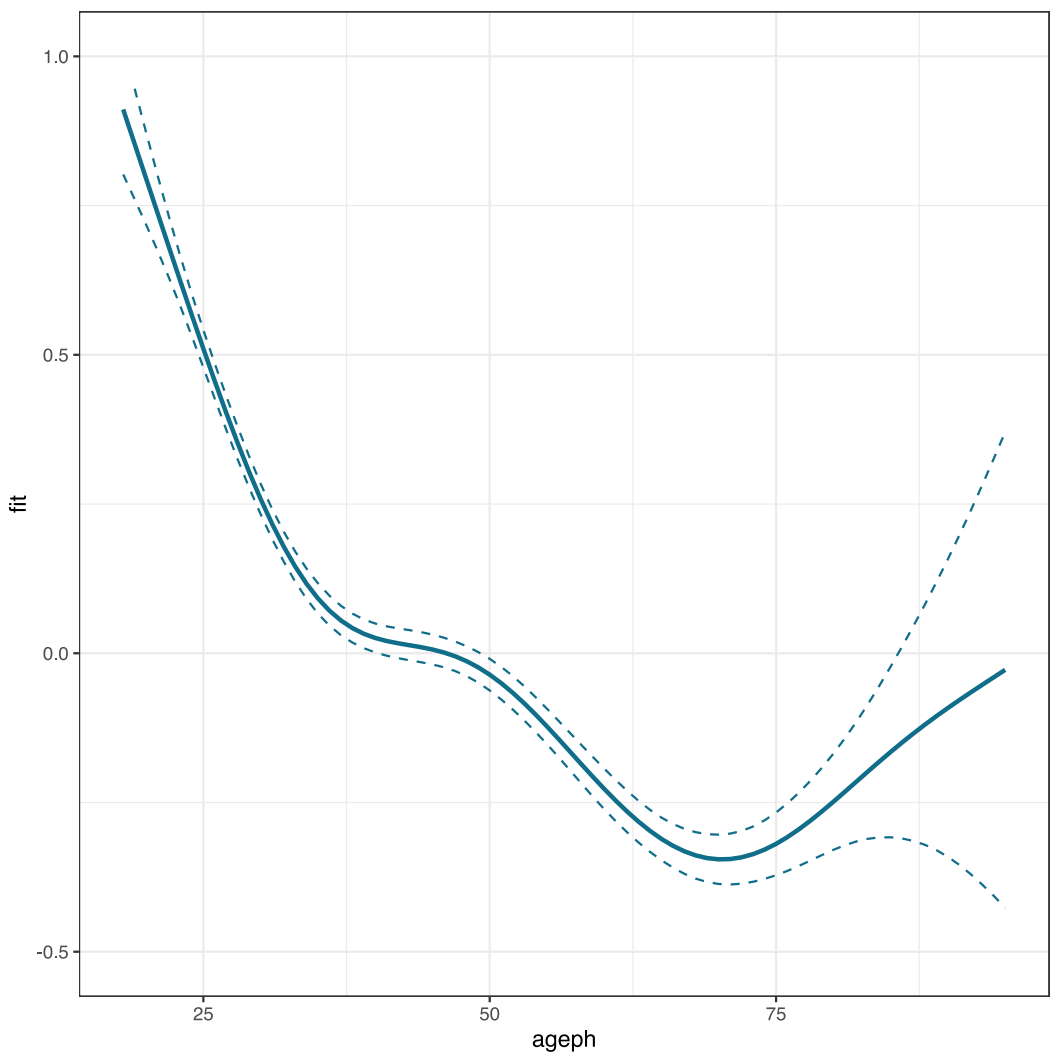
Model 3: `ageph` split into 5-year bins and then used in `glm`

```
level ← seq(min(mtpl$ageph), max(mtpl$ageph), by = 5)
freq_glm_age_c ← glm(nclaims ~ cut(ageph, level),
                     offset = log(expo),
                     data = mtpl,
                     family = poisson(link = "log"))
pred_glm_age_c ← predict(freq_glm_age_c,
          newdata = data.frame(ageph = a, expo = 1),
          type = "terms", se.fit = TRUE)
b_glm_age_c ← pred_glm_age_c$fit
l_glm_age_c ← pred_glm_age_c$fit
                   - qnorm(0.975)*pred_glm_age_c$se.fit
u_glm_age_c ← pred_glm_age_c$fit
                   + qnorm(0.975)*pred_glm_age_c$se.fit
df ← data.frame(a, b_glm_age_c,
                     l_glm_age_c, u_glm_age_c)
```



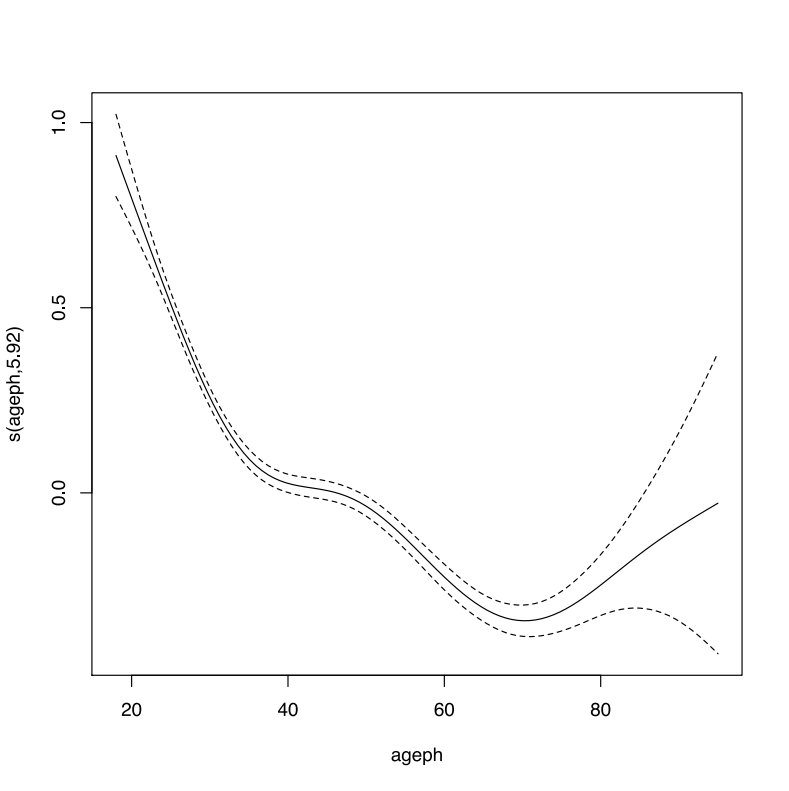
Model 4: smooth effect of `ageph` in `mgcv::gam`

```
library(mgcv)
freq_gam_age ← gam(nclaims ~ s(ageph),
                   offset = log(expo),
                   data = mtpl,
                   family = poisson(link = "log"))
pred_gam_age ← predict(freq_gam_age,
          newdata = data.frame(ageph = a, expo = 1),
          type = "terms", se.fit = TRUE)
b_gam_age ← pred_gam_age$fit
l_gam_age ← pred_gam_age$fit -
                  qnorm(0.975)*pred_gam_age$se.fit
u_gam_age ← pred_gam_age$fit +
                  qnorm(0.975)*pred_gam_age$se.fit
df ← data.frame(a, b_gam_age,
                   l_gam_age, u_gam_age)
```



Model 4 (revisited): picture smooth effect of `ageph` in `mgcv::gam` with built-in `plot`.

```
library(mgcv)
freq_gam ← gam(nclaims ~ s(ageph), offset = log(expo), family = poisson(link = "log"), data = mtpl)
plot(freq_gam, scheme = 4)
```



More on GAMs

So, a GAM is a GLM where the linear predictor depends on **smooth functions** of covariates.

Consider a GAM with the following predictor:

$$x^{'}\beta + f_j(x_j).$$

GAMs use **basis functions** to estimate the smooth effect $f_j(.)$

$$f_j(x_j) = \sum_{m=1}^{M} \beta_{jm} b_{jm}(x_j),$$

where the $b_{jm}(x)$ are known basis functions and β_{jm} are coefficients that have to be estimated.

GAMs avoid overfitting by adding a **wiggliness penalty** to the likelihood

$$\int \left(f_j(x)''\right)^2 = \beta_j^t \mathbf{S}_j \beta_j.$$

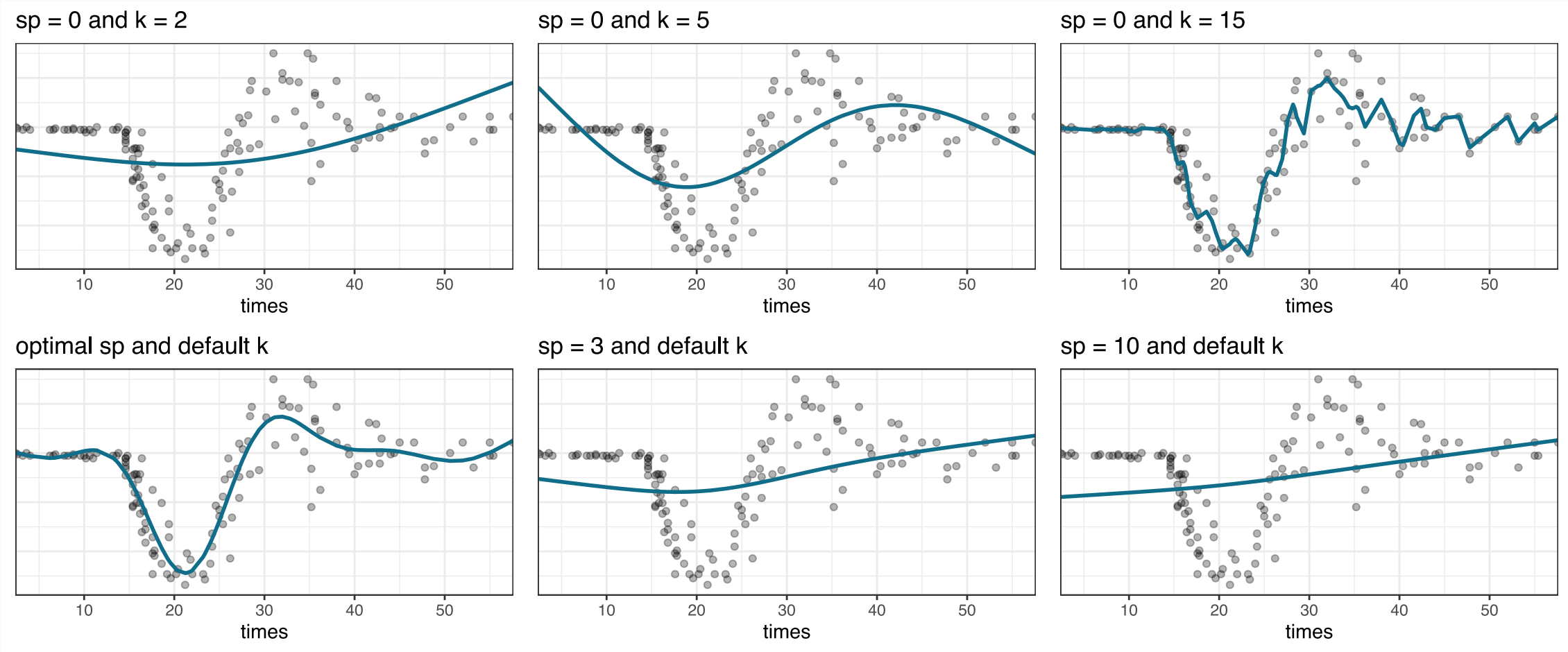
GAMs then balance goodness-of-fit and wiggliness via

$$\log \mathcal{L}(\beta, \beta_j) - \lambda_j \cdot \beta_j^t \mathbf{S}_j \beta_j,$$

with λ_j the **smoothing parameter**.

The smoothing parameter λ_j controls the trade-off between fit & smoothness.

Let's run some experiments to illustrate the effect of the smoothing parameter (`sp = .`), the number (`k = .`) and type of basis functions (`bs = .`). We use the `mcycle` data from {MASS}.



Your turn

You will further explore GAMs in R with the `gam(.)` function from the {mgcv} package.

Q: you will combine insights from building `glm` as well as `gam` objects by working through the following coding steps.

1. Fit a `gam` including some factor variables as well as a smooth effect of `ageph` and `bm`. Visualize the fitted smooth effects.

2. Specify risk profiles of drivers. Calculate their expected annual claim frequency from the constructed `gam`.

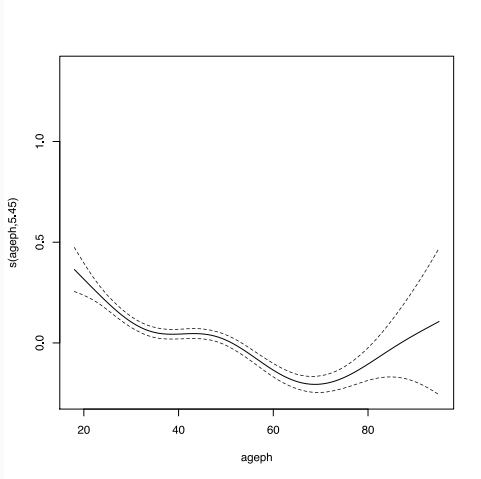
3. Explain (in words) which profiles would represent high vs low risk according to the constructed model.

Q.1 examine the following `gam` fit

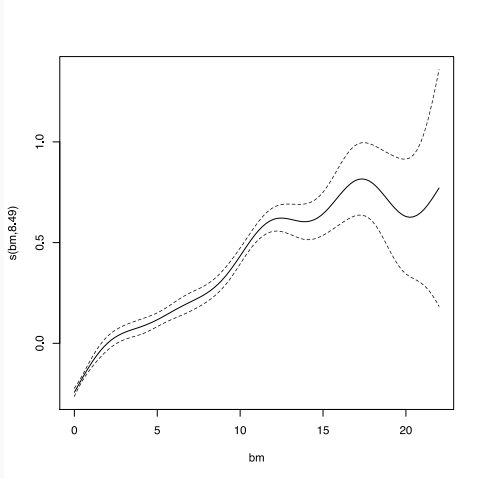
```
freq_gam_2 ← gam(nclaims ~ sex + fuel + use +
                   s(ageph) + s(bm),
                 offset = log(expo), data = mtpl,
                 family = poisson(link = "log"))
```

```
summary(freq_gam_2)
##
## Family: poisson
## Link function: log
##
## Formula:
## nclaims ~ sex + fuel + use + s(ageph) + s(bm)
##
## Parametric coefficients:
##              Estimate Std. Error  z value Pr(>|z|)
## (Intercept)  -1.917801   0.018124 -105.818   <2e-16
## sexmale       0.009177   0.016043    0.572   0.5673
## fuelgasoline -0.152756   0.015100  -10.116   <2e-16
## usework      -0.055156   0.033092   -1.667   0.0956
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.
##
## Approximate significance of smooth terms:
```

```
plot(freq_gam_2, select = 1)
```




```
plot(freq_gam_2, select = 2)
```



Q.2 define some risk profiles

```
drivers ← data.frame(expo = c(1, 1, 1),
                     sex = c("female", "female", "fema
                     fuel = c("diesel", "diesel", "die
                     use = c("private", "private", "pr
                     ageph = c(18, 45, 65), bm = c(20,
drivers
```

expo	sex	fuel	use	ageph	bm
1	female	diesel	private	18	20
1	female	diesel	private	45	5
1	female	diesel	private	65	0

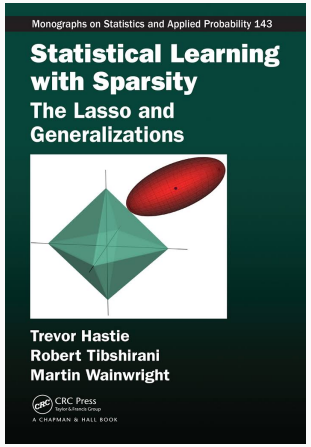
Now, you predict the annual expected claim frequency for these profiles.

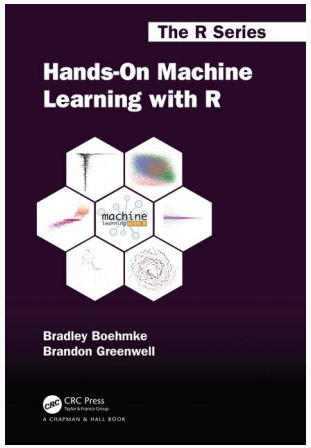
```
predict(freq_gam_2, newdata = drivers,
        type = "response")
```

x
0.3969310
0.1727430
0.0951124

Regularized (G)LMs met {glmnet}

Statistical learning with sparsity





Why?

- Sort through the mass of information and bring it down to **its bare essentials**.
- One form of simplicity is **sparsity**.
- Only a relatively small number of predictors play a role.

How? Automatic feature selection!

- Fit a model with all p predictors, but constrain or **regularize** the coefficient estimates.
- Shrinking the coeffcient estimates can signifcantly reduce their variance.
- Some types of shrinkage put some of the coefficients exactly equal to zero!

Ridge and lasso (least squares) regression

Ridge considers the least-squares optimization problem

$$\min_{\beta_0,\boldsymbol{\beta}} \sum_{i=1}^{n} \left(y_i - \beta_0 - \sum_{j=1}^{p} \beta_j x_{ij} \right)^2 = \min_{\beta_0,\boldsymbol{\beta}} \text{RSS}$$

subject to a **budget constraint**

$$\sum_{j=1}^{p} \beta_j^2 \leq t,$$

i.e. an ℓ_2 penalty.

Shrinks the coefficient estimates (not the intercept) to zero.

Lasso considers the least-squares optimization problem

$$\min_{\beta_0,\boldsymbol{\beta}} \sum_{i=1}^{n} \left(y_i - \beta_0 - \sum_{j=1}^{p} \beta_j x_{ij} \right)^2 = \min_{\beta_0,\boldsymbol{\beta}} \text{RSS}$$

subject to a **budget constraint**

$$\sum_{j=1}^{p} |\beta_j| \leq t,$$

i.e. an ℓ_1 penalty.

Shrinks the coefficient estimates (not the intercept) to zero and does variable selection!

Lasso is for **L**east **a**bsolute **s**hrinkage and **s**election **o**perator.

Ridge and lasso (least squares) regression (cont.)

The dual problem formulation:

- with ridge penalty:

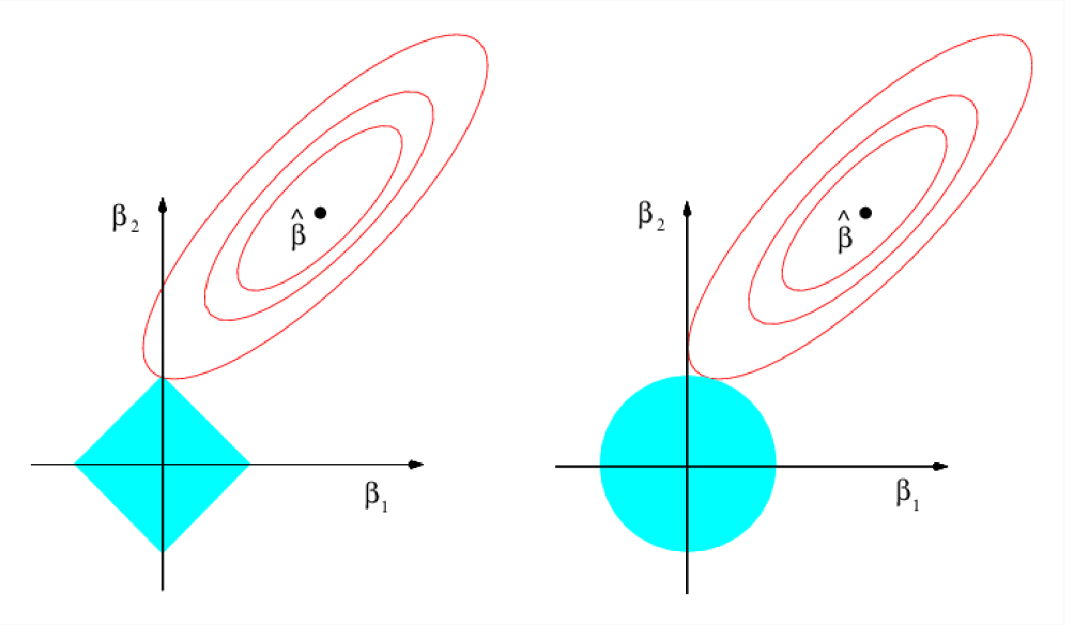
$$\min_{\beta_0, \boldsymbol{\beta}} \text{RSS} + \lambda \sum_{j=1}^{p} \beta_j^2$$

- with lasso penalty:

$$\min_{\beta_0, \boldsymbol{\beta}} \text{RSS} + \lambda \sum_{j=1}^{p} |\beta_j|.$$

λ is a tuning parameter; use resampling methods to pick a value!

Both ridge and lasso require **centering and scaling** of the features.



Ellipses (around least-squares solution) represent regions of constant RSS.

Lasso budget on the left and ridge budget on the right.

Source: James et al. (2013) on An introduction to statistical learning.

Regularized GLMs

We now focus on generalizations of linear models and the lasso.

Minimize

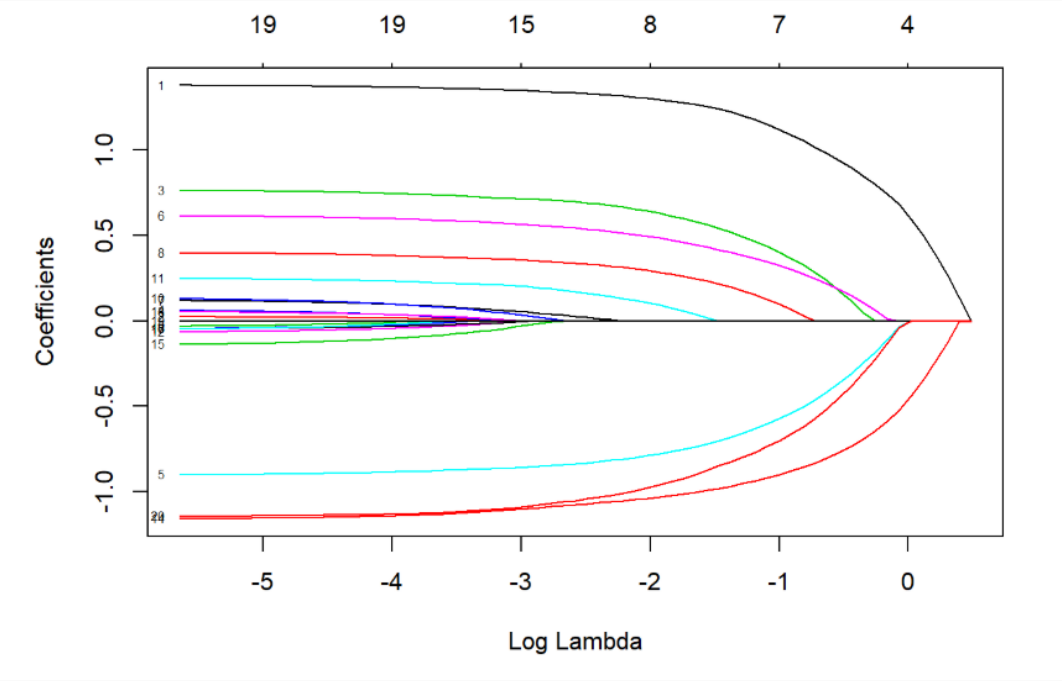
$$\min_{\beta_0,\,\beta} -\frac{1}{n}\mathcal{L}(\beta_0,\ \beta;\ y,\ X) + \lambda\|\boldsymbol{\beta}\|_1.$$

Here:

- $\mathcal{L}$ is the log-likelihood of a GLM.
- n is the sample size
- $\|\boldsymbol{\beta}\|_1 = \sum_{j=1}^{p} \beta_j$ the ℓ_1 penalty.

What happens if:

- $\lambda \to 0$?
- $\lambda \to \infty$?



The R package {glmnet} fits linear, logistic and multinomial, Poisson, and Cox regression models.

Fit a GLM with lasso regularization in {glmnet}

{glmnet} is a package that fits a generalized linear model via penalized maximum likelihood.

Main function call (with a selection of arguments, see `? glmnet` for a complete list)

```
fit ← glmnet(x, y, family = ., alpha = ., weights = ., offset = ., nlambda = ., standardize = ., intercept = .)
```

where

- `x` is the input matrix and `y` is the response variable
- `family` the response type, e.g. `family = poisson`
- `weights` and `offset`
- `nlambda` is the number of λ values, default is 100
- `standardize` should `x` be standardized prior to fitting the model sequence?
- `intercept` should incercept be fitted?
- `alpha` a value between 0 and 1, such that the penalty becomes

$$\lambda P_\alpha(\boldsymbol{\beta}) = \lambda \cdot \sum_{j=1}^{p} \left\{ \frac{(1-\alpha)}{2} \beta_j^2 + \alpha |\beta_j| \right\}.$$

Thus, with $\alpha = 1$ the lasso penalty and $\alpha = 0$ the ridge penalty results.

A first example of {glmnet}

Following the vignette we start with penalized linear regression

```
library(glmnet)
data(QuickStartExample)
```

This example loads an input matrix `x` and vector `y` of outcomes. The input matrix `x` is not standardized yet (check this!).

We calibrate a lasso linear regression model

```
fit ← glmnet(x, y, family = "gaussian",
             alpha = 1, standardize = TRUE,
             intercept = TRUE)
summary(fit)
```

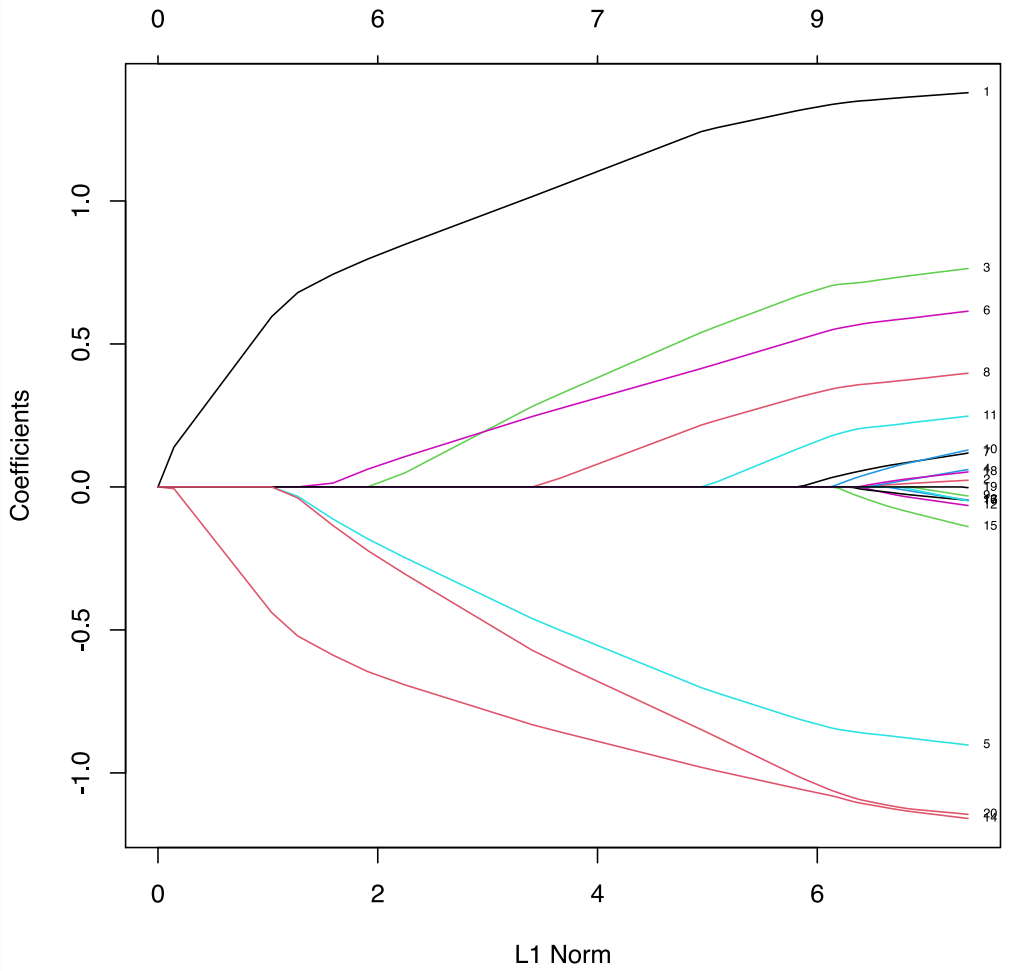
Note that the formula notation `y ~ x` can not be used with `glmnet`.

Some `tidy` instructions are available for `glmnet` objects (but not all), e.g.

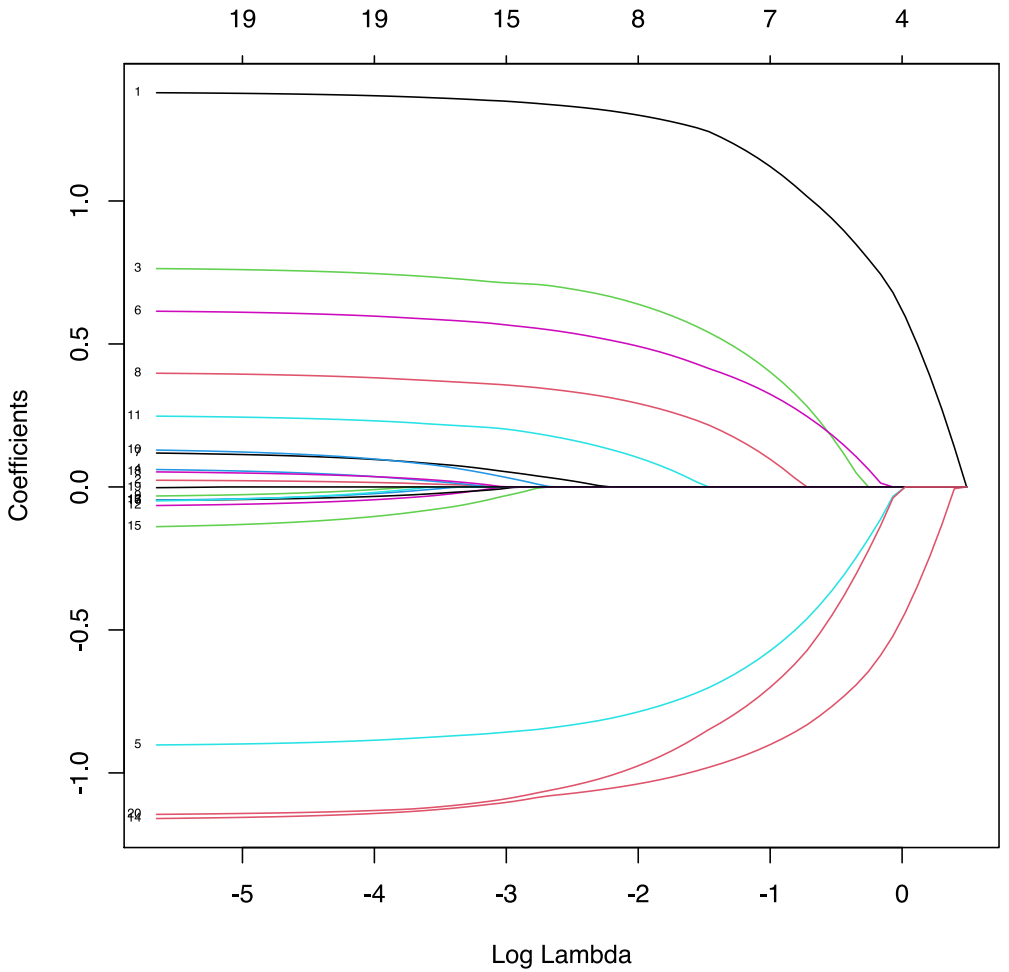
```
library(broom)
tidy(fit)
```

term	step	estimate	lambda	dev.ratio
(Intercept)	1	0.6607581	1.630762	0.0000000
(Intercept)	2	0.6312350	1.485890	0.0552832
(Intercept)	3	0.5874616	1.353887	0.1458910
(Intercept)	4	0.5475769	1.233612	0.2211153
(Intercept)	5	0.5112354	1.124021	0.2835678

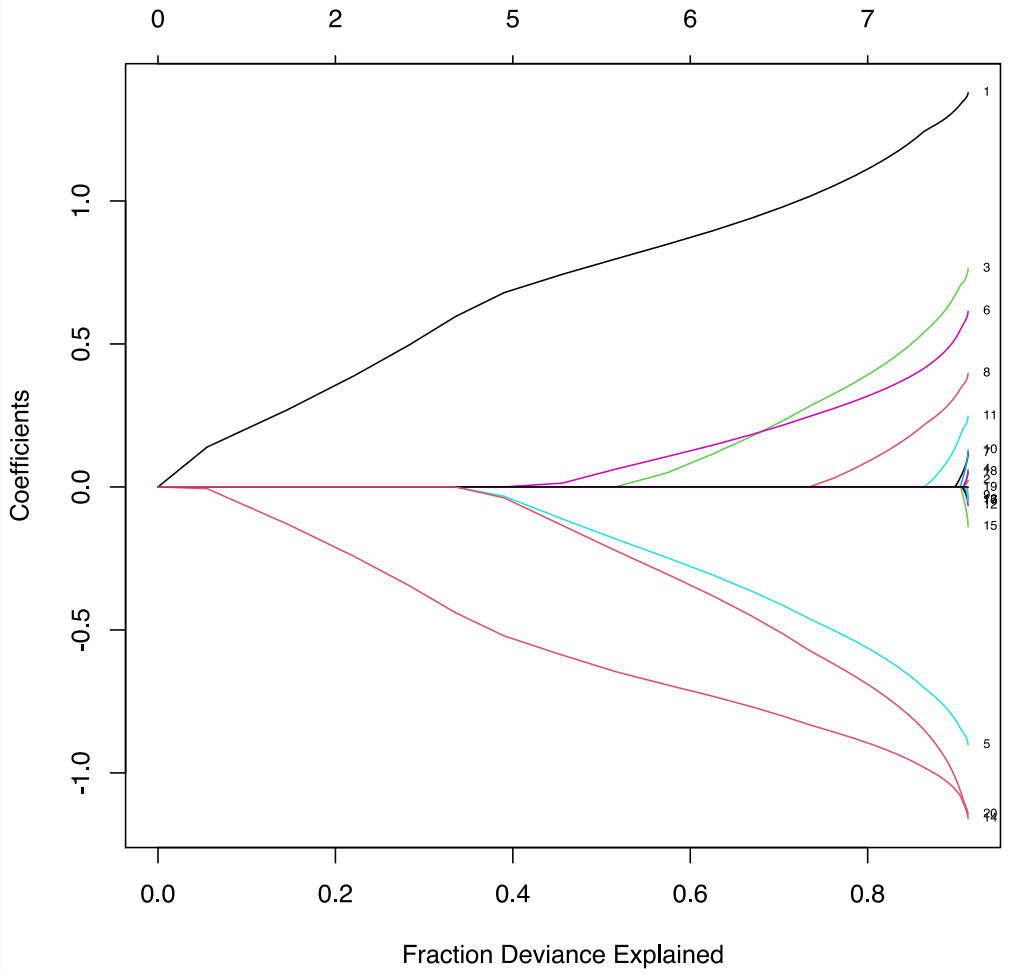
```
plot(fit, label = TRUE)
```




```
plot(fit, label = TRUE, xvar = 'lambda')
```



```
plot(fit, xvar = 'dev', label = TRUE)
```




```
print(fit)
##
## Call:  glmnet(x = x, y = y, family = "gaussian", alp
##
##     Df  %Dev  Lambda
## 1    0  0.00 1.63100
## 2    2  5.53 1.48600
## 3    2 14.59 1.35400
## 4    2 22.11 1.23400
## 5    2 28.36 1.12400
## 6    2 33.54 1.02400
## 7    4 39.04 0.93320
## 8    5 45.60 0.85030
## 9    5 51.54 0.77470
## 10   6 57.35 0.70590
## 11   6 62.55 0.64320
## 12   6 66.87 0.58610
## 13   6 70.46 0.53400
## 14   6 73.44 0.48660
## 15   7 76.21 0.44330
## 16   7 78.57 0.40400
## 17   7 80.53 0.36810
## 18   7 82.15 0.33540
## 19   7 83.50 0.30560
## 20   7 84.62 0.27840
## 21   7 85.55 0.25370
```

Get estimated coefficients for handpicked value

```
coef(fit, s = 0.1)
```

```
## 21 x 1 sparse Matrix of class "dgCMatrix"
##                       1
## (Intercept)  0.150928072
## V1           1.320597195
## V2           .
## V3           0.675110234
## V4           .
## V5          -0.817411518
## V6           0.521436671
## V7           0.004829335
## V8           0.319415917
## V9           .
## V10          .
## V11          0.142498519
## V12          .
## V13          .
## V14         -1.059978702
## V15          .
## V16          .
## V17          .
## V18          .
```

`glmnet` returns a sequence of models for the users to choose from, i.e. a model for every `lambda`.

How do we select the most appropriate model?

Use cross-validation to pick a `lambda` value. The default is 10-folds cross-validation.

```
cv_fit ← cv.glmnet(x, y)
```

We can pick the `lambda` that minimizes the cross-validation error.

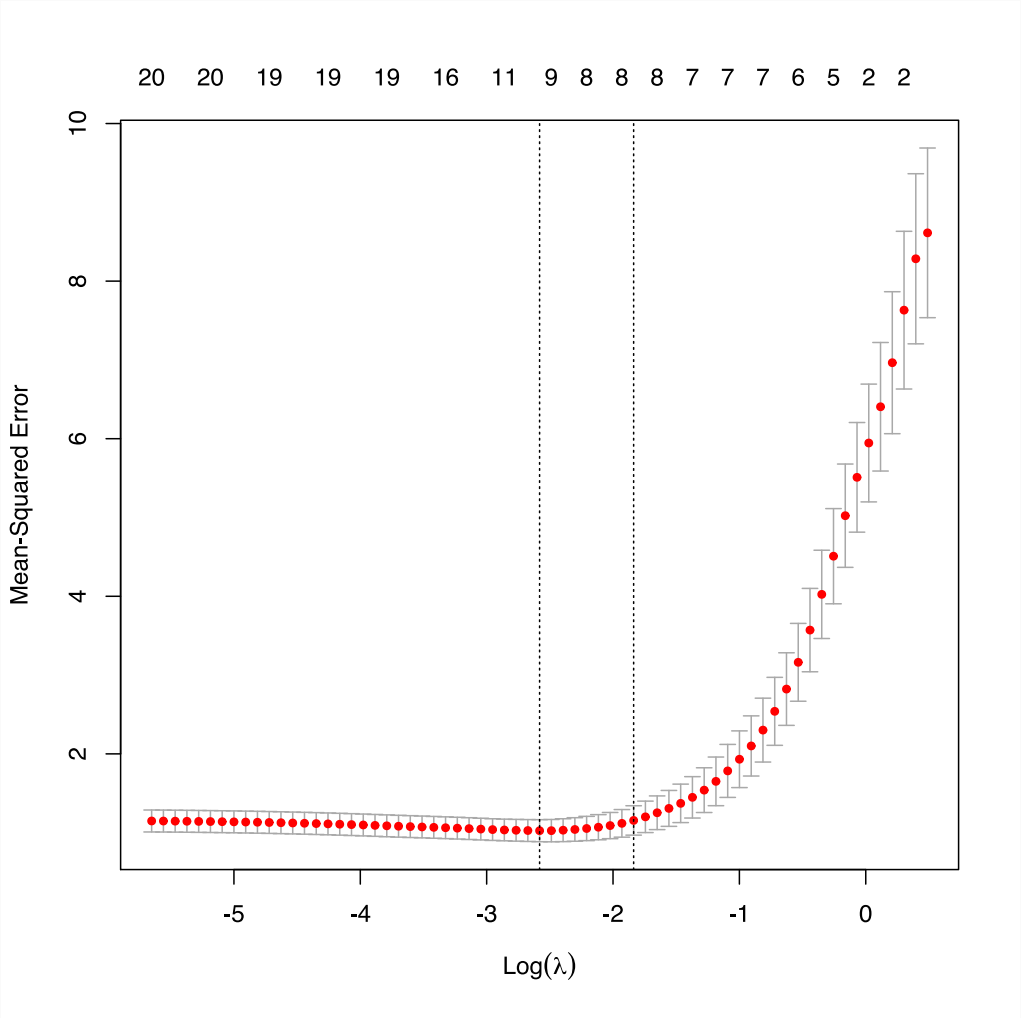
```
cv_fit$lambda.min
## [1] 0.07569327
```

Or we use the one-standard-error-rule.

```
cv_fit$lambda.1se
## [1] 0.1593271
```

We plot the cross-validation error for the inspected grid of `lambda` values.

```
plot(cv_fit)
```



For the selected `lambda` (via `cv_fit$lambda.min`) we inspect which parameters are non-zero (on the right).

Now, compare this to the selected variables obtained via `cv_fit$lambda.1se`.

```
coef(fit, s = cv_fit$lambda.min)
```

```
## 21 x 1 sparse Matrix of class "dgCMatrix"
##                       1
## (Intercept)  0.14867414
## V1           1.33377821
## V2           .
## V3           0.69787701
## V4           .
## V5          -0.83726751
## V6           0.54334327
## V7           0.02668633
## V8           0.33741131
## V9           .
## V10          .
## V11          0.17105029
## V12          .
## V13          .
## V14         -1.07552680
## V15          .
## V16          .
## V17          .
## V18          .
## V19          .
## V20         -1.05278699
```

The variables `V1`, `V3`, `V5-8`, `V11`, `V14` and `V20` are selected in the regression model.

However, the corresponding estimates (on the left) are biased, and shrunk to zero.

To remove this bias, we refit the model, only using the selected variables.

```
subset ← data.frame(y = y, V1 = x[, 1], V3 = x[, 3],
                    V5 = x[, 5], V6 = x[, 6],
                    V7 = x[, 7], V8 = x[, 8],
                    V11 = x[, 11], V14 = x[, 14],
                    V20 = x[, 20])
final_model ← lm(y ~ V1 + V3 + V5 + V6 + V7 + V8 +
                     V11 + V14 + V20, data = subset)
final_model %>% broom::tidy()
```

What is your judgement about `V7` (see coefficients on the right)?

What do you observe when comparing the estimates below with those shown on the previous sheet?

term	estimate	std.error	statistic	p.value
(Intercept)	0.1416891	0.0995658	1.4230704	0.1581730
V1	1.3746695	0.0968211	14.1980421	0.0000000
V3	0.7688247	0.0942568	8.1567012	0.0000000
V5	-0.8991610	0.1033747	-8.6980793	0.0000000
V6	0.6115910	0.0900882	6.7888025	0.0000000
V7	0.0947279	0.0972959	0.9736059	0.3328618
V8	0.3933822	0.0920456	4.2737767	0.0000477
V11	0.2600734	0.0994215	2.6158659	0.0104367
V14	-1.1239616	0.0885267	-12.6963039	0.0000000
V20	-1.1491267	0.1117142	-10.2863111	0.0000000

{glmnet} and the MTPL data set

Next, we fit a **Poisson regression model with lasso penalty** on the `mtpl` data set.

The regularization penalty helps us to select the interesting features from the data set.

`glmnet` requires the features as input matrix `x` and the target as a vector `y`.

Recall:

- `mtpl` has **continuous** features (e.g. `ageph`, `bm`, `power`)
- `mtpl` has **factor** variables with **two levels** (e.g. `sex`, `fleet`)
- but also factor variables with **more than 2 levels** (`coverage`)

Consider different types of coding factor variables.

Apply the `contrasts` function to the variable `coverage`

```
map(mtpl[, c("coverage")], contrasts,
    contrasts = FALSE)
## $coverage
##     FO PO TPL
## FO   1  0   0
## PO   0  1   0
## TPL  0  0   1
```

```
map(mtpl[, c("coverage")], contrasts,
    contrasts = TRUE)
## $coverage
##     PO TPL
## FO   0   0
## PO   1   0
## TPL  0   1
```

What's the difference?

{glmnet} and the MTPL data set (cont.)

We construct the input matrix for `glmnet`.

```
y ← mtpl$nclaims

x ← model.matrix( ~ coverage + fuel + use + fleet + sex + ageph + bm +
                    agec + power, data = mtpl,
                  contrasts.arg = map(mtpl[, c("coverage")], contrasts,
                                      contrasts = FALSE))[,-1]

x[1:10,]
```

Put the response or outcome variable in `y`.

In the `mtpl` data set we build a Poisson model for `nclaims`.

{glmnet} and the MTPL data set (cont.)

We construct the input matrix for `glmnet`.

```
y ← mtpl$nclaims

x ← model.matrix( ~ coverage + fuel + use + fleet + sex + ageph + bm +
                    agec + power, data = mtpl,
                    contrasts.arg = map(mtpl[, c("coverage")], contrasts,
                                        contrasts = FALSE))[,-1]
```

Use `model.matrix` to create the input matrix `x`.

We code the factor variable `coverage` with one-hot-encoding. Here, three dummy variables will be created for the three levels of `coverage`.

The other factor variables `fuel`, `use`, `fleet`, `sex` are dummy coded, with one dummy variable.

{glmnet} and the MTPL data set (cont.)

We construct the input matrix for `glmnet`.

```
y ← mtpl$nclaims

x ← model.matrix( ~ coverage + fuel + use + fleet + sex + ageph + bm +
                    agec + power, data = mtpl,
                    contrasts.arg = map(mtpl[, c("coverage")], contrasts,
                                        contrasts = FALSE))[,-1]
```

Use `model.matrix` to create the input matrix `x`.

We remove the first column, representing the intercept, from the `model.matrix`.

{glmnet} and the MTPL data set (cont.)

Let's check the input matrix `x`

```
##   coverageFO coveragePO coverageTPL fuelgasoline usework fleetY sexmale ageph bm agec
## 1          0          0           1            1       0      0       1    50  5   12
## 2          0          1           0            1       0      0       0    64  5    3
## 3          0          0           1            0       0      0       1    60  0   10
## 4          0          0           1            1       0      0       1    77  0   15
## 5          0          0           1            1       0      0       0    28  9    7
## 6          0          0           1            1       0      0       1    26 11   12
##   power
## 1    77
## 2    66
## 3    70
## 4    57
## 5    70
## 6    70
```

You are now ready to fit a regularized Poisson GLM for `y` with input `x`.

Let's go!

Your turn

You will fit a regularized Poisson GLM on the `mtpl` data with the {glmnet} package.

Q: using the constructed `y` and `x`

1. Fit a `glmnet` with lasso penalty and store the fitted object in `mtpl_glmnet`. Use the following arguments `family = "poisson", offset = ___`.
2. Display the order of the variables and their names via `row.names(mtpl_glmnet$beta)`.
3. Plot the solutions path. Pick a meaningful value for `lambda` via cross-validation.
4. Which variables are selected in the lasso model? As a last step, you will fit a Poisson GLM with the selected variables. What do you see?
5. List some pros and cons of the above strategy.

Q.1 fit a regularized Poisson GLM

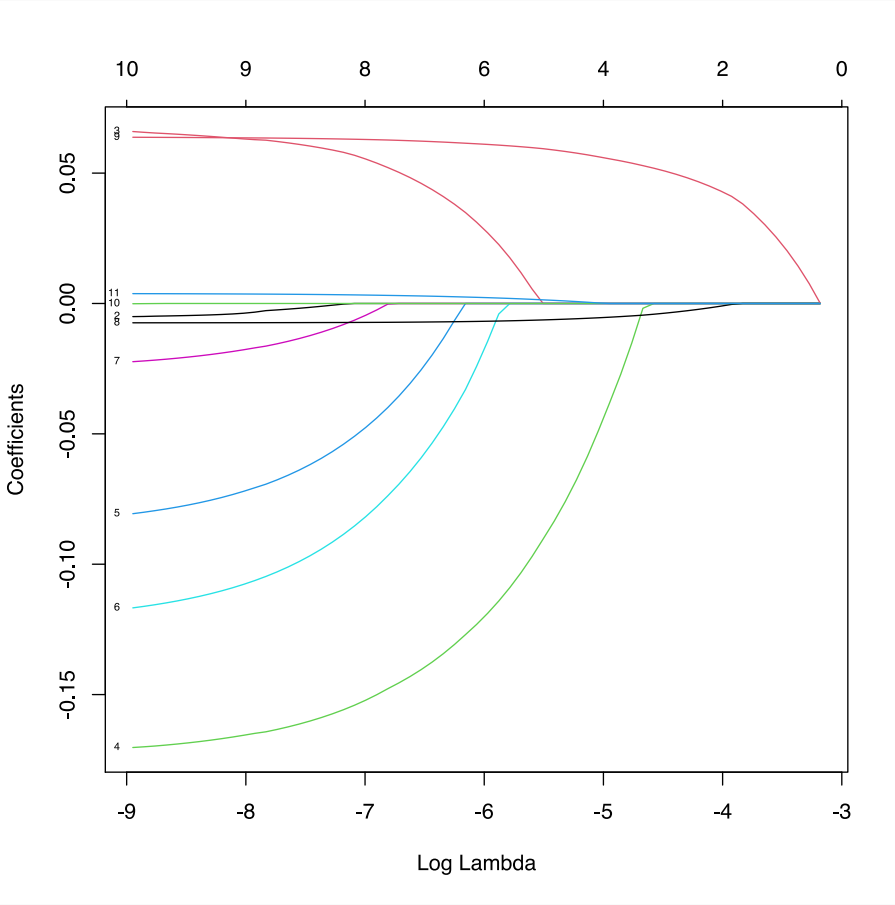
```
alpha ← 1 # for lasso penalty
mtpl_glmnet ← glmnet(x = x, y = y,
                     family = "poisson",
                     offset = log(mtpl$expo),
                     alpha = alpha,
                     standardize = TRUE,
                     intercept = TRUE)
```

Q.2 display the variables via

```
row.names(mtpl_glmnet$beta)
##  [1] "coverageFO"   "coveragePO"   "coverageTPL"  "f
##  [6] "fleetY"       "sexmale"      "ageph"        "b
## [11] "power"
```

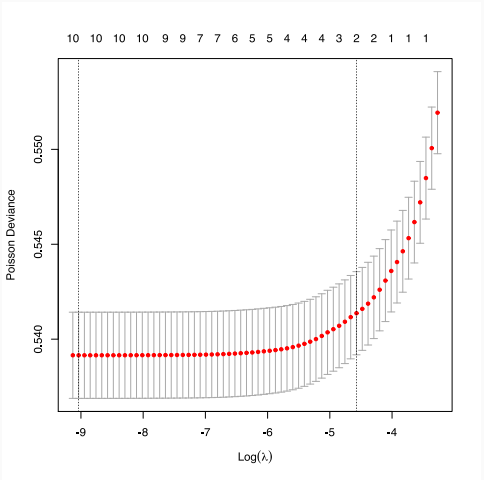
Q.3 plot the solutions path

```
plot(mtpl_glmnet, xvar = 'lambda', label = TRUE)
```



Q.3 pick a value for `lambda`

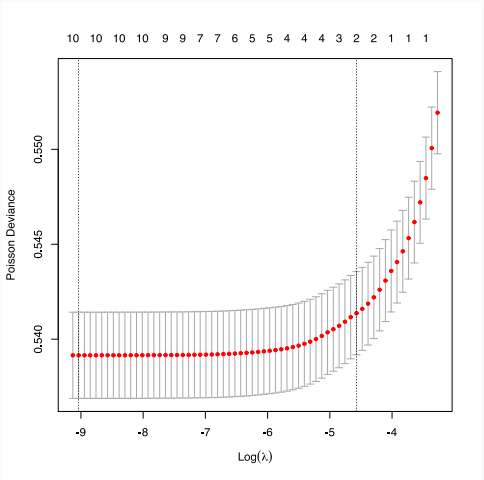
```
set.seed(123)
fold_id ← sample(rep(1:10, length.out = nrow(mtpl)),
                 nrow(mtpl))
mtpl_glmnet_cv ← cv.glmnet(x, y, family = "poisson",
                           alpha = alpha,
                           nfolds = 10,
                           foldid = fold_id,
                           type.measure = "deviance",
                           standardize = TRUE,
                           intercept = TRUE)
plot(mtpl_glmnet_cv)
```




```
coef(mtpl_glmnet_cv, s = "lambda.min")
## 12 x 1 sparse Matrix of class "dgCMatrix"
##                            1
## (Intercept)  -2.106680933
## coverageFO   -0.006499730
## coveragePO    .
## coverageTPL   0.050002173
## fuelgasoline -0.165864612
## usework      -0.069292342
## fleetY       -0.049283838
## sexmale      -0.013718073
## ageph        -0.006347490
## bm            0.058564280
## agec         -0.002004356
## power         0.003448081
```

Q.3 pick a value for `lambda`

```
set.seed(123)
fold_id ← sample(rep(1:10, length.out = nrow(mtpl)),
                 nrow(mtpl))
mtpl_glmnet_cv ← cv.glmnet(x, y, family = "poisson",
                           alpha = alpha,
                           nfolds = 10,
                           foldid = fold_id,
                           type.measure = "deviance",
                           standardize = TRUE,
                           intercept = TRUE)
plot(mtpl_glmnet_cv)
```




```
coef(mtpl_glmnet_cv, s = "lambda.1se")
## 12 x 1 sparse Matrix of class "dgCMatrix"
##                             1
## (Intercept)  -2.124039910
## coverageFO    .
## coveragePO    .
## coverageTPL   .
## fuelgasoline  .
## usework       .
## fleetY        .
## sexmale       .
## ageph        -0.002916928
## bm            0.046163778
## agec          .
## power         .
```

Q.4 refit the models using only the selected features

```
mtpl$coverage ← relevel(mtpl$coverage, "P0")
mtpl_formula_refit ← nclaims ~ 1 + coverage +
                     fuel + use + fleet + sex +
                     ageph + bm + agec + power
mtpl_glm_refit ← glm(mtpl_formula_refit,
                     data = mtpl,
                     offset = log(mtpl$expo),
                     family = poisson())
```

The selection obtained via `lambda.min`

term	estimate	std.error	statistic	p.value
(Intercept)	-1.9892872	0.0401325	-49.5679730	0.0000000
coverageFO	0.0044293	0.0244274	0.1813238	0.8561134
coverageTPL	0.0743796	0.0172363	4.3152799	0.0000159
fuelgasoline	-0.1731052	0.0153266	-11.2944557	0.0000000
usework	-0.0862841	0.0334470	-2.5797233	0.0098880
fleetY	-0.1226498	0.0435289	-2.8176618	0.0048375
sexmale	-0.0253198	0.0162468	-1.5584505	0.1191265
ageph	-0.0074262	0.0005391	-13.7764864	0.0000000
bm	0.0639249	0.0017328	36.8902457	0.0000000
agec	-0.0004698	0.0019368	-0.2425874	0.8083251
power	0.0038535	0.0003799	10.1421096	0.0000000

Q.4 refit the models using only the selected features

```
mtpl_formula_refit_2 ← nclaims ~ 1 + ageph + bm
mtpl_glm_refit_2 ← glm(mtpl_formula_refit_2,
                       data = mtpl,
                       offset = log(mtpl$expo),
                       family = poisson())
```

The selection obtained via `lambda.1se`

term	estimate	std.error	statistic	p.value
(Intercept)	-1.8251292	0.0282345	-64.64189	0
ageph	-0.0083839	0.0005274	-15.89605	0
bm	0.0625774	0.0017141	36.50764	0

Thanks!

Slides created with the R package xaringan.

Course material available via

https://github.com/katrienantonio/hands-on-machine-learning-R-module-1