

# Hands-on Machine Learning with R - Module 2

Hands-on webinar

---

Katrien Antonio & Roel Henckaerts

[hands-on-machine-learning-R-module-2](#) | January 14 & 21, 2021

# Prologue

---

# Introduction

## Course


 <https://github.com/katrienantonio/hands-on-machine-learning-R-module-2>

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](mailto:katrien.antonio@kuleuven.be) & [roel.henckaerts@kuleuven.be](mailto: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)?

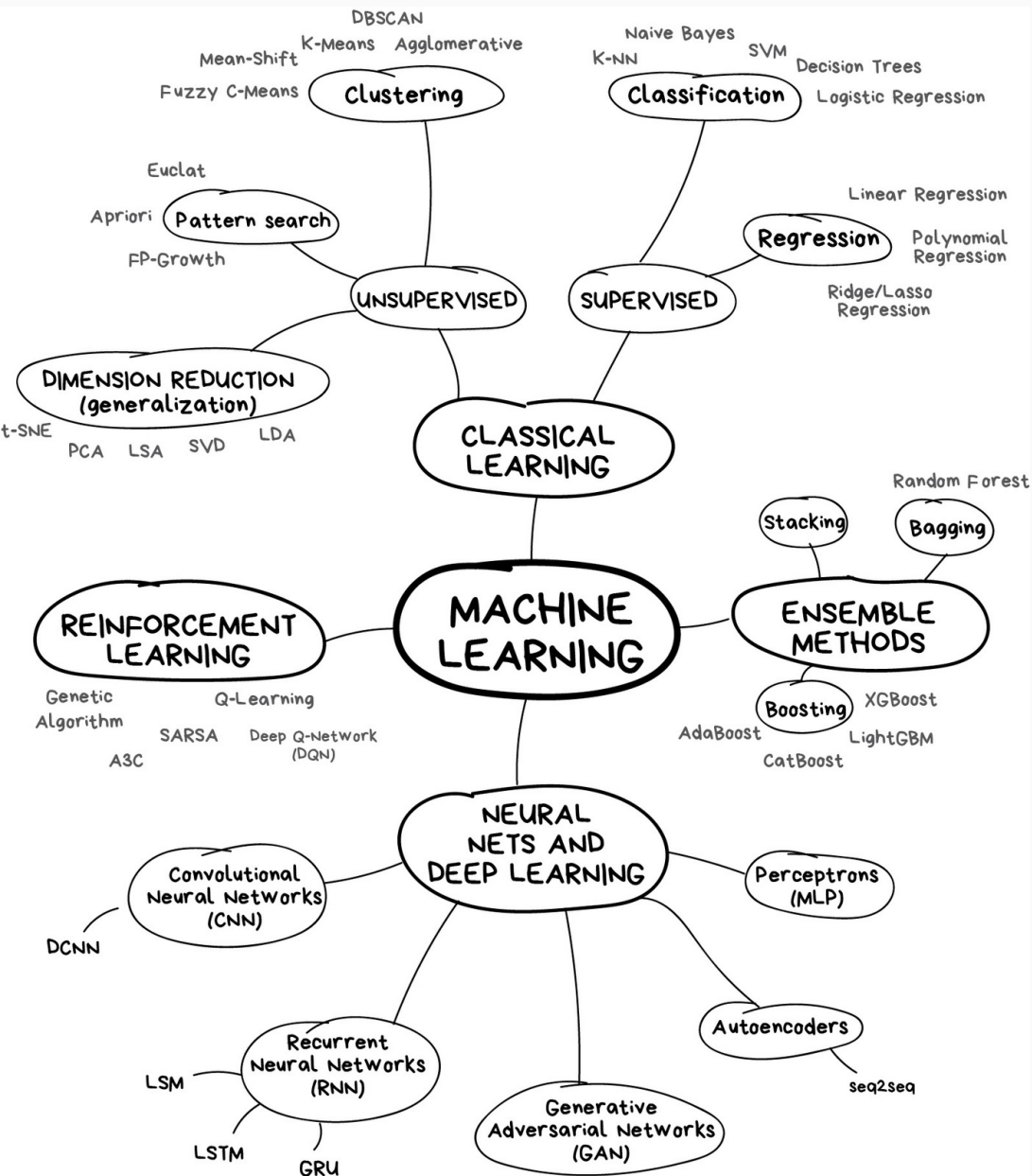
# Why this course?

## The goals of this module

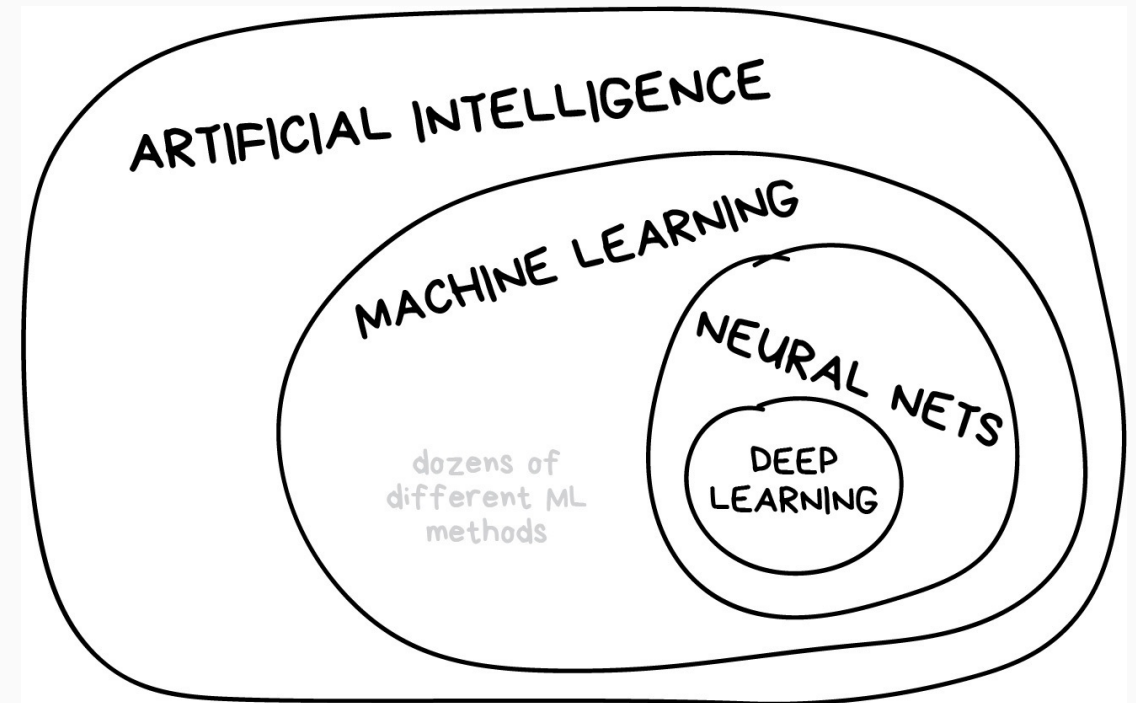
- develop foundations of working with **regression and decision trees**
- step from simple trees to ensembles of trees, with **bagging** and **boosting**
- focus on the use of these ML methods for the **analysis of frequency + severity data**
- discuss and construct some useful **interpretation tools**, e.g. variable importance plots, partial dependence plots.

# Module 2's Outline

- Prologue
- Decision tree
  - what is tree-based machine learning?
  - tree basics: structure, terminology, growing process
  - using {rpart}
  - pruning via cross-validation
  - examples on regression and classification
  - modelling claim frequency and severity data with trees
- Interpretation tools
  - feature importance
  - partial dependence plot
  - the {vip} and {pdp} packages
- Bagging
  - from a single tree to Bootstrap Aggregating
  - out-of-bag error
- Random forest
  - from bagging to random forests
  - tuning
- Gradient boosting
  - (stochastic) gradient boosting with trees
  - training process and tuning parameters
  - using {gbm}
  - modelling claim frequencies and severities
  - using {xgboost}



Some roadmaps to explore the ML landscape...



Source: [Machine Learning for Everyone In simple words. With real-world examples. Yes, again.](#)

# Background reading



Henckaerts et al. (2020) paper on [Boosting insights in insurance tariff plans with tree-based machine learning methods](#)

- full algorithmic details of regression trees, bagging, random forests and gradient boosting machines
- with focus on claim frequency and severity modelling
- including interpretation tools (VIP, PDP, ICE, H-statistic)
- model comparison (GLMs, GAMs, trees, RFs, GBMs)
- managerial tools (e.g. loss ratio, discrimination power).

The paper comes with two notebooks, see [examples tree-based paper](#) and [severity modelling](#).

The paper comes with an R package for fitting random forests on insurance data, see [distRforest](#).



# What is tree-based machine learning?

Machine learning (ML) according to [Wikipedia](#):

*"Machine learning algorithms build a **mathematical model** based on sample data, known as training data, in order to make predictions or decisions without being explicitly programmed to perform the task."*

This definition goes all the way back to [Arthur Samuel](#), who coined the term "machine learning" in 1959.

**Tree-based ML** makes use of a **tree** as building block for the mathematical model.



Single tree



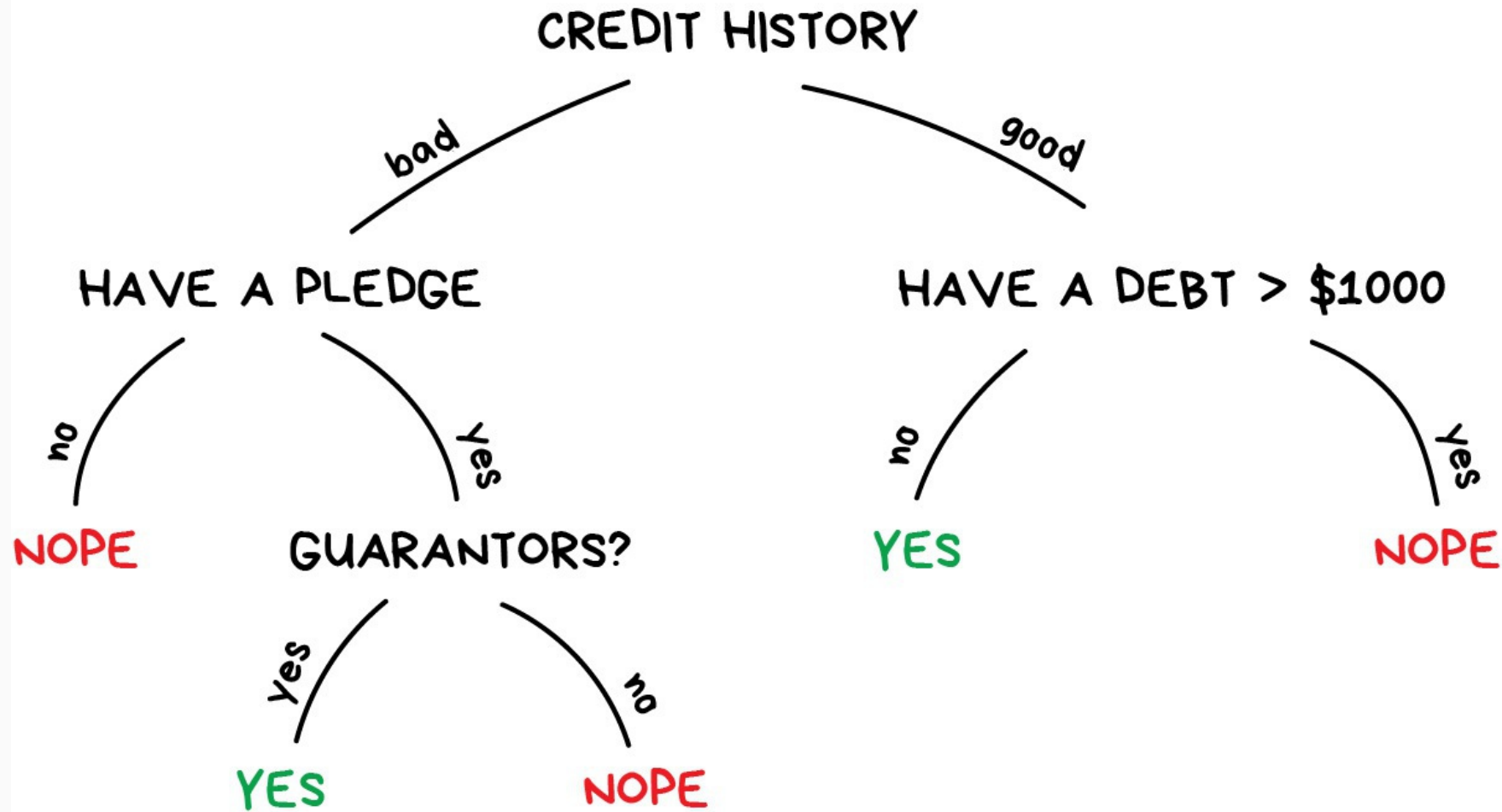
Ensemble of trees

So, a natural question to start from is: what is a **tree**?

# Tree basics

---

# GIVE A LOAN?



DECISION TREE

# Tree structure and terminology

The top of the tree contains all available training observations: the **root node**.

We **partition** the data into homogeneous non-overlapping subgroups: the **nodes**

We create subgroups via **simple yes-no questions**.

A tree then predicts the output in a **leaf node** as follows:

- average of the response for regression
- majority voting for classification

# Tree structure and terminology

The top of the tree contains all available training observations: the **root node**.

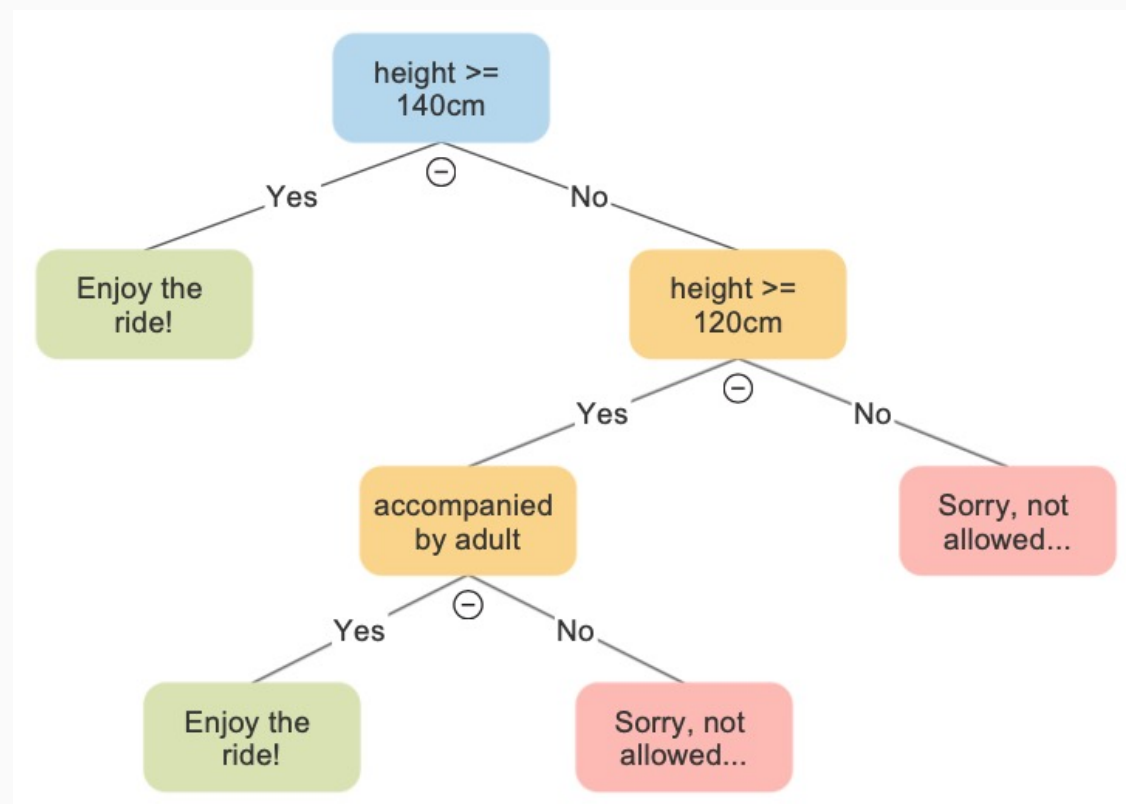
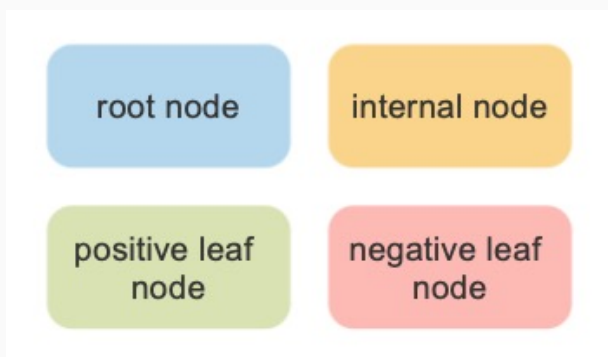
We **partition** the data into homogeneous non-overlapping subgroups: the **nodes**.

We create subgroups via **simple yes-no questions**.

A tree then predicts the output in a **leaf node** as follows:

- average of the response for regression
- majority voting for classification.

Different types of nodes:



# Tree growing process

A golden standard is the Classification And Regression Tree algorithm: **CART** (Breiman et al., 1984).

CART uses **binary recursive partitioning** to split the data in subgroups.

In each node, we search for the best feature to partition the data into two regions:  $R_1$  and  $R_2$  (hence, **binary**).

## Take-away - what is **best**?

Minimize the **overall loss** between observed responses and leaf node prediction


- overall loss = loss in region  $R_1$  + loss in region  $R_2$
- for regression: mean squared or absolute error, deviance,...
- for classification: cross-entropy, Gini index,...

After splitting the data, this process is repeated for region  $R_1$  and  $R_2$  separately (hence, **recursive**).

Repeat until **stopping criterion** is satisfied, e.g., maximum depth of a tree or minimum loss improvement.

# Using {rpart}

```
rpart(formula, data, method,  
      control = rpart.control(cp, maxdepth, minsplit, minbucket))
```

- `formula`: a formula as *response ~ feature1 + feature2 + ...*  no need to include the interactions!
- `data`: the observation data containing the response and features
- `method`: a string specifying which **loss function** to use
  - "anova" for regression (SSE as loss)
  - "class" for classification (Gini as loss)
  - "poisson" for Poisson regression (Poisson deviance as loss, see more later)
- `cp`: complexity parameter specifying the proportion by which the overall error should improve for a split to be attempted
- `maxdepth`: the maximum depth of the tree
- `minsplit`: minimum number of observations in a node for a split to be attempted
- `minbucket`: minimum number of observations in a leaf node.

# Toy example of a regression tree

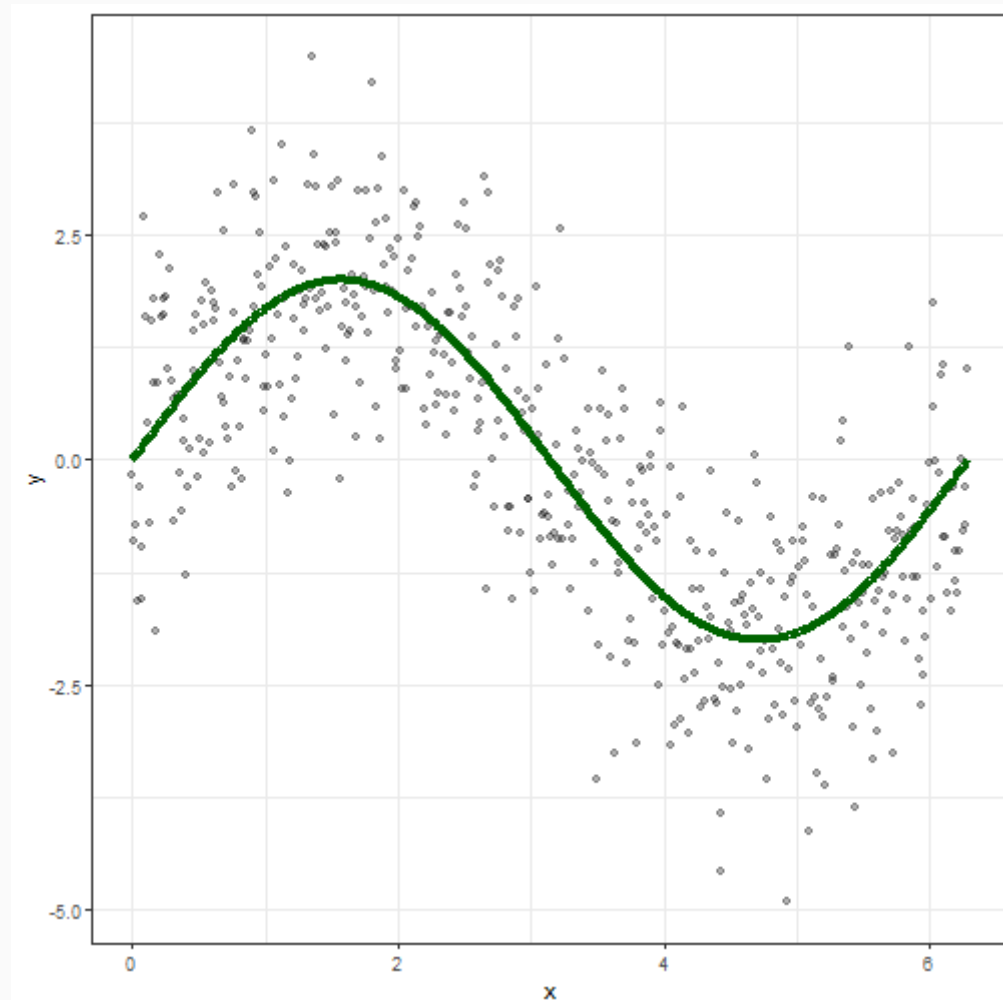
---



# Simulated data

```
library(tidyverse)
set.seed(54321) # reproducibility
dfr <- tibble::tibble(
  x = seq(0, 2 * pi, length.out = 500),
  m = 2 * sin(x),
  y = m + rnorm(length(x), sd = 1)
)
```

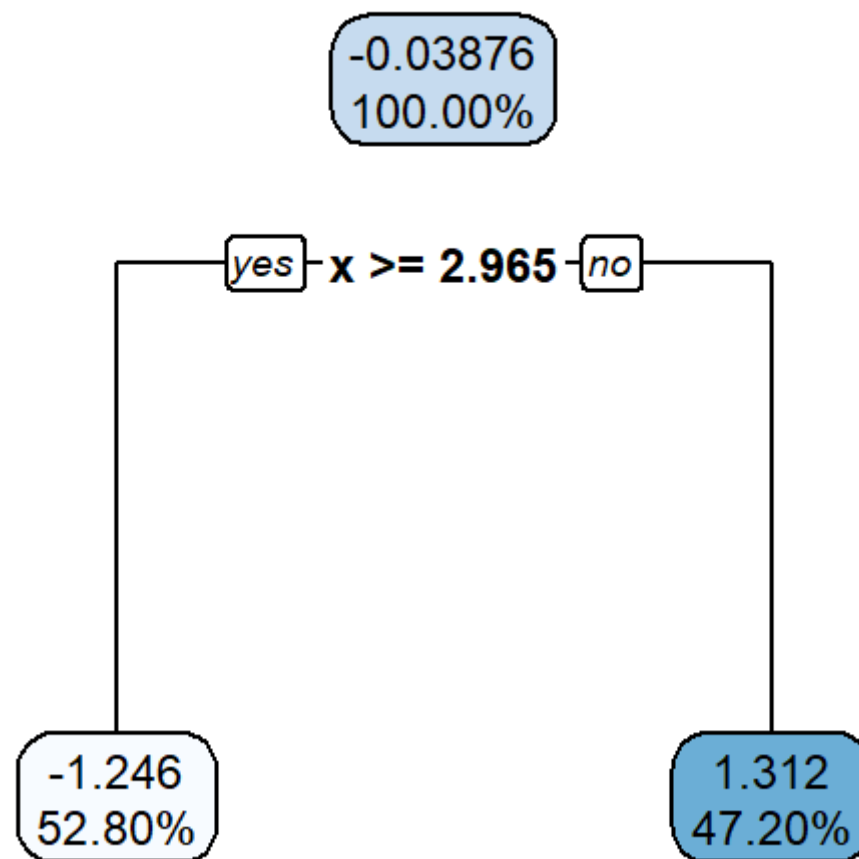
##		x	m	y
## 1		0.000000000	0.000000000	-0.1789007
## 2		0.01259155	0.02518244	-0.9028617
## 3		0.02518311	0.05036089	-0.7336728
## 4		0.03777466	0.07553136	-1.5750691
## 5		0.05036621	0.10068985	-0.3073767
## 6		0.06295777	0.12583237	-0.9696970
## 7		0.07554932	0.15095495	-1.5412872
## 8		0.08814088	0.17605359	2.6920994
## 9		0.10073243	0.20112432	1.5964765
## 10		0.11332398	0.22616316	0.4061405



# Decision stump - a tree with only one split

```
library(rpart)
fit <- rpart(formula = y ~ x,
             data = dfr,
             method = 'anova',
             control = rpart.control(
               maxdepth = 1
             )
             )
print(fit)
## n= 500
##
## node), split, n, deviance, yval
##      * denotes terminal node
##
## 1) root 500 1498.4570 -0.03876172
##   2) x ≥ 2.965311 264 384.3336 -1.24604800 *
##   3) x < 2.965311 236 298.8888 1.31176200 *
```

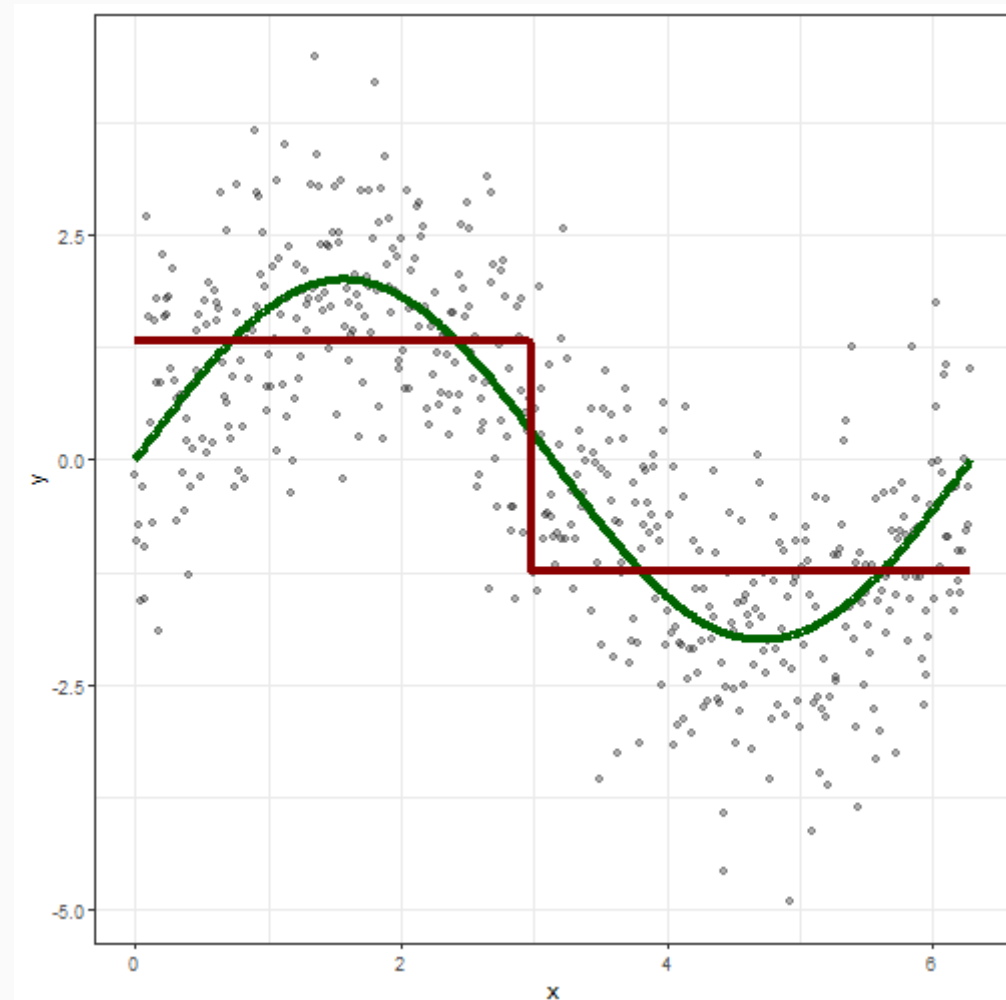
```
library(rpart.plot) # for nice plots
rpart.plot(fit, digits = 4, cex = 2)
```



# Decision stump - a tree with only one split

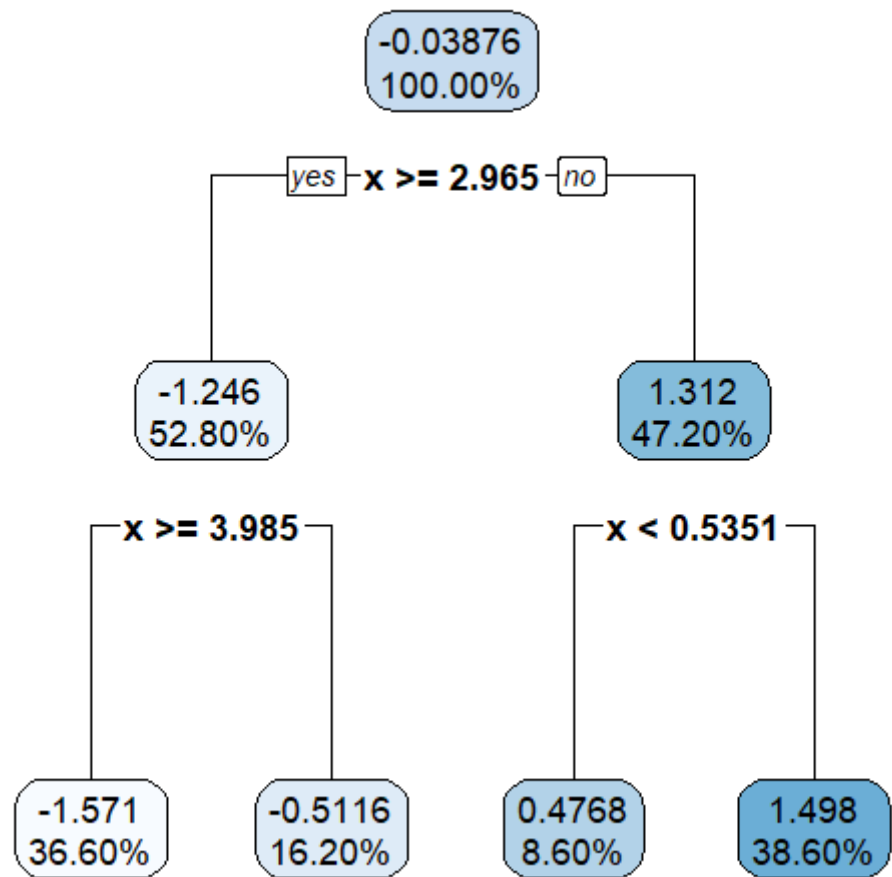
```
fit <- rpart(formula = y ~ x,  
             data = dfr,  
             method = 'anova',  
             control = rpart.control(  
               maxdepth = 1  
             )  
          )  
  
print(fit)  
## n= 500  
##  
## node), split, n, deviance, yval  
##      * denotes terminal node  
##  
## 1) root 500 1498.4570 -0.03876172  
##   2) x ≥ 2.965311 264 384.3336 -1.24604800 *  
##   3) x < 2.965311 236 298.8888 1.31176200 *
```

```
# Get predictions via the predict function  
pred <- predict(fit, dfr)
```



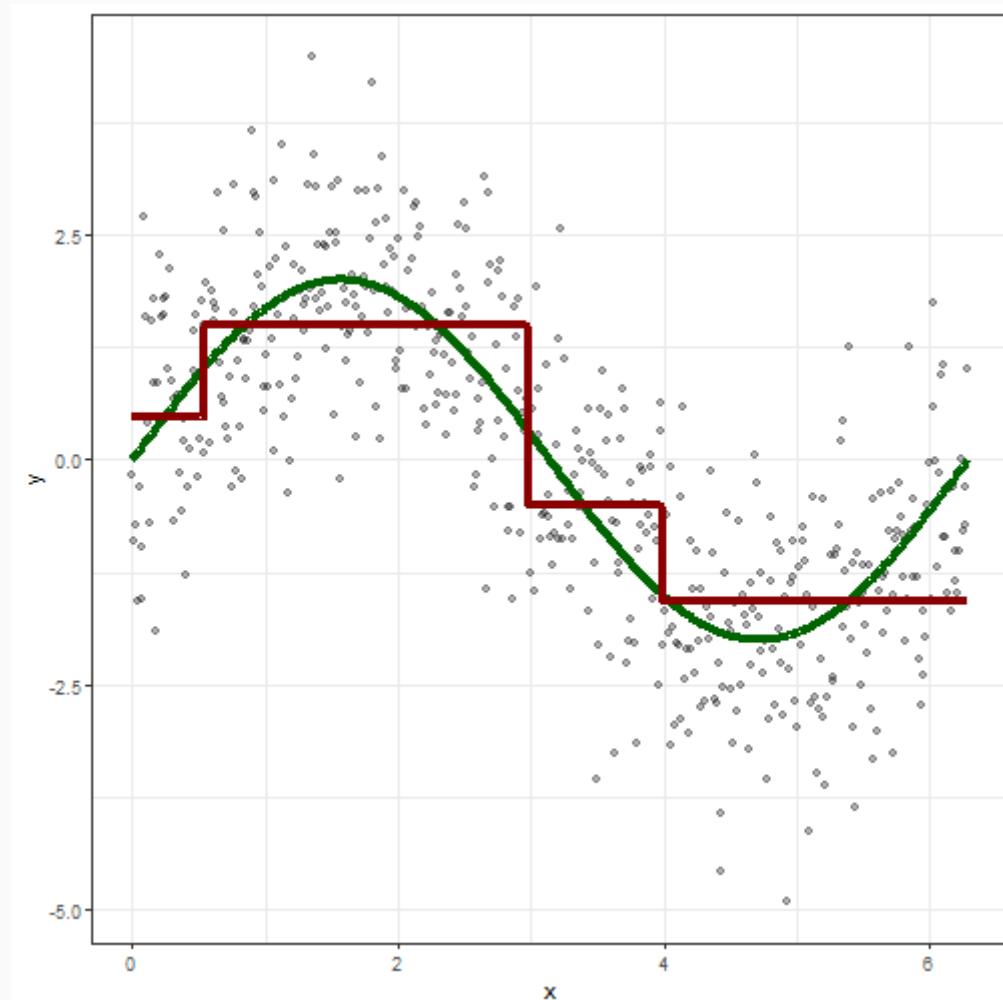
# Adding splits

```
fit <- rpart(formula = y ~ x,  
             data = dfr,  
             method = 'anova',  
             control = rpart.control(  
               maxdepth = 2  
             )  
           )  
print(fit)  
## n= 500  
##  
## node), split, n, deviance, yval  
##      * denotes terminal node  
##  
## 1) root 500 1498.45700 -0.03876172  
##    2) x ≥ 2.965311 264 384.33360 -1.24604800  
##      4) x ≥ 3.985227 183 228.44490 -1.57111200 *  
##      5) x < 3.985227 81 92.86428 -0.51164310 *  
##    3) x < 2.965311 236 298.88880 1.31176200  
##      6) x < 0.535141 43 55.23637 0.47680020 *  
##      7) x ≥ 0.535141 193 206.99550 1.49779000 *
```



# Adding splits (cont.)

```
fit <- rpart(formula = y ~ x,
             data = dfr,
             method = 'anova',
             control = rpart.control(
               maxdepth = 2
             )
             )
print(fit)
## n= 500
##
## node), split, n, deviance, yval
##      * denotes terminal node
##
## 1) root 500 1498.45700 -0.03876172
##    2) x ≥ 2.965311 264 384.33360 -1.24604800
##      4) x ≥ 3.985227 183 228.44490 -1.57111200 *
##      5) x < 3.985227 81 92.86428 -0.51164310 *
##    3) x < 2.965311 236 298.88880 1.31176200
##      6) x < 0.535141 43 55.23637 0.47680020 *
##      7) x ≥ 0.535141 193 206.99550 1.49779000 *
```





## Your turn

Let's get familiar with the structure of a decision tree.

**Q:** choose one of the trees from the previously discussed examples and pick a leaf node, but keep it simple for now.

1. Replicate the **predictions** for that leaf node, based on the split(s) and the training data.
2. Replicate the **deviance** measure for that leaf node, based on the split(s), the training data and your predictions from Q1.

Hint: the deviance used in an anova {rpart} tree is the **Sum of Squared Errors (SSE)**:

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

Take for example the tree with depth two:

```
print(fit)
## n= 500
##
## node), split, n, deviance, yval
##      * denotes terminal node
##
## 1) root 500 1498.45700 -0.03876172
##    2) x ≥ 2.965311 264 384.33360 -1.24604800
##      4) x ≥ 3.985227 183 228.44490 -1.57111200 *
##      5) x < 3.985227 81 92.86428 -0.51164310 *
##    3) x < 2.965311 236 298.88880 1.31176200
##      6) x < 0.535141 43 55.23637 0.47680020 *
##      7) x ≥ 0.535141 193 206.99550 1.49779000 *
```

Let's predict the values for leaf node 6.

**Q.1:** calculate the prediction

```
# Subset observations in node 6
obs ← dfr %>% dplyr::filter(x < 0.535141)

# Predict
pred ← obs$y %>% mean
pred
## [1] 0.4768002
```

**Q.2:** calculate the deviance

```
# Deviance
dev ← (obs$y - pred)^2 %>% sum
dev
## [1] 55.23637
```

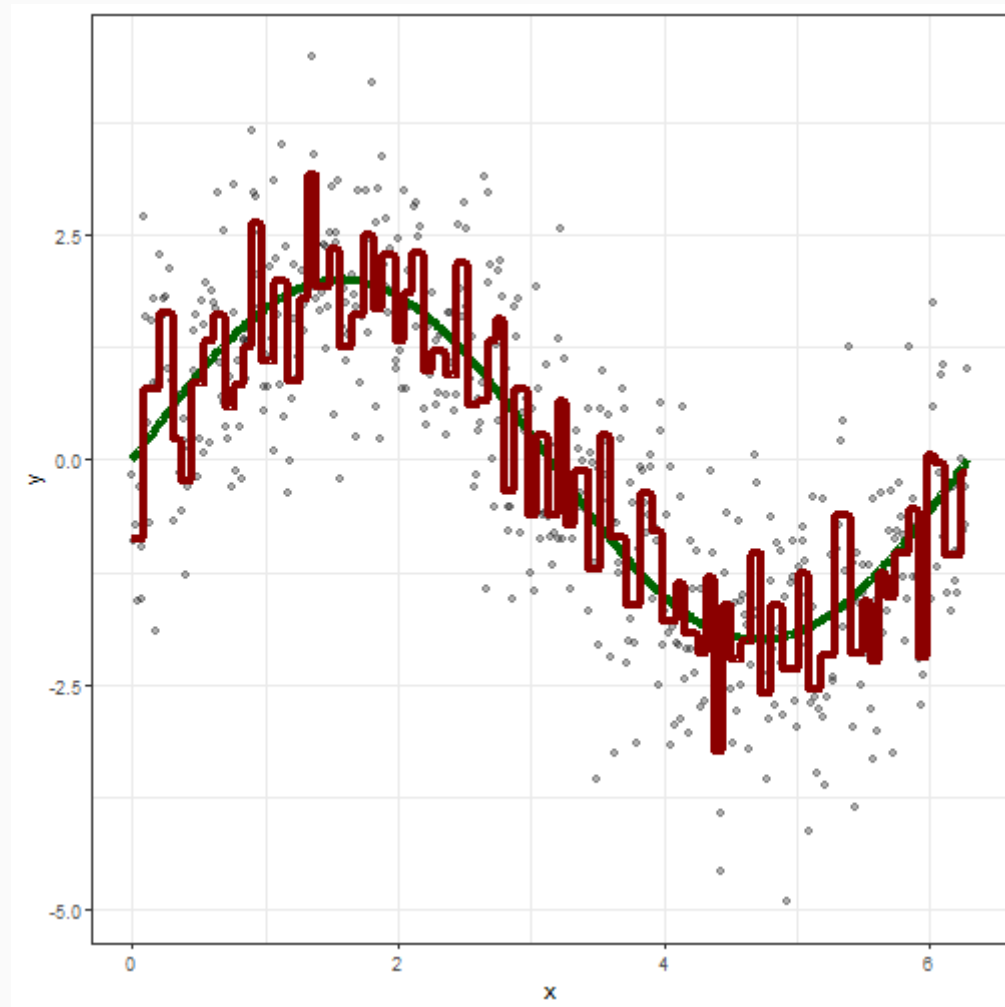
# A very deep tree

```
fit <- rpart(formula = y ~ x,  
             data = dfr,  
             method = 'anova',  
             control = rpart.control(  
               maxdepth = 20,  
               minsplit = 10,  
               minbucket = 5,  
               cp = 0  
             )  
)
```

**Take-away**  - understanding the `cp`

parameter:

- unitless in {rpart} (different from original CART)
- `cp = 1` returns a **root node**, without splits
- `cp = 0` returns the **deepest tree possible**, allowed by the other stopping criteria.

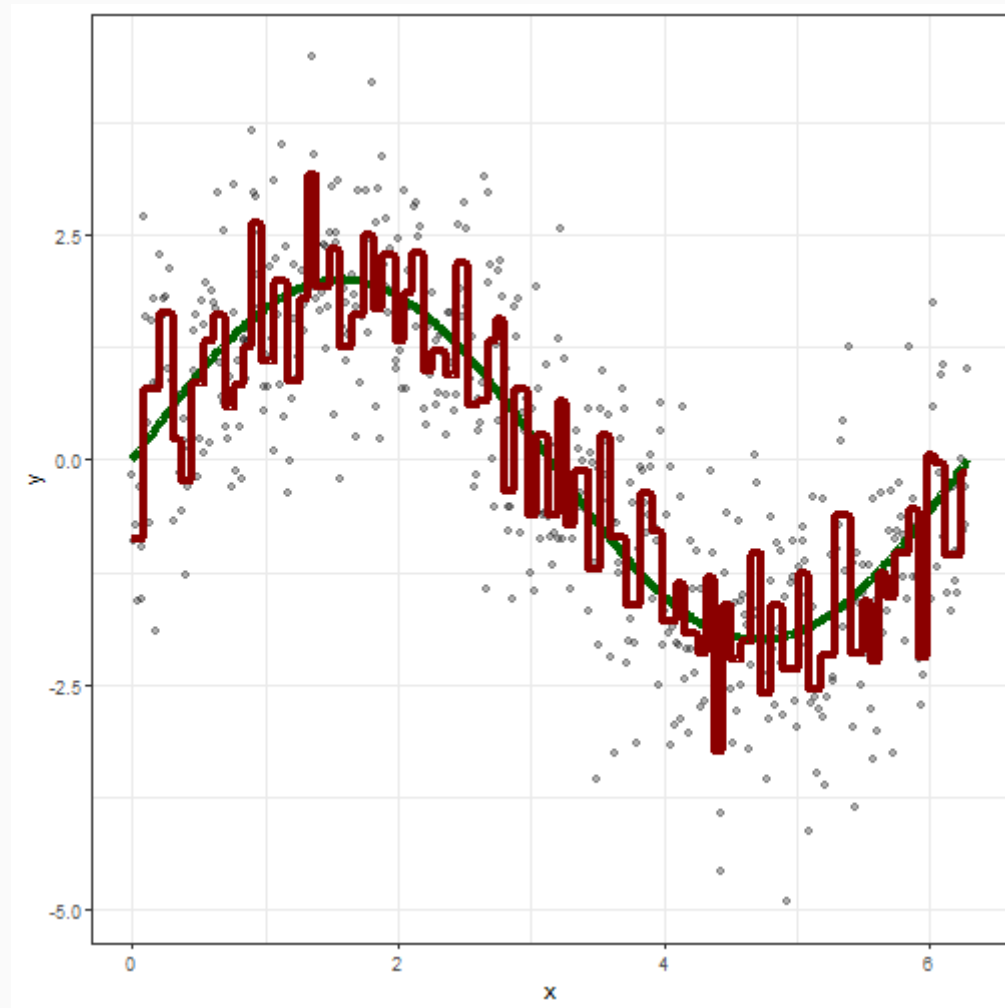




# A very deep tree (cont.)

```
fit <- rpart(formula = y ~ x,  
             data = dfr,  
             method = 'anova',  
             control = rpart.control(  
               maxdepth = 20,  
               minsplit = 10,  
               minbucket = 5,  
               cp = 0  
             )  
)
```

What is your opinion on the tree shown on the right?







# Pruning via cross-validation in {rpart}

---

# How deep should a tree be?

The **bias-variance trade off**:

- a **shallow** tree will underfit:  
bias  and variance 
- a **deep** tree will overfit:  
bias  and variance 
- find right **balance** between bias and variance!





Typical approach to get the right fit:

- fit an overly complex **deep tree**
- **prune** the tree to find the **optimal subtree**.

How to **prune**?

# How deep should a tree be?

The **bias-variance trade off**:

- a **shallow** tree will underfit:  
bias  and variance 
- a **deep** tree will overfit:  
bias  and variance 
- find right **balance** between bias and variance!

Typical approach to get the right fit:

- fit an overly complex **deep tree**
- **prune** the tree to find the **optimal subtree**.

How to **prune**?

Look for the smallest subtree that minimizes a **penalized loss function**:

$$\min\{f_{\text{loss}} + \alpha \cdot |T|\}$$

- loss function  $f_{\text{loss}}$
- complexity parameter  $\alpha$
- number of leaf nodes  $|T|$ .

A shallow tree results when  $\alpha$  is large and a deep tree when  $\alpha$  is small.

Perform **cross-validation** on the complexity parameter:

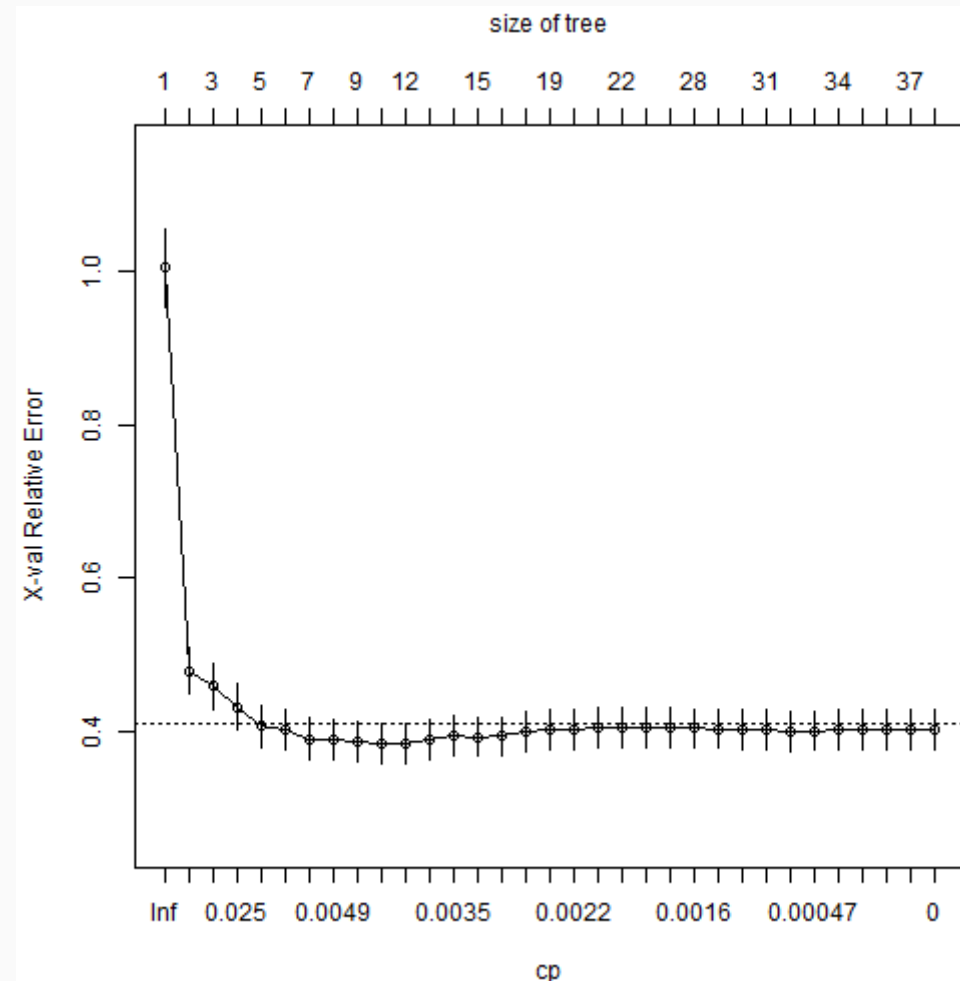
- `cp` is the complexity parameter in `{rpart}`
- `cp` is  $\alpha$  divided by  $f_{\text{loss}}$  evaluated in root node.

Cfr. tuning of the regularization parameter in lasso from `{glmnet}` in **Module 1**.

# Pruning via cross-validation

```
set.seed(87654) # reproducibility
fit <- rpart(formula = y ~ x,
             data = dfr,
             method = 'anova',
             control = rpart.control(
               maxdepth = 10,
               minsplit = 20,
               minbucket = 10,
               cp = 0,
               xval = 5
             )
)
```

```
# Plot the cross-validation results
plotcp(fit)
```



# Pruning via cross-validation

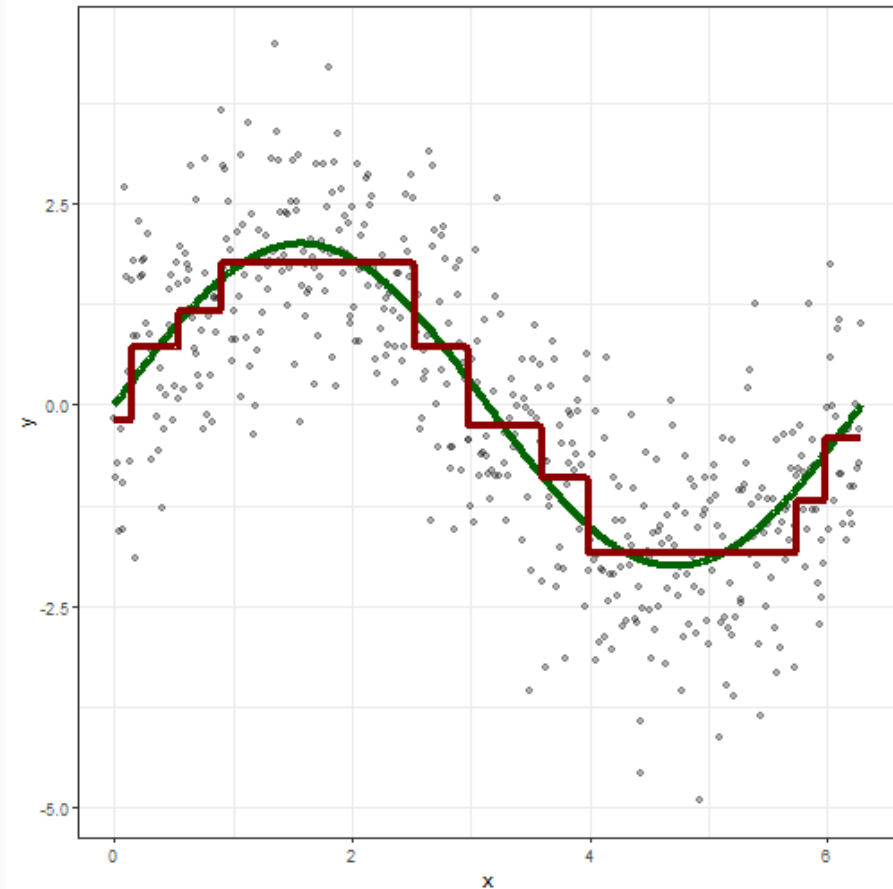
```
set.seed(87654) # reproducibility
fit <- rpart(formula = y ~ x,
             data = dfr,
             method = 'anova',
             control = rpart.control(
               maxdepth = 10,
               minsplit = 20,
               minbucket = 10,
               cp = 0,
               xval = 5
             )
# Get xval results via 'cptable' attribute
cpt <- fit$cptable

print(cpt[1:20,])
# Which cp value do we choose?
min_xerr <- which.min(cpt[, 'xerror'])
se_rule <- min(which(cpt[, 'xerror'] <
  (cpt[min_xerr, 'xerror'] + cpt[min_xerr, 'xstd'])))
```

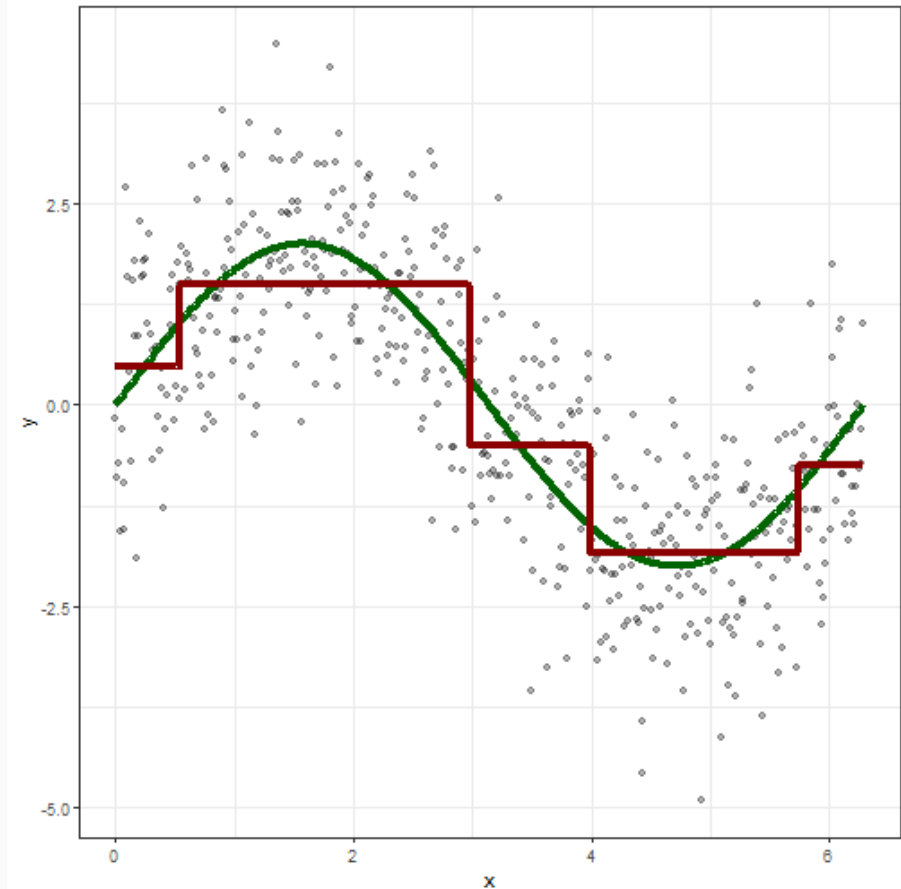
##		CP	nsplit	rel error	xerror	xstd
## 1	0.54404922	0	1.000000	1.004726	0.0514072	
## 2	0.04205955	1	0.455951	0.479691	0.0306899	
## 3	0.02638545	2	0.413891	0.459565	0.0303987	
## 4	0.02446313	3	0.387506	0.432619	0.0288631	
## 5	0.01686947	4	0.363043	0.407090	0.0271596	
## 6	0.00556730	5	0.346173	0.402555	0.0269263	
## 7	0.00537029	6	0.340606	0.390939	0.0263032	
## 8	0.00455035	7	0.335236	0.389550	0.0259170	
## 9	0.00438010	8	0.330685	0.387857	0.0262972	
## 10	0.00437052	9	0.326305	0.384689	0.0262569	
## 11	0.00417651	11	0.317564	0.384689	0.0262569	
## 12	0.00413572	12	0.313388	0.389304	0.0264134	
## 13	0.00288842	13	0.309252	0.394634	0.0263896	
## 14	0.00248513	14	0.306363	0.393097	0.0255738	
## 15	0.00230656	16	0.301393	0.394084	0.0254549	
## 16	0.00227479	17	0.299087	0.401089	0.0260820	
## 17	0.00222192	18	0.296812	0.403132	0.0258395	
## 18	0.00218218	19	0.294590	0.403132	0.0258395	
## 19	0.00189012	20	0.292408	0.405123	0.0258289	
## 20	0.00177060	21	0.290518	0.405770	0.0258239	

# Minimal CV error or 1 SE rule

```
fit_1 ← prune(fit, cp = cpt[min_xerr, 'CP'])
```



```
fit_2 ← prune(fit, cp = cpt[se_rule, 'CP'])
```





## Your turn

**Q:** Trees are often associated with **high variance**, meaning that the resulting model can be very sensitive to the input data. Let's explore this statement!

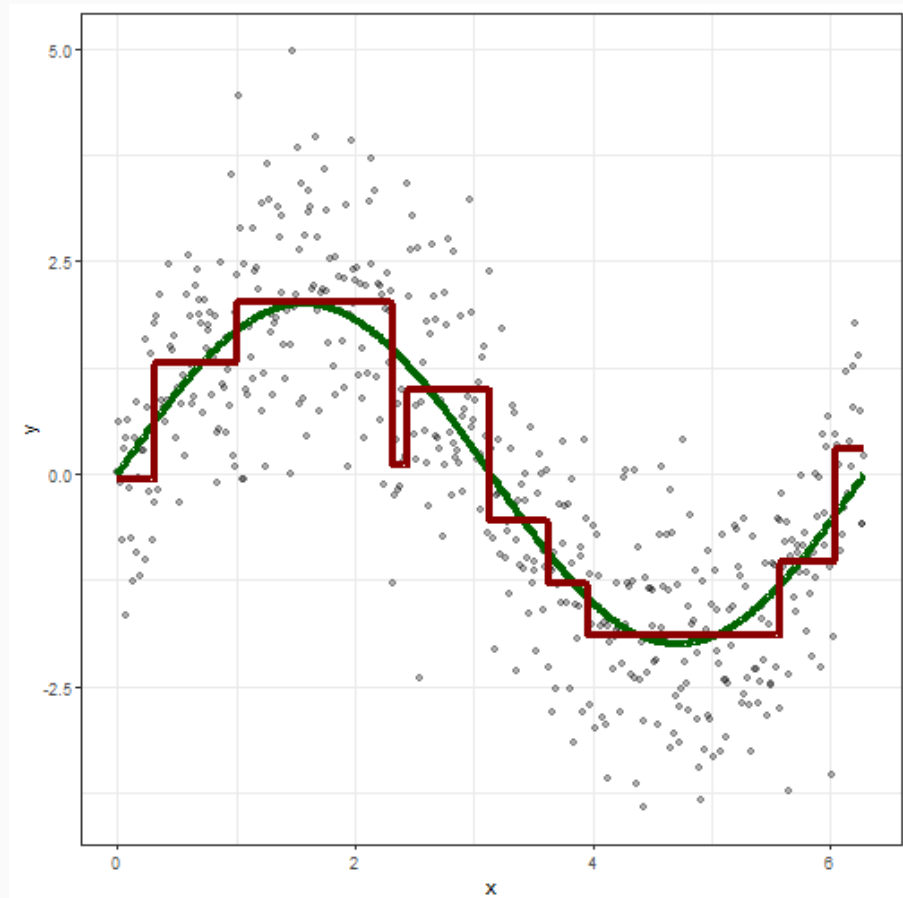
1. Generate a second data set `dfr2` with a different seed.
2. Fit an optimal tree to this data following the pruning strategy.
3. Can you spot substantial differences with the trees from before?

**Q.1:** a brand new data set

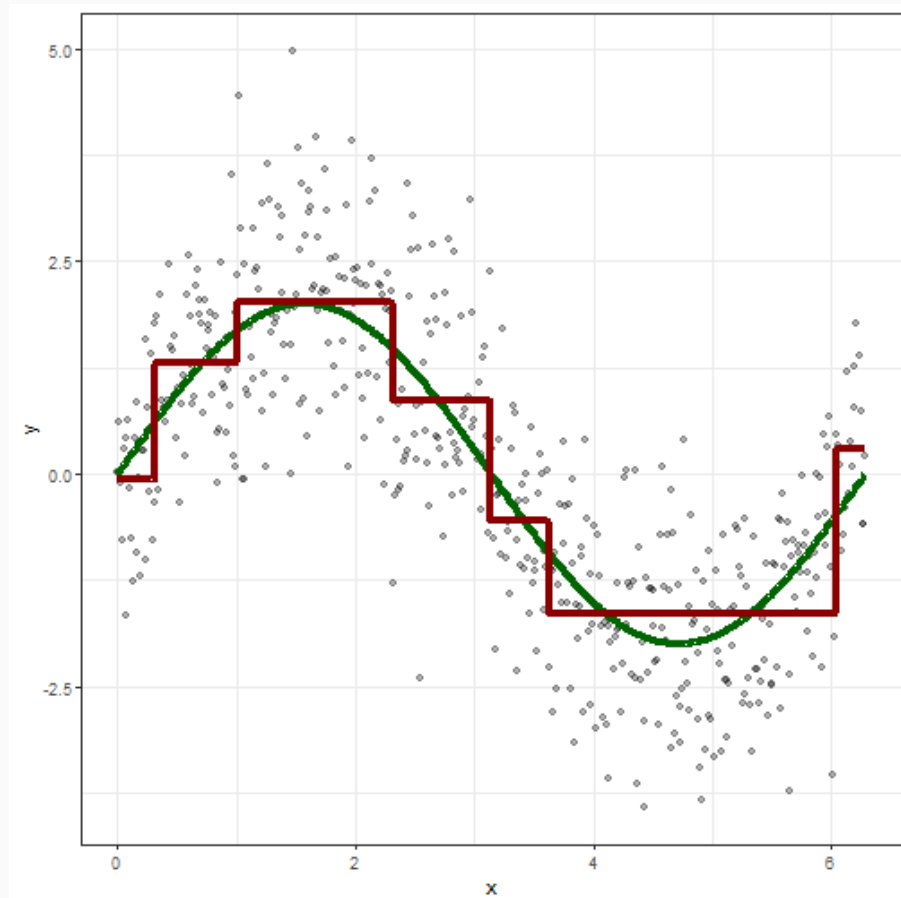
```
# Generate the data
set.seed(83625493)
dfr2 <- tibble(
  x = seq(0, 2 * pi, length.out = 500),
  m = 2 * sin(x),
  y = m + rnorm(length(x), sd = 1)
)
```



**Q.2a:** optimal tree with **min CV error**



**Q.2b:** optimal tree with **one SE rule**



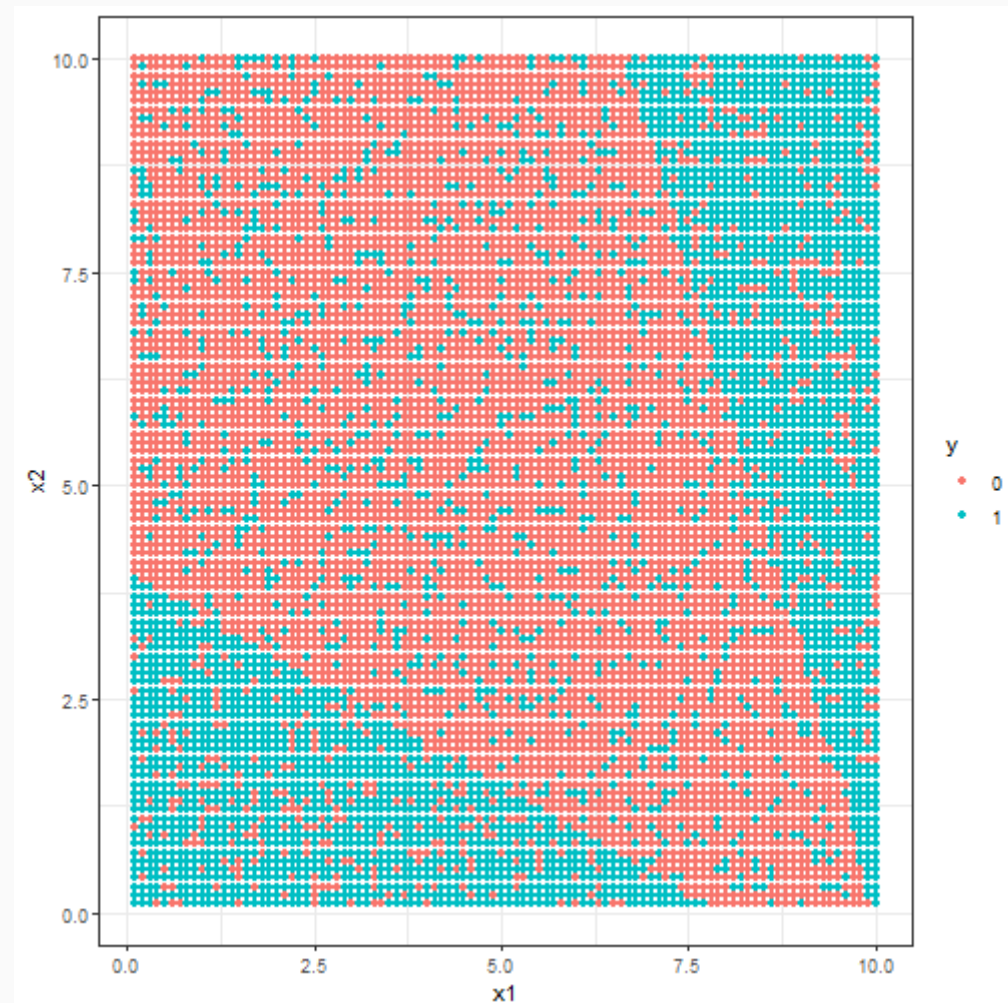
**Q.3:** trees look **rather different** compared to those from before, even though they try to approximate the same function.

# Toy example of a classification tree

---

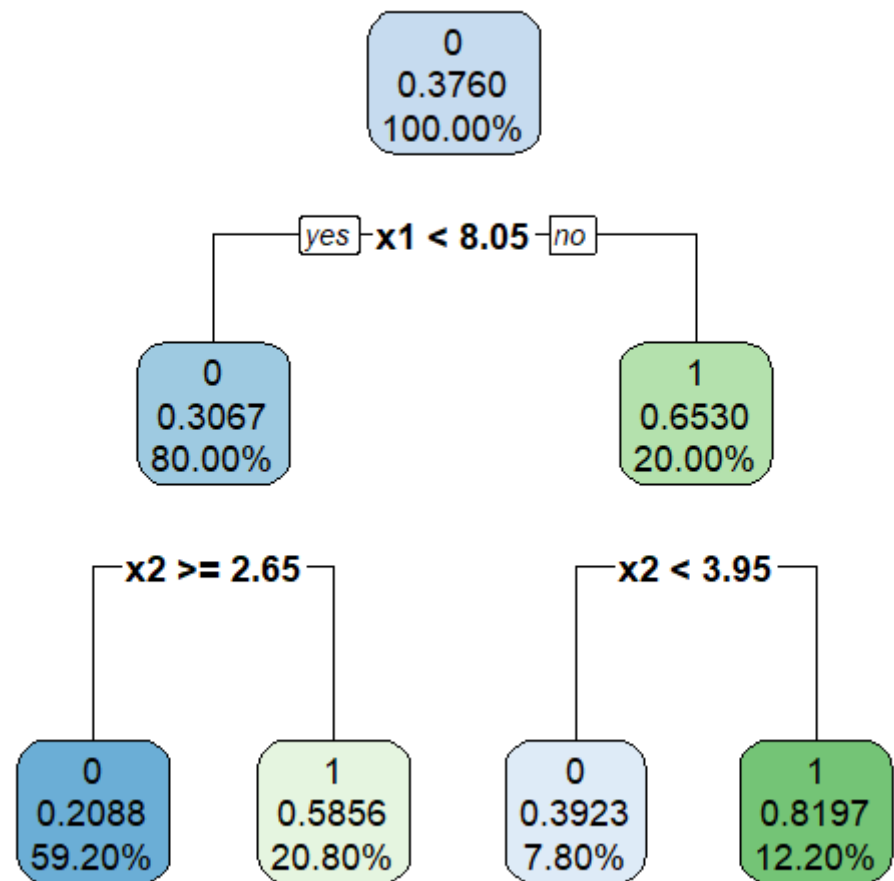
# Simulated data

```
set.seed(54321) # reproducibility
dfc <- tibble::tibble(
  x1 = rep(seq(0.1,10,by = 0.1), times = 100),
  x2 = rep(seq(0.1,10,by = 0.1), each = 100),
  y = as.factor(
    pmin(1,
      pmax(0,
        round(
          1*(x1+2*x2<8) + 1*(3*x1+x2>30) +
          rnorm(10000,sd = 0.5))
        )
      )
    )
  )
```



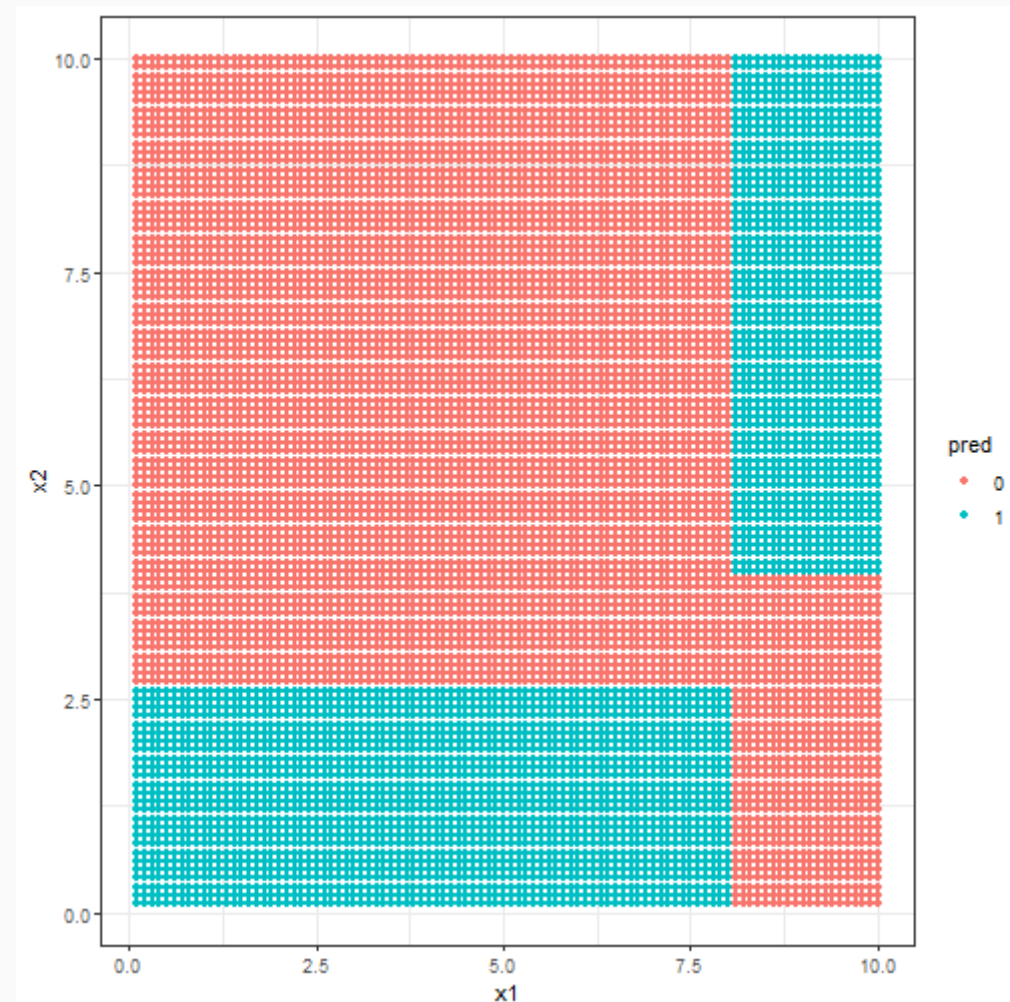
# Fitting a simple tree

```
fit <- rpart(formula = y ~ x1 + x2,  
             data = dfc,  
             method = 'class',  
             control = rpart.control(  
               maxdepth = 2  
             )  
           )  
print(fit)  
## n= 10000  
##  
## node), split, n, loss, yval, (yprob)  
##      * denotes terminal node  
##  
## 1) root 10000 3760 0 (0.6240000 0.3760000)  
##    2) x1< 8.05 8000 2454 0 (0.6932500 0.3067500)  
##      4) x2 ≥ 2.65 5920 1236 0 (0.7912162 0.2087838) *  
##      5) x2< 2.65 2080 862 1 (0.4144231 0.5855769) *  
##    3) x1 ≥ 8.05 2000 694 1 (0.3470000 0.6530000)  
##      6) x2< 3.95 780 306 0 (0.6076923 0.3923077) *  
##      7) x2 ≥ 3.95 1220 220 1 (0.1803279 0.8196721) *
```



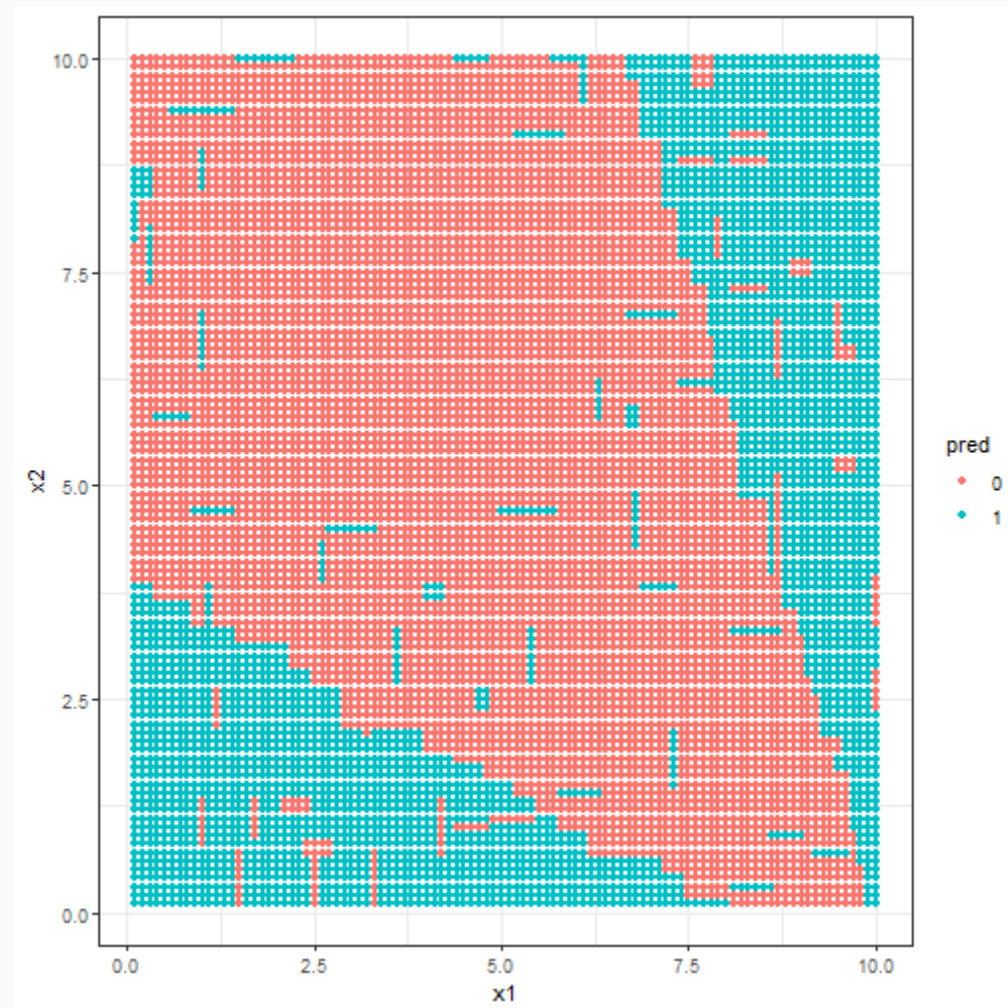
# Fitting a simple tree (cont.)

```
fit <- rpart(formula = y ~ x1 + x2,
             data = dfc,
             method = 'class',
             control = rpart.control(
               maxdepth = 2
             )
             )
print(fit)
## n= 10000
##
## node), split, n, loss, yval, (yprob)
##      * denotes terminal node
##
## 1) root 10000 3760 0 (0.6240000 0.3760000)
##   2) x1 < 8.05 8000 2454 0 (0.6932500 0.3067500)
##     4) x2 ≥ 2.65 5920 1236 0 (0.7912162 0.2087838) *
##     5) x2 < 2.65 2080 862 1 (0.4144231 0.5855769) *
##   3) x1 ≥ 8.05 2000 694 1 (0.3470000 0.6530000)
##     6) x2 < 3.95 780 306 0 (0.6076923 0.3923077) *
##     7) x2 ≥ 3.95 1220 220 1 (0.1803279 0.8196721) *
```



# What about an overly complex tree?

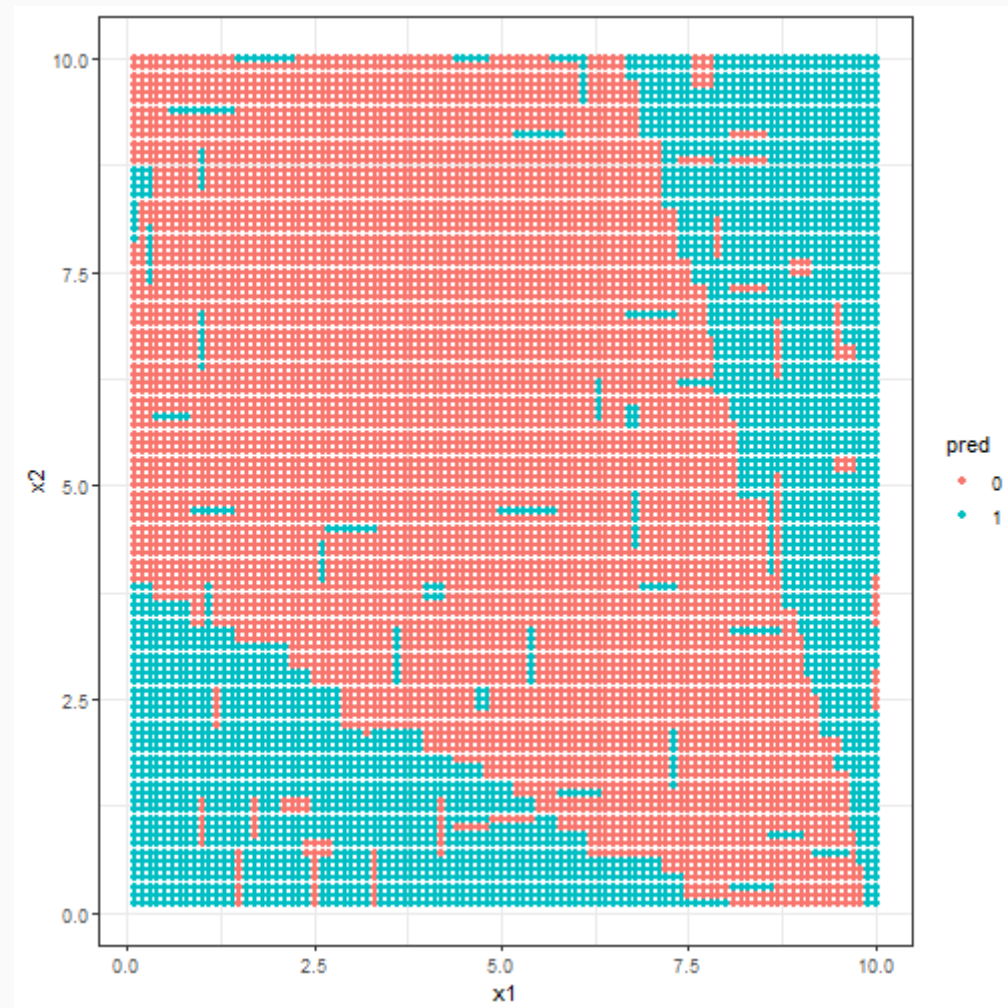
```
fit <- rpart(formula = y ~ x1 + x2,  
             data = dfc,  
             method = 'class',  
             control = rpart.control(  
               maxdepth = 20,  
               minsplit = 10,  
               minbucket = 5,  
               cp = 0  
             )  
)
```



# What about an overly complex tree?

```
fit <- rpart(formula = y ~ x1 + x2,  
             data = dfc,  
             method = 'class',  
             control = rpart.control(  
               maxdepth = 20,  
               minsplit = 10,  
               minbucket = 5,  
               cp = 0  
             )  
)
```

Clearly **overfitting**!







## Your turn

Let's find a satisfying fit for this classification example.

**Q:** perform **cross-validation** on `cp` to find the **optimal pruned subtree**.

1. Set `xval = 5` in `rpart.control()` (do not forget to set a **seed** beforehand).
2. Graphically inspect the xval results via `plotcp()`.
3. Extract the xval results via `$cptable`.
4. Apply the min xerror and/or the one SE rule to find the **optimal** `cp`.
5. Show the resulting classifier graphically.

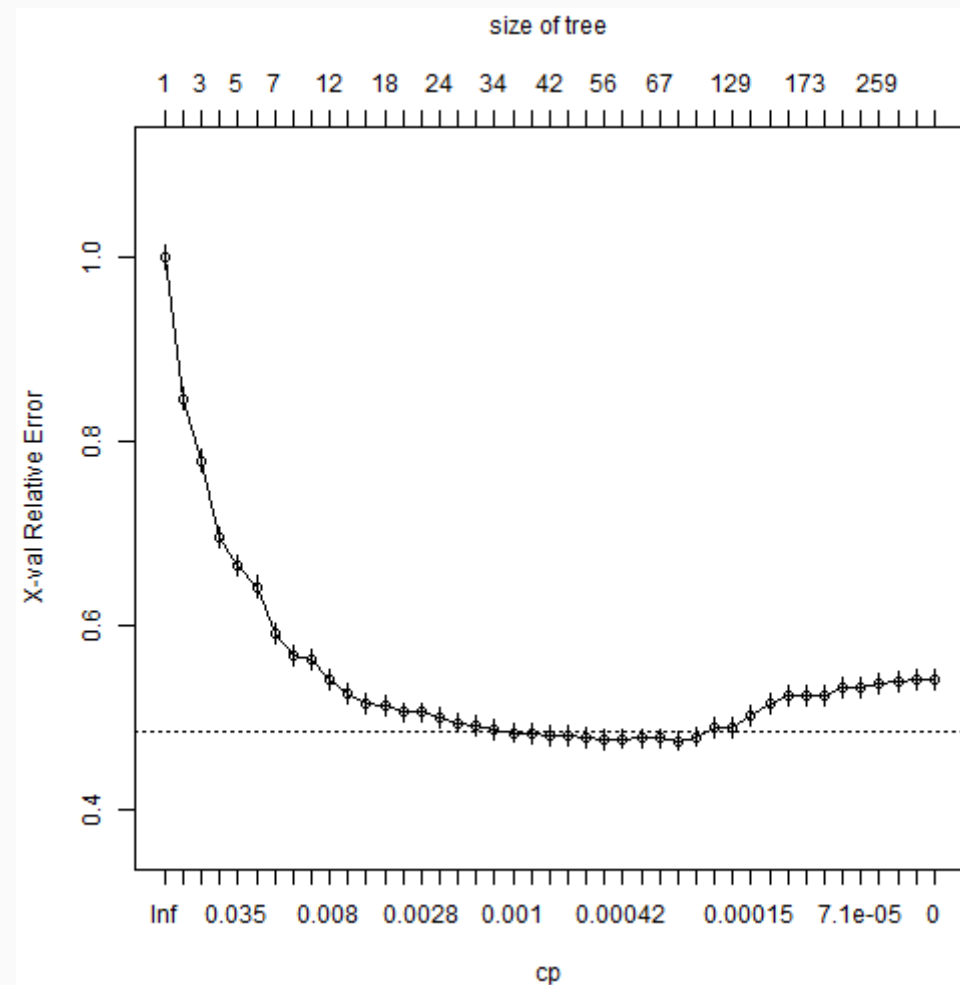


## Q.1: fit a complex tree and perform cross-validation

```
set.seed(87654) # reproducibility
fit <- rpart(formula = y ~ x1 + x2,
             data = dfc,
             method = 'class',
             control = rpart.control(
               maxdepth = 20,
               minsplit = 10,
               minbucket = 5,
               cp = 0,
               xval = 5
             )
)
```

## Q.2: inspect the xval results graphically

```
plotcp(fit)
```



**Q.3:** extract the xval results in a table

```
# Get xval results via 'cptable' attribute
cpt ← fit$cptable
```

**Q.4:** optimal `cp` via min CV error or one SE rule

```
# Which cp value do we choose?
min_xerr ← which.min(cpt[, 'xerror'])

se_rule ← min(which(cpt[, 'xerror'] <
  (cpt[min_xerr, 'xerror'] + cpt[min_xerr, 'xstd'])))
```

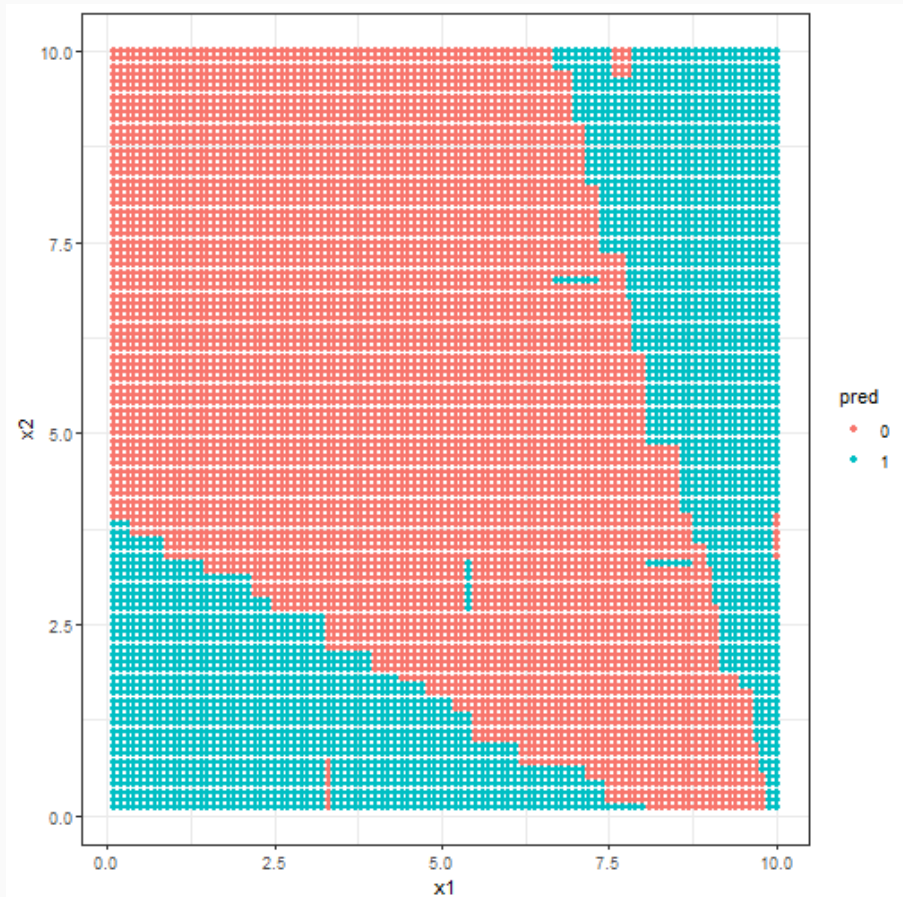
```
unnamed(min_xerr)
## [1] 29
```

```
se_rule
## [1] 20
```

```
print(cpt[16:35,], digits = 6)
##           CP nsplit rel error   xerror   xstd
## 16 0.001861702    23  0.471543 0.500000 0.0103913
## 17 0.001595745    25  0.467819 0.494149 0.0103443
## 18 0.001329787    26  0.466223 0.490957 0.0103184
## 19 0.001063830    33  0.456915 0.487234 0.0102880
## 20 0.000930851    34  0.455851 0.483245 0.0102552
## 21 0.000797872    36  0.453989 0.482713 0.0102508
## 22 0.000709220    41  0.450000 0.480319 0.0102310
## 23 0.000664894    44  0.447872 0.480319 0.0102310
## 24 0.000531915    50  0.443883 0.478457 0.0102155
## 25 0.000443262    55  0.441223 0.476330 0.0101978
## 26 0.000398936    58  0.439894 0.476596 0.0102000
## 27 0.000354610    60  0.439096 0.477660 0.0102089
## 28 0.000332447    66  0.436968 0.477660 0.0102089
## 29 0.000265957    74  0.434309 0.474734 0.0101844
## 30 0.000199468   103  0.426330 0.478989 0.0102200
## 31 0.000177305   112  0.424468 0.488830 0.0103011
## 32 0.000166223   128  0.421543 0.488830 0.0103011
## 33 0.000132979   139  0.419681 0.502128 0.0104082
## 34 0.000113982   153  0.417819 0.515160 0.0105105
## 35 0.000106383   167  0.416223 0.523936 0.0105780
```

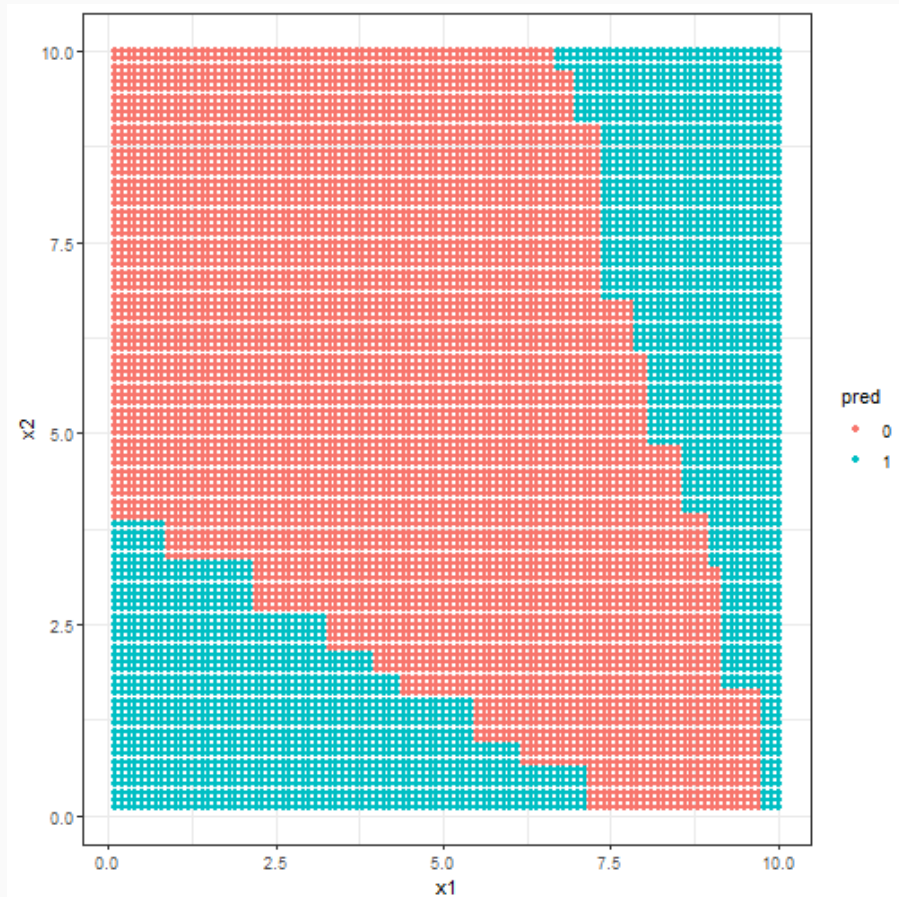
### Q.5a: optimal subtree via min CV error

```
fit_1 <- prune(fit, cp = cpt[min_xerr, 'CP'])
```



### Q.5b: optimal subtree via one SE rule

```
fit_2 <- prune(fit, cp = cpt[se_rule, 'CP'])
```



# Claim frequency and severity modeling with {rpart}

---

# Claim frequency prediction on the MTPL data

Recall the MTPL data set introduced in Module 1.

The **Poisson GLM** is a classic approach for modelling **claim frequency** data.

How to deal with claim counts in a decision tree?

Use the **Poisson deviance** as **loss function**:

$$D^{\text{Poi}} = 2 \cdot \sum_{i=1}^n y_i \cdot \ln \frac{y_i}{\text{expo}_i \cdot \hat{f}(x_i)} - \{y_i - \text{expo}_i \cdot \hat{f}(x_i)\},$$

with **expo** the exposure measure.

Here we go:

```
# Read the MTPL data
#setwd(dirname(rstudioapi::getActiveDocumentContext())$path))
mtpl <- read.table('../data/PC_data.txt',
                  header = TRUE, stringsAsFactors = TRUE) %>%
  as_tibble() %>% rename_all(tolower) %>% rename(expo = exp)
```

# Fitting a simple tree to the MTPL data

```
fit <- rpart(formula =  
  cbind(expo,nclaims) ~  
  ageph + agec + bm + power +  
  coverage + fuel + sex + fleet + use,  
  data = mtpl,  
  method = 'poisson',  
  control = rpart.control(  
    maxdepth = 3,  
    cp = 0)  
)
```

```
print(fit)
```

**Take-away**  - **Poisson tree** in {rpart}:

- Poisson deviance via `method = 'poisson'`
- response as two-column matrix: `cbind(expo,y)`.

```
## n= 163231  
##  
## node), split, n, deviance, yval  
##      * denotes terminal node  
##  
## 1) root 163231 89944.320 0.13933520  
##    2) bm< 6.5 127672 63455.290 0.11784050  
##      4) bm< 1.5 88621 41252.130 0.10550490  
##        8) ageph ≥ 55.5 33646 14281.360 0.08899811 *  
##        9) ageph< 55.5 54975 26835.800 0.11598320 *  
##    5) bm ≥ 1.5 39051 21872.010 0.14641040  
##      10) ageph ≥ 57.5 8463 4324.098 0.11963920 *  
##      11) ageph< 57.5 30588 17496.720 0.15408620 *  
##    3) bm ≥ 6.5 35559 24843.720 0.22188630  
##      6) bm< 10.5 22657 15022.440 0.19808030  
##        12) ageph ≥ 26.5 17196 10950.970 0.18565170 *  
##        13) ageph< 26.5 5461 4025.443 0.23668440 *  
##    7) bm ≥ 10.5 12902 9678.292 0.26753260  
##      14) agec< 6.5 4472 3181.981 0.23435030 *  
##      15) agec ≥ 6.5 8430 6471.783 0.28640140 *
```

# Fitting a simple tree to the MTPL data

```
fit <- rpart(formula =  
  cbind(expo,nclaims) ~  
  ageph + agec + bm + power +  
  coverage + fuel + sex + fleet + use,  
  data = mtpl,  
  method = 'poisson',  
  control = rpart.control(  
    maxdepth = 3,  
    cp = 0)  
)
```

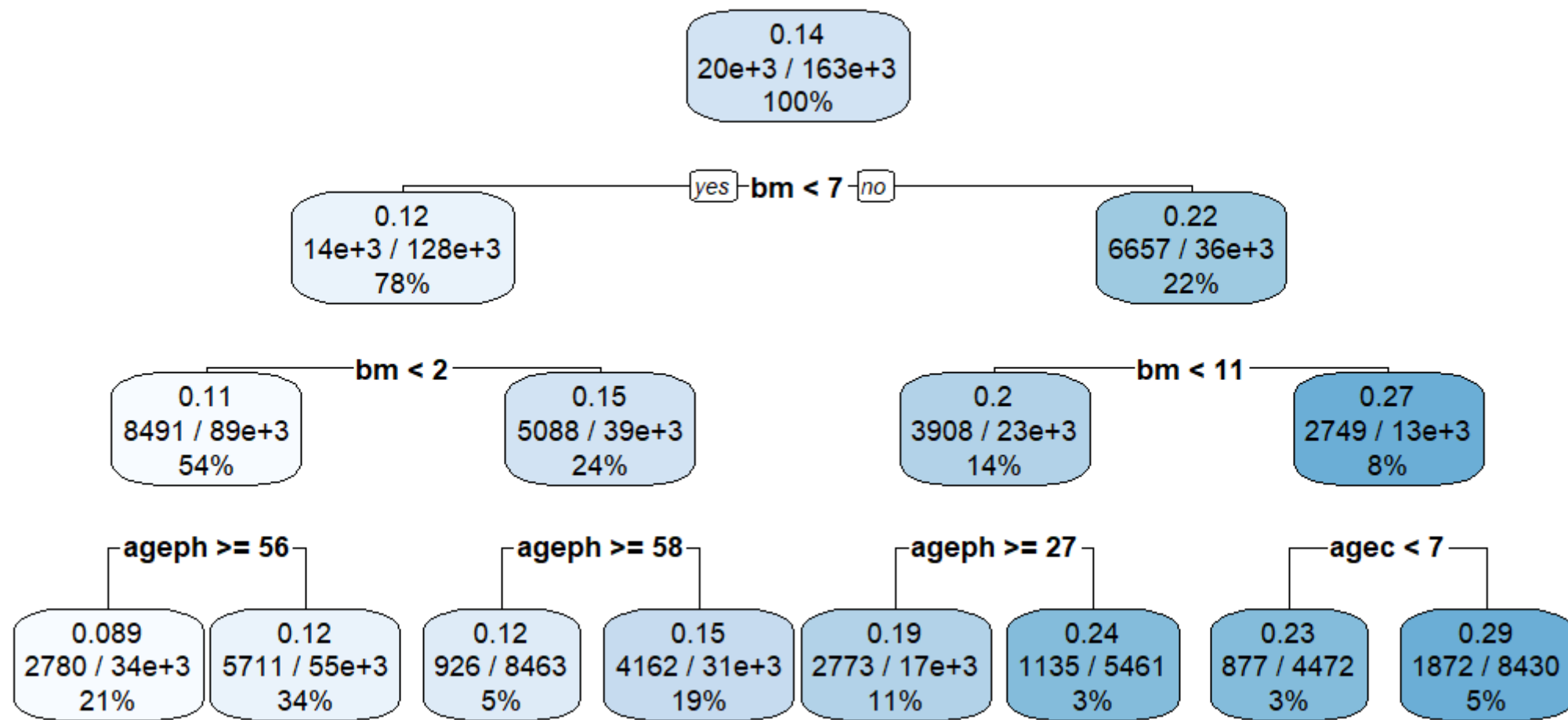
```
print(fit)
```

Easier way to **understand** this tree?

Try `rpart.plot` from the package `{rpart.plot}`

```
## n= 163231  
##  
## node), split, n, deviance, yval  
##      * denotes terminal node  
##  
## 1) root 163231 89944.320 0.13933520  
##    2) bm< 6.5 127672 63455.290 0.11784050  
##      4) bm< 1.5 88621 41252.130 0.10550490  
##        8) ageph ≥ 55.5 33646 14281.360 0.08899811 *  
##        9) ageph< 55.5 54975 26835.800 0.11598320 *  
##    5) bm ≥ 1.5 39051 21872.010 0.14641040  
##      10) ageph ≥ 57.5 8463 4324.098 0.11963920 *  
##      11) ageph< 57.5 30588 17496.720 0.15408620 *  
##    3) bm ≥ 6.5 35559 24843.720 0.22188630  
##      6) bm< 10.5 22657 15022.440 0.19808030  
##        12) ageph ≥ 26.5 17196 10950.970 0.18565170 *  
##        13) ageph< 26.5 5461 4025.443 0.23668440 *  
##    7) bm ≥ 10.5 12902 9678.292 0.26753260  
##      14) agec< 6.5 4472 3181.981 0.23435030 *  
##      15) agec ≥ 6.5 8430 6471.783 0.28640140 *
```

# Fitting a simple tree to the MTPL data







## Your turn

Verify whether the **prediction** in a leaf node is **what you would expect**.

**Q:** take the leftmost node as an example: `bm < 2` and `ageph ≥ 56`.

1. Subset the data accordingly.
2. Calculate the expected claim frequency as `sum(nclaims)/sum(expo)`.
3. Compare with the {rpart} prediction of 0.08899811.

**Q.1-Q.2:** subset the data and calculate the claim frequency

```
mtpl %>%  
  dplyr::filter(bm < 2,  
                ageph ≥ 56) %>%  
  dplyr::summarise(claim_freq =  
                    sum(nclaims)/sum(expo))
```

```
##   claim_freq  
## 1 0.08898655
```

**Q.3:** the prediction and our DIY calculation **do not match!**

Is this due to a rounding error?

Or is there something spooky 👻 going on?

# Unraveling the mystery of {rpart}

**Conceptually:** no events in a leaf node lead to division by zero in the deviance!

Solution: assume **Gamma prior** on the mean of the Poisson in the leaf nodes:

- set  $\mu = \sum y_i / \sum \text{expo}_i$
- use coefficient of variation  $k = \sigma / \mu$  as **user input**
- $k = 0$  extreme **pessimism** (all leaf nodes equal)
- $k = \infty$  extreme **optimism** (let the data speak)
- default in {rpart}:  $k = 1$ .

The resulting leaf node prediction:

$$\frac{\alpha + \sum Y_i}{\beta + \sum \text{expo}_i}, \quad \alpha = 1/k^2, \quad \beta = \alpha/\mu.$$

```
k ← 1

alpha ← 1/k^2

mu ← mtpl %>%
  with(sum(nclaims)/sum(expo))

beta ← alpha/mu

mtpl %>%
  dplyr::filter(bm < 2, ageph ≥ 56) %>%
  dplyr::summarise(prediction =
    (alpha + sum(nclaims))/(beta + sum(expo)))

##      prediction
## 1 0.08899811
```

More details in Section 8.2 of the **vignette** on Poisson regression.

# Coefficient of variation very low

```
fit <- rpart(formula =  
             cbind(expo,nclaims) ~  
             ageph + agec + bm + power +  
             coverage + fuel + sex + fleet + use,  
             data = mtpl,  
             method = 'poisson',  
             control = rpart.control(  
               maxdepth = 3,  
               cp = 0),  
             parms = list(shrink = 10^-5)  
             )
```

```
## n= 163231  
##  
## node), split, n, deviance, yval  
##      * denotes terminal node  
##  
## 1) root 163231 89944.320 0.1393352  
##    2) bm< 6.5 127672 63858.770 0.1393352  
##      4) bm< 1.5 88621 41974.470 0.1393352 *  
##      5) bm ≥ 1.5 39051 21884.300 0.1393352  
##        10) ageph ≥ 57.5 8463 4346.787 0.1393352 *  
##        11) ageph< 57.5 30588 17537.510 0.1393352 *  
##    3) bm ≥ 6.5 35559 26085.560 0.1393353 *
```

Notice that **all** leaf nodes predict the **same value**.

# Coefficient of variation very high

```
fit <- rpart(formula =
  cbind(expo,nclaims) ~
  ageph + agec + bm + power +
  coverage + fuel + sex + fleet + use,
  data = mtpl,
  method = 'poisson',
  control = rpart.control(
    maxdepth = 3,
    cp = 0),
  parms = list(shrink = 10^5)
)
```

```
# Remember this number?
mtpl %>%
  dplyr::filter(bm < 2, ageph ≥ 56) %>%
  dplyr::summarise(claim_freq =
    sum(nclaims)/sum(expo))
```

```
##   claim_freq
## 1 0.08898655
```

```
## n= 163231
##
## node), split, n, deviance, yval
##      * denotes terminal node
##
## 1) root 163231 89944.320 0.13933520
##    2) bm< 6.5 127672 63455.290 0.11783920
##      4) bm< 1.5 88621 41252.130 0.10550180
##        8) ageph ≥ 55.5 33646 14281.360 0.08898655 *
##        9) ageph< 55.5 54975 26835.800 0.11597980 *
##    5) bm ≥ 1.5 39051 21872.010 0.14641180
##      10) ageph ≥ 57.5 8463 4324.098 0.11962090 *
##      11) ageph< 57.5 30588 17496.720 0.15409010 *
##    3) bm ≥ 6.5 35559 24843.720 0.22190600
##      6) bm< 10.5 22657 15022.440 0.19810170
##        12) ageph ≥ 26.5 17196 10950.970 0.18567400 *
##        13) ageph< 26.5 5461 4025.443 0.23683020 *
##    7) bm ≥ 10.5 12902 9678.292 0.26762210
##      14) agec< 6.5 4472 3181.980 0.23453270 *
##      15) agec ≥ 6.5 8430 6471.783 0.28656300 *
```



## Your turn

**Q:** Follow the **pruning strategy** to develop a proper frequency tree model for the MTPL data.

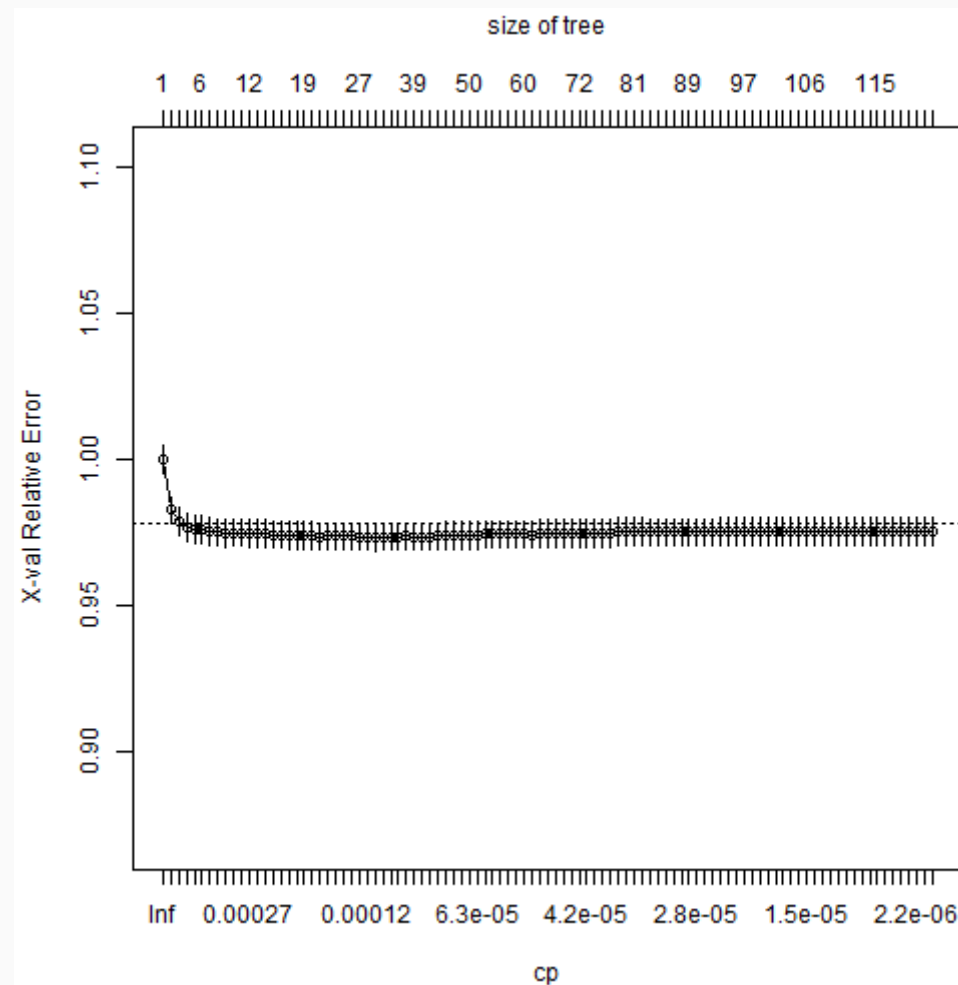
1. Start from an overly complex tree. Do not forget your favorite random **seed** upfront!
2. Inspect the cross-validation results.
3. Choose the `cp` value minimizing `xerror` for **pruning**.
4. Visualize the pruned tree with `rpart.plot`.

## Q.1: fit an overly complex tree

```
set.seed(9753) # reproducibility
fit <- rpart(formula =
  cbind(expo,nclaims) ~
  ageph + agec + bm + power +
  coverage + fuel + sex + fleet + use,
  data = mtpl,
  method = 'poisson',
  control = rpart.control(
    maxdepth = 20,
    minsplit = 2000,
    minbucket = 1000,
    cp = 0,
    xval = 5
  )
)
```

## Q.2: inspect the cross-validation results

```
plotcp(fit)
```



**Q.3:** choose the `cp` value that minimizes `xerror` for **pruning**

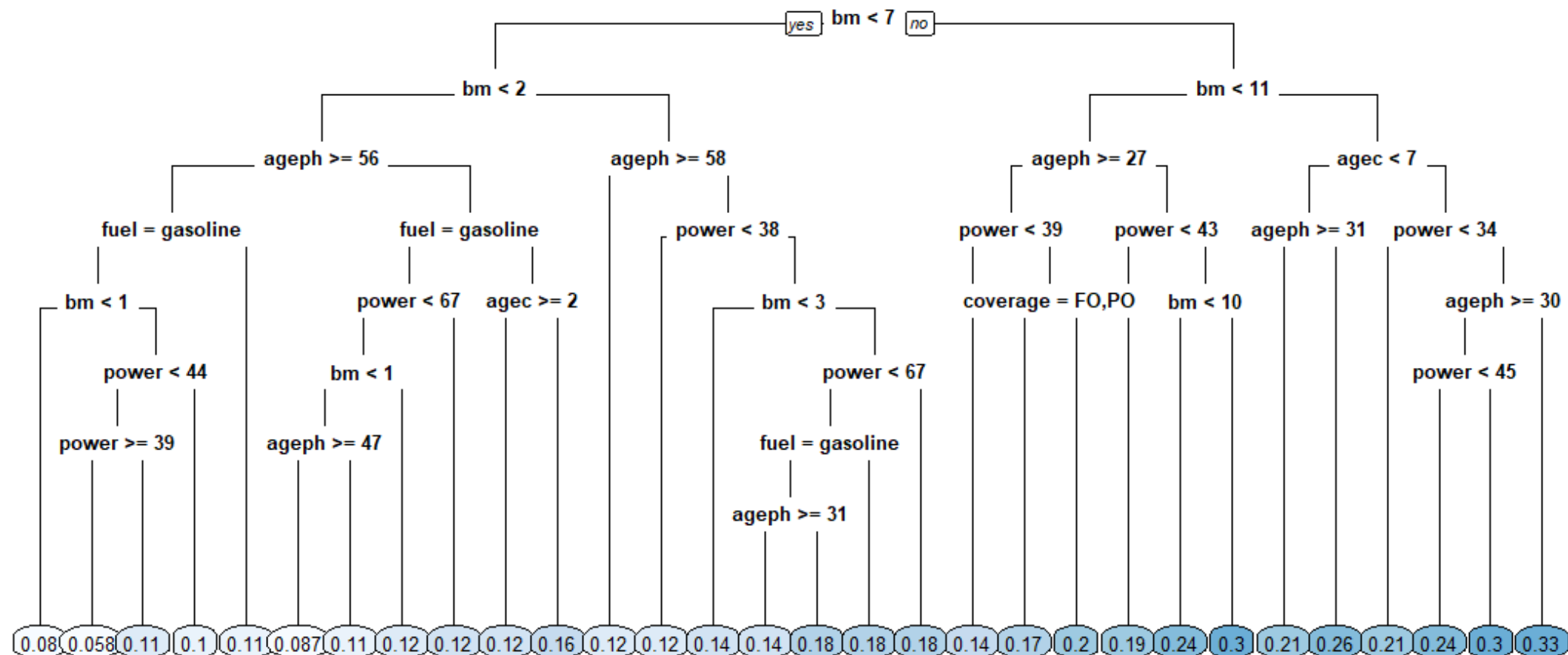
```
# Get the cross-validation results
cpt <- fit$cptable

# Look for the minimal xerror
min_xerr <- which.min(cpt[, 'xerror'])
cpt[min_xerr,]
##           CP      nsplit    rel error      xerror      xstd
## 1.152833e-04 2.900000e+01 9.693815e-01 9.735286e-01 4.668765e-03

# Prune the tree
fit_srt <- prune(fit,
                 cp = cpt[min_xerr, 'CP'])
```



**Q.4:** try to understand how the final model looks like. Can you make sense of it?



# Interpretation tools

---

# Interpreting a tree model

Interpretability depends on the **size of the tree**

- is easy with a **shallow** tree but hard with a **deep** tree
- luckily there are some **tools** to aid you.

## Feature importance

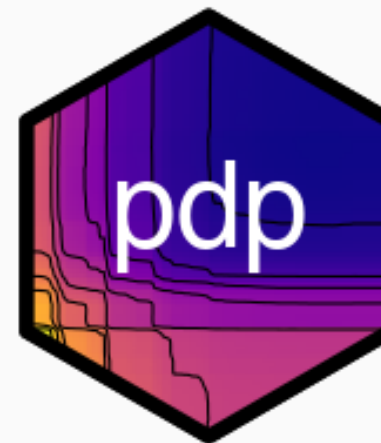
- identify the most **important** features
- implemented in the package {vip}.

## Partial dependence plot

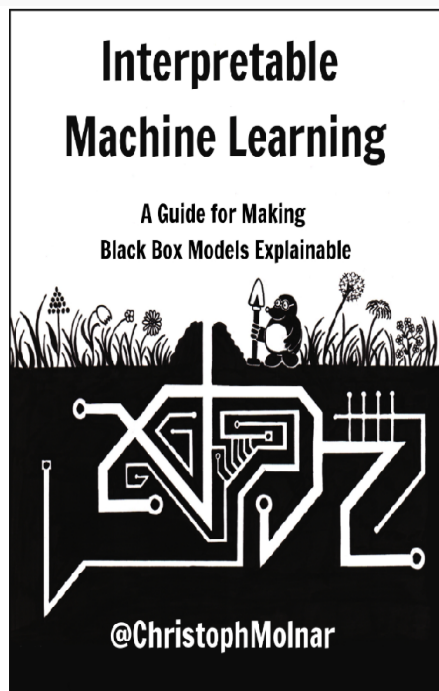
- measure the **marginal effect** of a feature
- implemented in the package {pdp}.

Excellent source on interpretable machine learning:

**Interpretable Machine Learning** book by Christophe Molnar.



# Feature importance and partial dependence



With **feature importance**:

- sum improvements in loss function over all splits on a variable  $x_\ell$
- important variables appear high and often in a tree.

With **partial dependence**:

- univariate

$$\bar{f}_\ell(x_\ell) = \frac{1}{n} \sum_{i=1}^n f_{\text{tree}}(x_\ell, \mathbf{x}_{-\ell}^i)$$

- bivariate

$$\bar{f}_{k,\ell}(x_k, x_\ell) = \frac{1}{n} \sum_{i=1}^n f_{\text{tree}}(x_k, x_\ell, \mathbf{x}_{-k,\ell}^i)$$

- marginal effects, interactions can stay hidden!

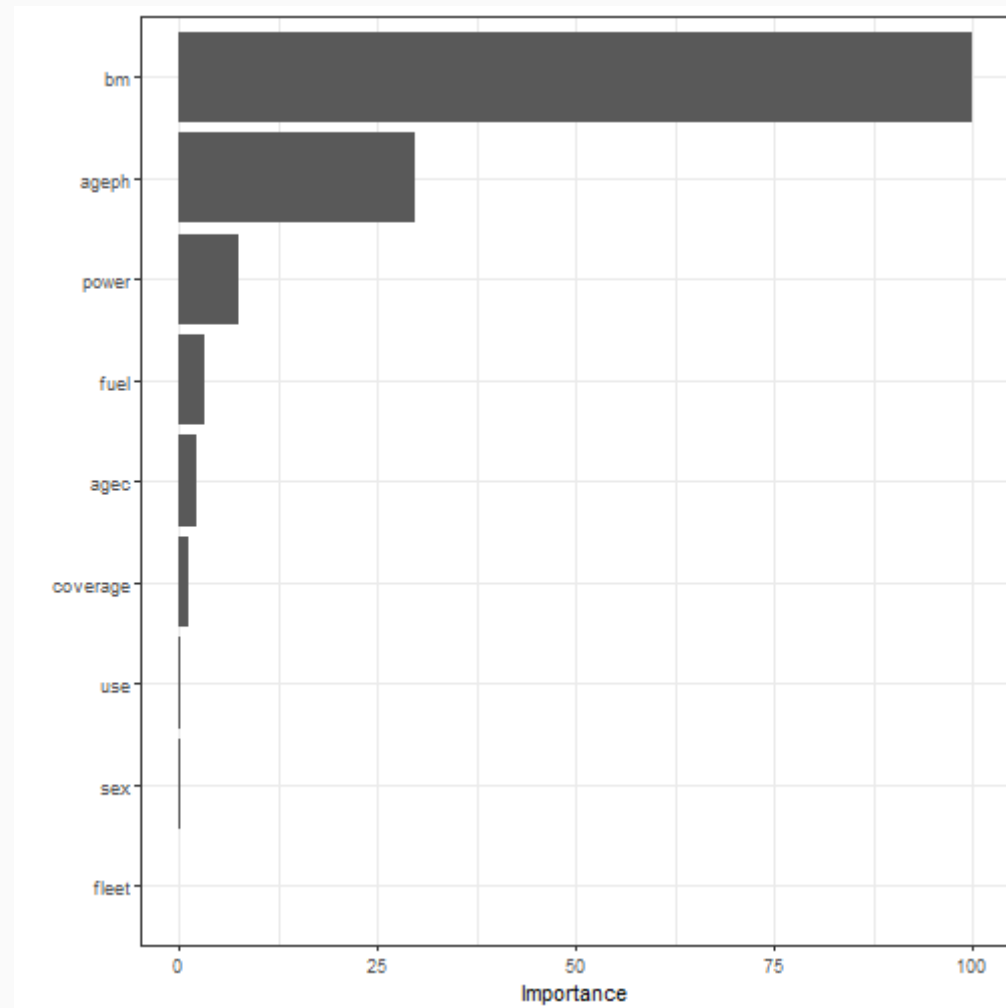
# Feature importance



```
library(vip)
# Function vi gives you the data
var_imp <- vip::vi(fit_srt)
```

```
## Variable Importance
## 1      bm 2186.5664729
## 2    ageph 651.1768792
## 3    power 164.2502735
## 4     fuel  70.5542003
## 5     agec  45.8086193
## 6 coverage 24.9202279
## 7      use   2.2088357
## 8      sex   0.7547625
## 9    fleet   0.2642045
```

```
# Function vip makes the plot
vip::vip(fit_srt, scale = TRUE)
```



# Partial dependence plot



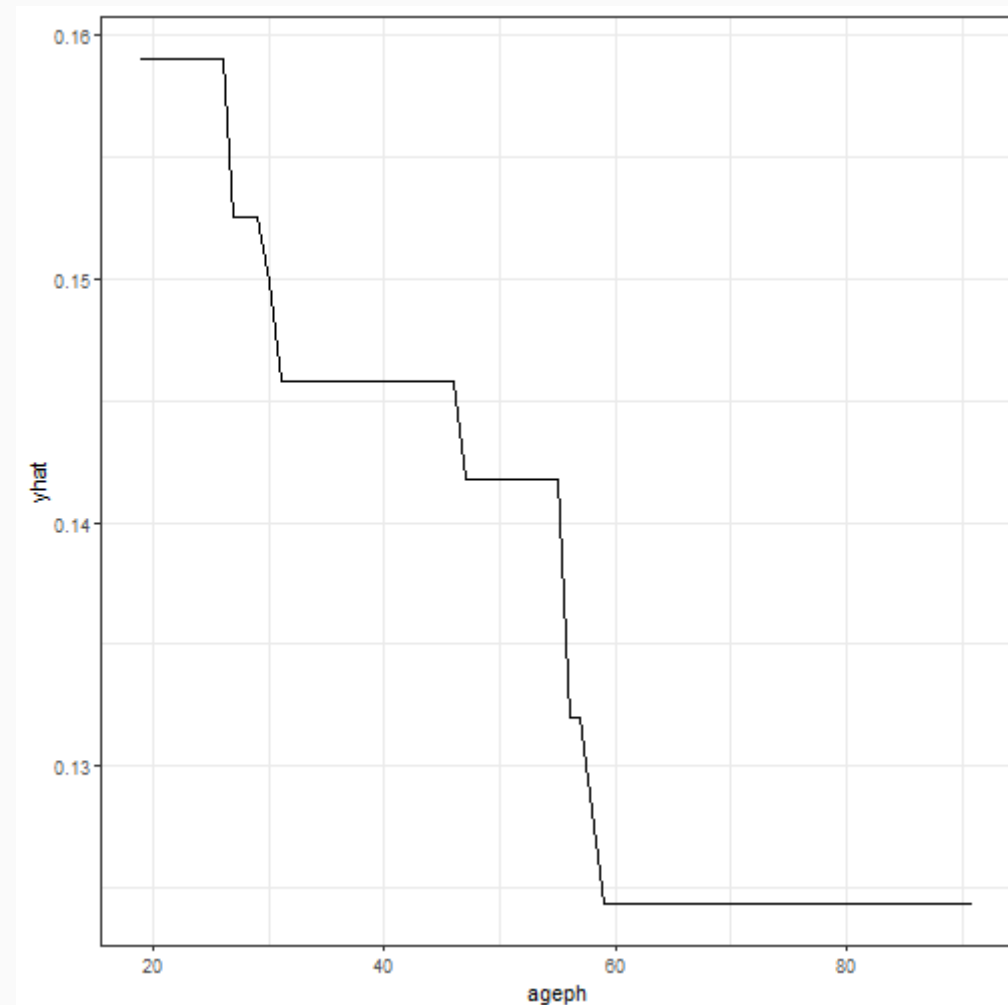
```
library(pdp)
# Need to define this helper function for Poisson
pred.fun <- function(object, newdata){
  mean(predict(object, newdata))
}

# Sample 5000 observations to speed up pdp generation
set.seed(48927)
pdp_ids <- mtpl %>% nrow %>% sample(size = 5000)
```

```
# partial: computes the marginal effect
# autoplot: creates the graph using ggplot2
fit_srt %>%
```

```
  pdp::partial(pred.var = 'ageph',
               pred.fun = pred.fun,
               train = mtpl[pdp_ids,]) %>%
```

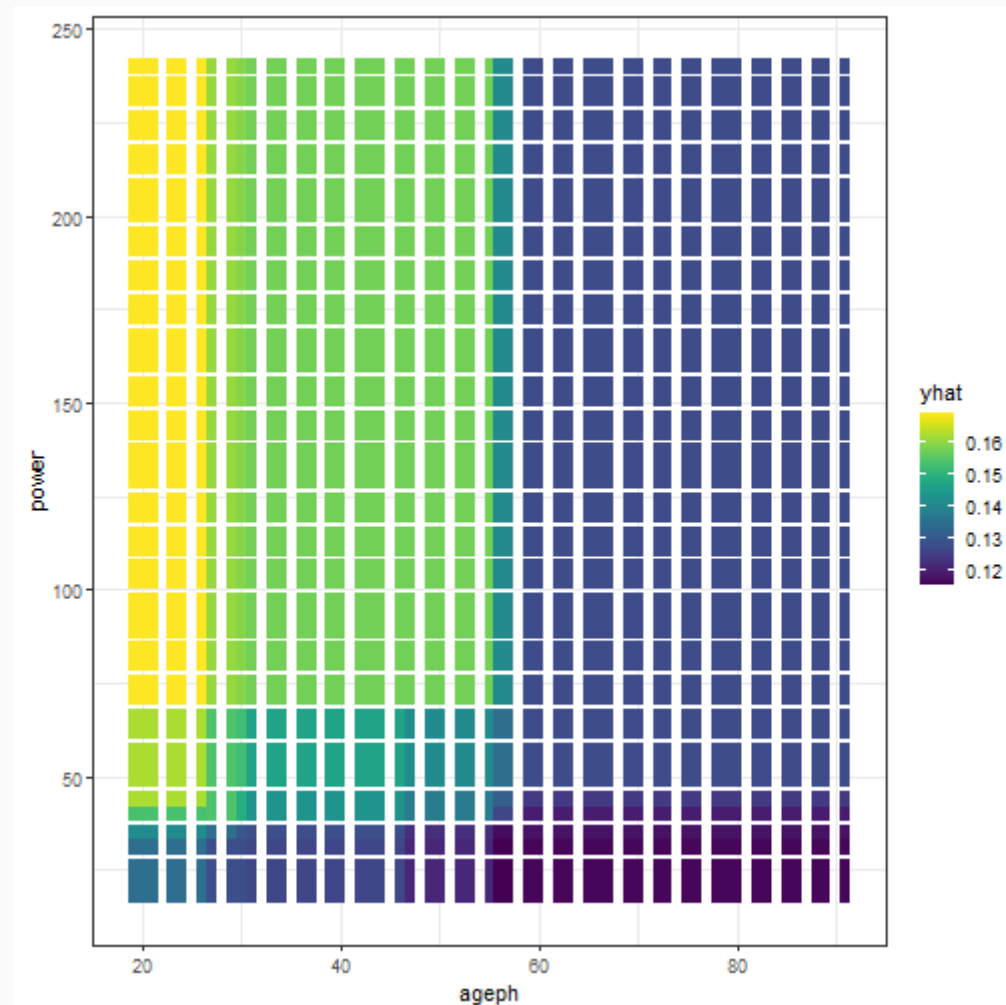
```
  autoplot()
```



# Partial dependence plot in two dimensions



```
# partial: computes the marginal effect  
# autoplot: creates the graph using ggplot2  
fit_srt %>%  
  pdp::partial(pred.var = c('ageph', 'power'),  
               pred.fun = pred.fun,  
               train = mtpl[pdp_ids,]) %>%  
  autoplot()
```



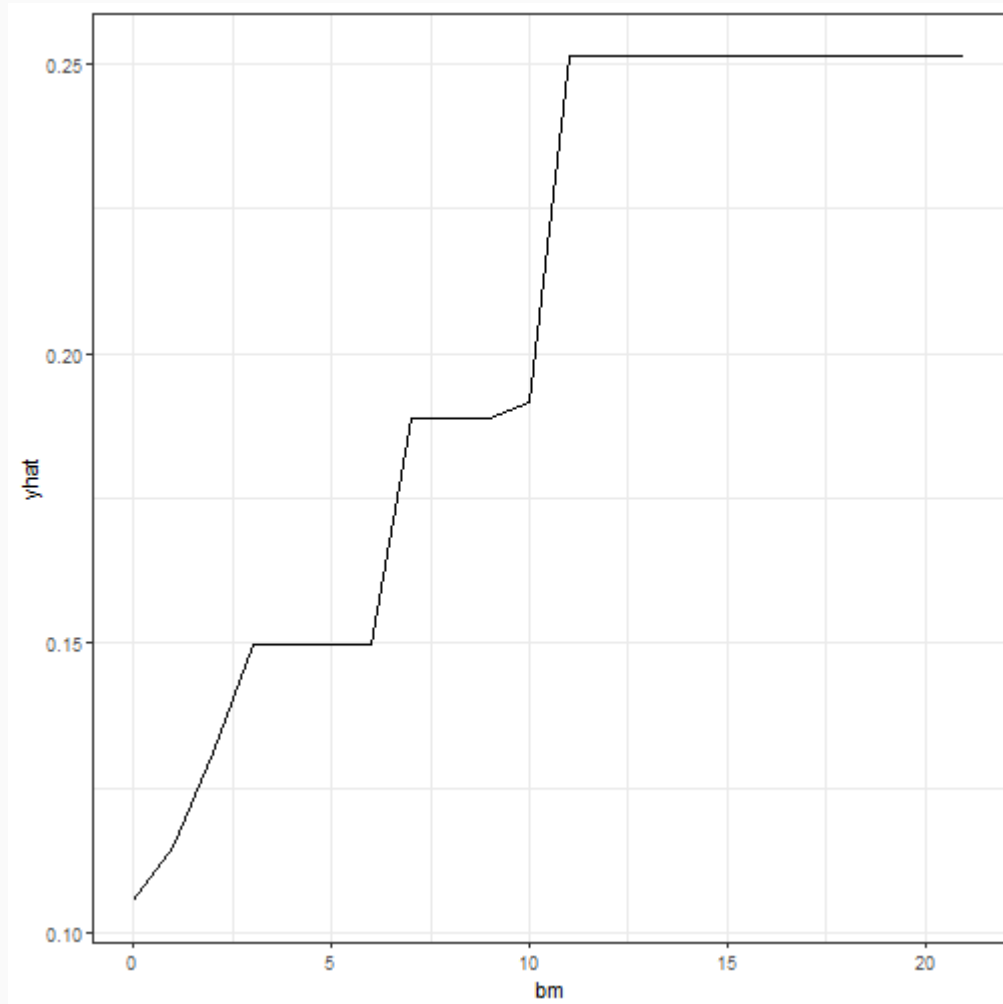


Use partial dependence plots for **other features** to gain **understanding** of your model.

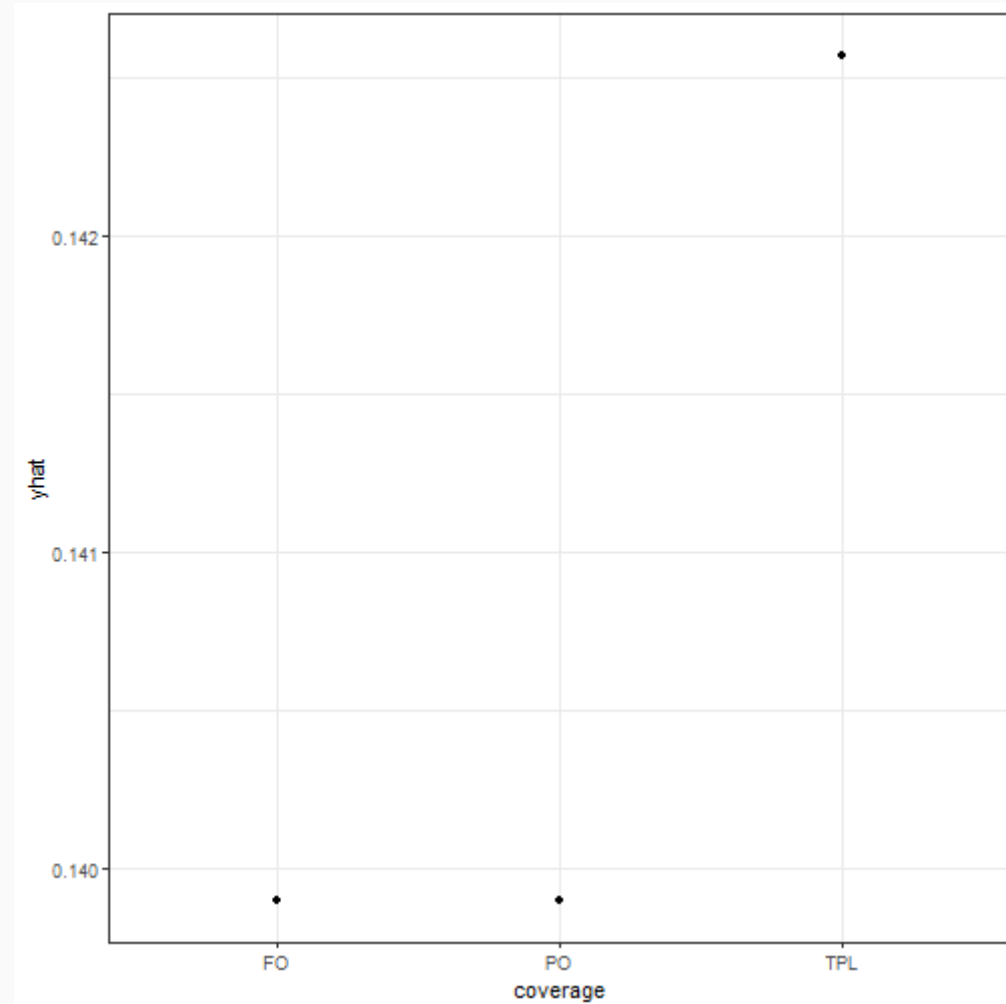
## Your turn



## Level in the bonus-malus scale



## Type of coverage



# That's a wrap on single trees!

## Advantages 😊

- Shallow tree is easy to **explain** graphically.
- Closely mirror the human **decision-making** process.
- Handle all types of features **without** pre-processing.
- **Fast** and very scalable to big data.
- **Automatic** variable selection.
- Surrogate splits can handle **missing** data.

# That's a wrap on single trees!

## Advantages 😊

- Shallow tree is easy to **explain** graphically.
- Closely mirror the human **decision-making** process.
- Handle all types of features **without** pre-processing.
- **Fast** and very scalable to big data.
- **Automatic** variable selection.
- Surrogate splits can handle **missing** data.

## Disadvantages 😞

- Tree uses **step** functions to approximate the effect.
- Greedy heuristic approach chooses **locally** optimal split (i.e., based on all previous splits).
- Data becomes **smaller** and smaller down the tree.
- All this results in **high variance** for a tree model...
- ... which harms **predictive performance**.

# From a single tree to ensembles of trees

---

# Ensembles of trees

Remember: prediction error = bias + variance.

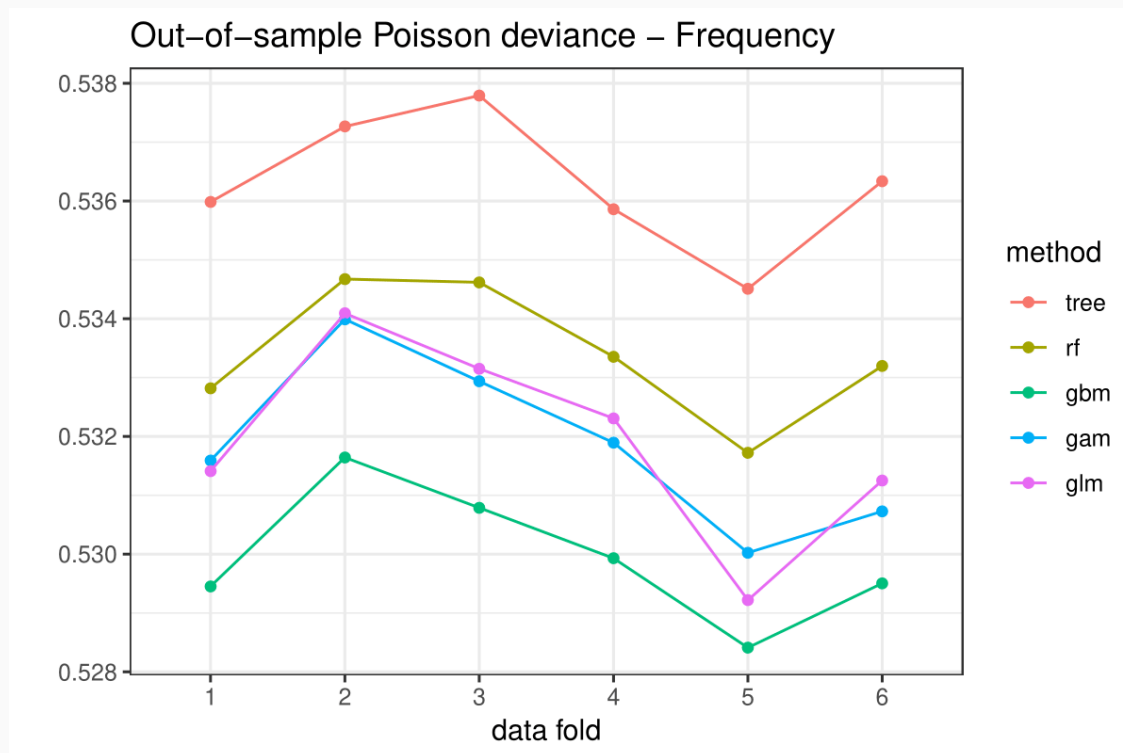
Good **predictive performance** requires low bias **AND** low variance.

Two popular **ensemble** algorithms (that can be applied to any type of model, not just trees) are:

- **bagging**:
  - low **bias** via detailed individual models (think: deep trees)
  - low **variance** via averaging of those models
  - **random forest** is a modification on bagging for trees to further improve the variance reduction.
- **boosting**:
  - low **variance** via simple individual models
  - low **bias** by incrementing the model sequentially.

# From our own experience

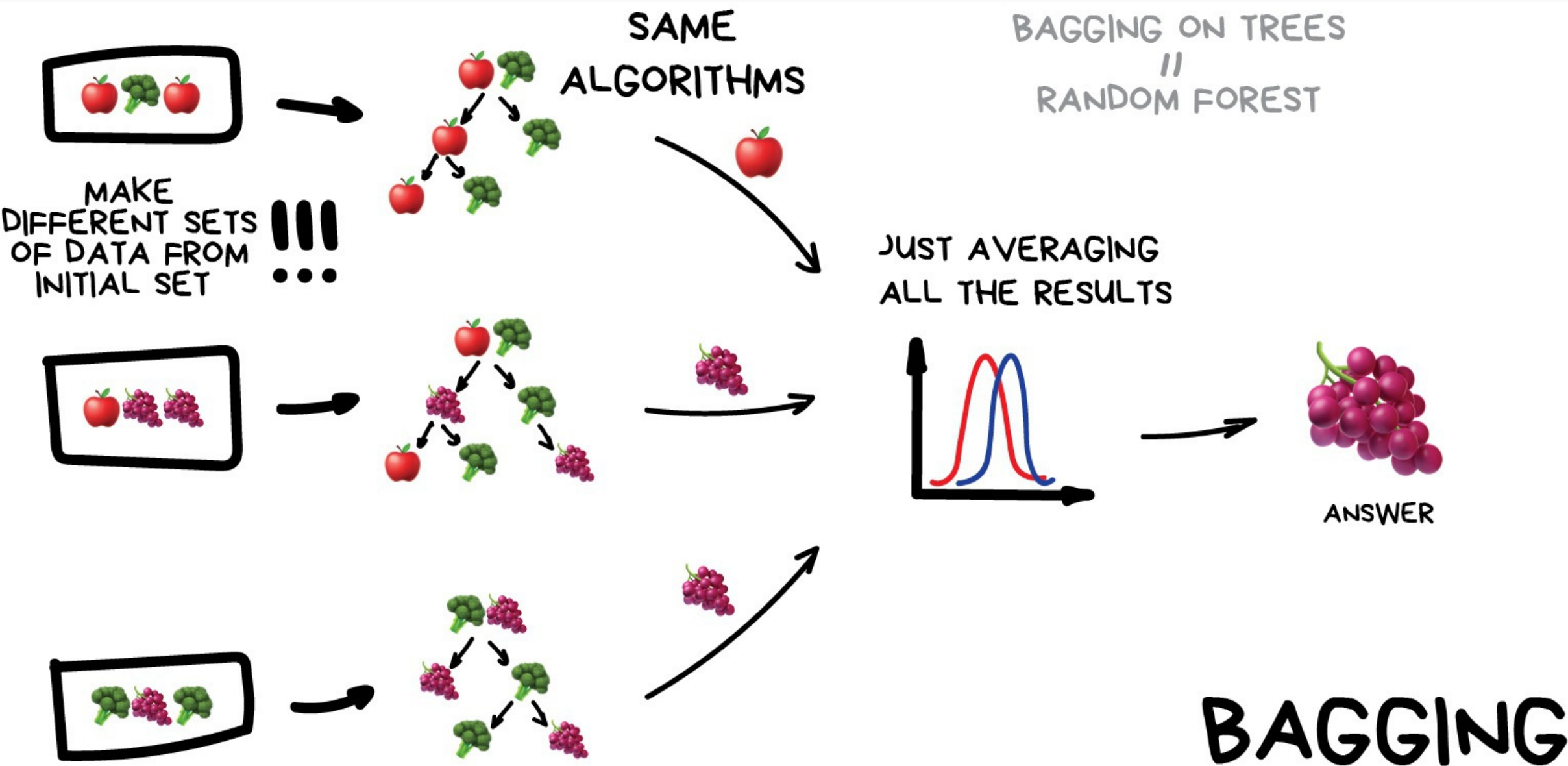
Boosting > Random forest > Bagging > Decision tree



Detailed discussion in our North American Actuarial Journal (2020) paper on [Boosting insights in insurance tariff plans with tree-based machine learning methods](#). See [Henckaerts et al. \(2020\)](#).

# Introducing bagging

---





# Bagging

Bagging is for Bootstrap AGGregatING.

Simple idea:

- build a lot of different **base learners** on bootstrapped samples of the data
- **combine** their predictions.

Model **averaging** helps to:

- reduce variance
- avoid overfitting.

Bagging works best for **base learners** with:

- **low bias** and **high variance**
- for example: deep decision trees.

# Bagging

Bagging is for Bootstrap AGGREGatING.

Simple idea:

- build a lot of different **base learners** on bootstrapped samples of the data
- **combine** their predictions.

Model **averaging** helps to:

- reduce variance
- avoid overfitting.

Bagging works best for **base learners** with:

- **low bias** and **high variance**
- for example: deep decision trees.

Bagging with trees?

- Do **B** times:
  - create **bootstrap sample** by drawing with replacement from the original data
  - fit a **deep tree** to the bootstrap sample.
- **Combine** the predictions of the B trees
  - **average** prediction for regression
  - **majority** vote for classification.

This is implemented in the {ipred} package, using {rpart} under the hood.

# Bootstrap samples

```
# Set a seed for reproducibility
set.seed(45678)

# Generate the first bootstrapped sample
bsample_1 <- dfr %>% nrow %>%
  sample(replace = TRUE)

# Generate another bootstrapped sample
bsample_2 <- dfr %>% nrow %>%
  sample(replace = TRUE)

# Use the indices to sample the data
dfr_b1 <- dfr %>%
  dplyr::slice(bsample_1)
dfr_b2 <- dfr %>%
  dplyr::slice(bsample_2)

# Let's have a look at the sampled data
dfr_b1 %>% dplyr::arrange(x) %>% head()
dfr_b2 %>% dplyr::arrange(x) %>% head()
```

Sample 1:

##		x	m	y
##	1	0.02518311	0.05036089	-0.7336728
##	2	0.02518311	0.05036089	-0.7336728
##	3	0.03777466	0.07553136	-1.5750691
##	4	0.06295777	0.12583237	-0.9696970
##	5	0.10073243	0.20112432	1.5964765
##	6	0.11332398	0.22616316	0.4061405

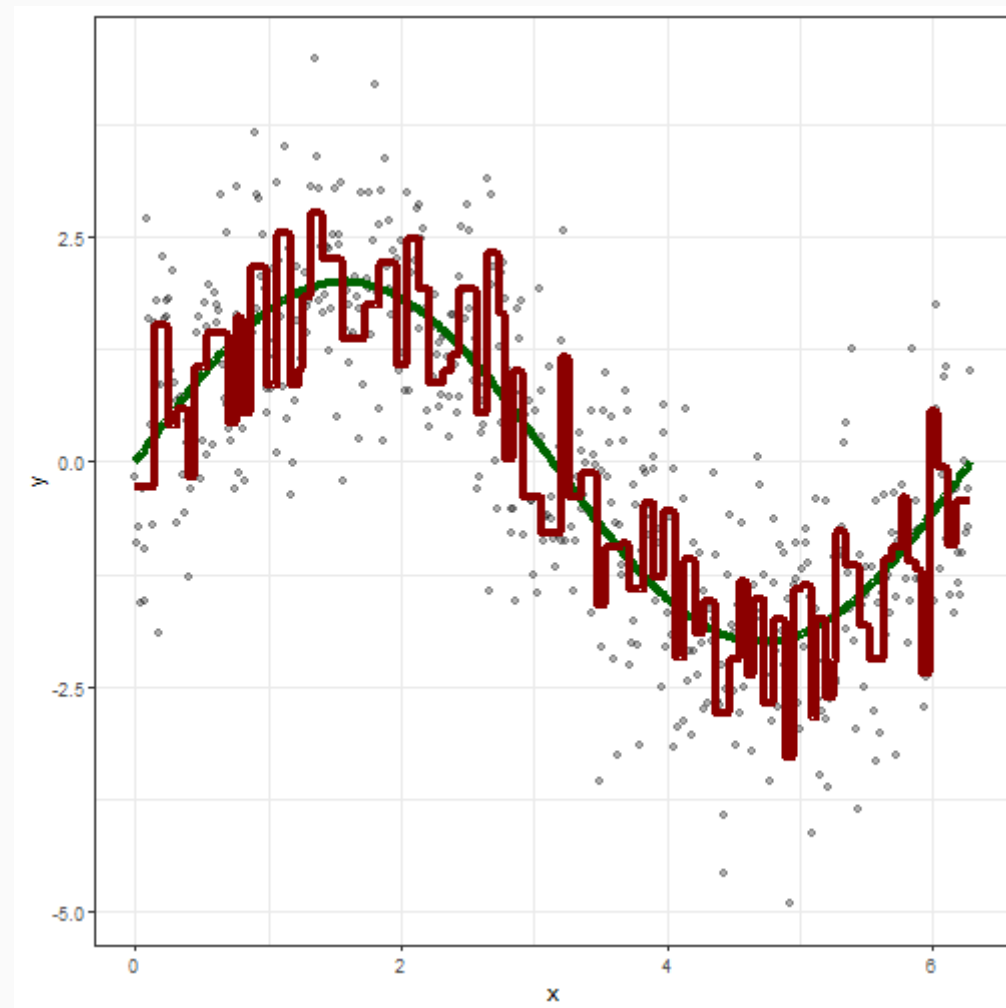
Sample 2:

##		x	m	y
##	1	0.00000000	0.00000000	-0.1789007
##	2	0.00000000	0.00000000	-0.1789007
##	3	0.00000000	0.00000000	-0.1789007
##	4	0.01259155	0.02518244	-0.9028617
##	5	0.06295777	0.12583237	-0.9696970
##	6	0.07554932	0.15095495	-1.5412872

# Decision tree on sample 1

```
fit_b1 <- rpart(formula = y ~ x,  
  data = dfr_b1,  
  method = 'anova',  
  control = rpart.control(  
    maxdepth = 20,  
    minsplit = 10,  
    minbucket = 5,  
    cp = 0  
  )  
)
```

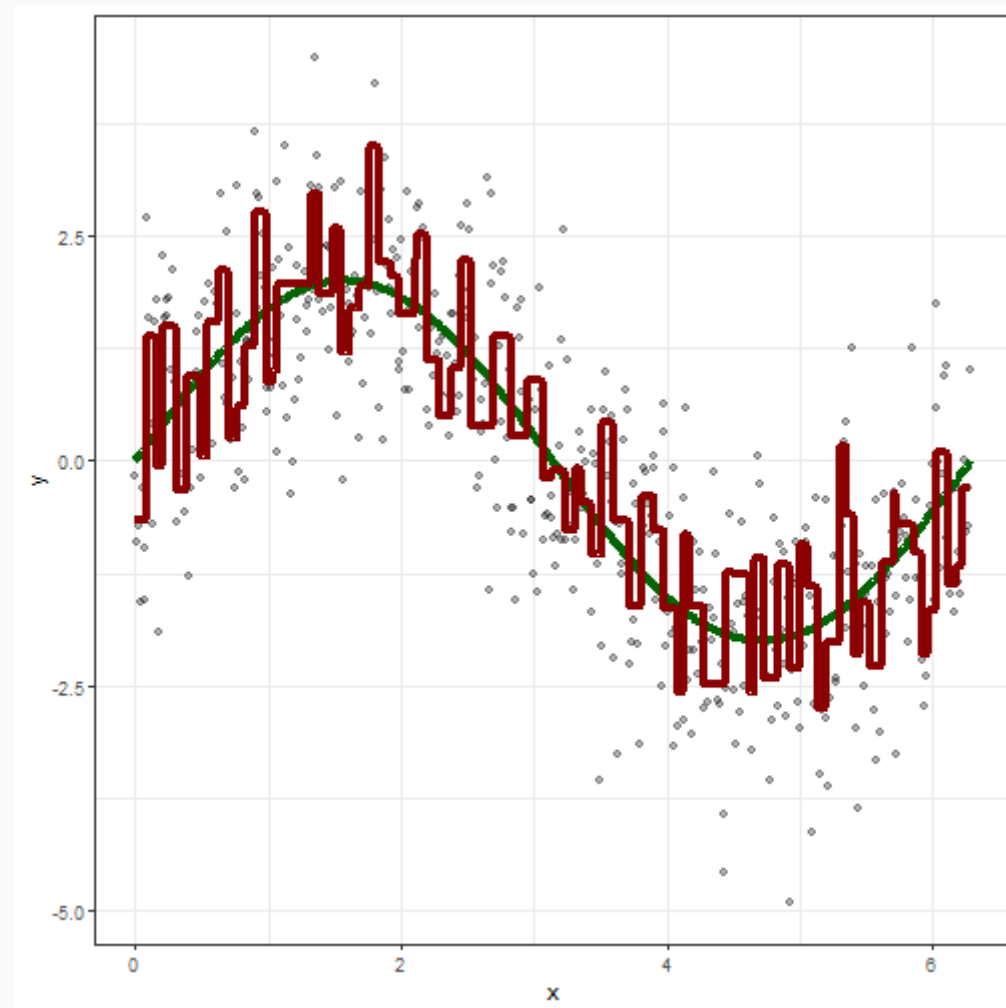
On it's own, this is a **noisy prediction** with very **high variance**!



# Decision tree on sample 2

```
fit_b2 <- rpart(formula = y ~ x,  
  data = dfr_b2,  
  method = 'anova',  
  control = rpart.control(  
    maxdepth = 20,  
    minsplit = 10,  
    minbucket = 5,  
    cp = 0  
  )  
)
```

Again, very **high variance** on it's own!



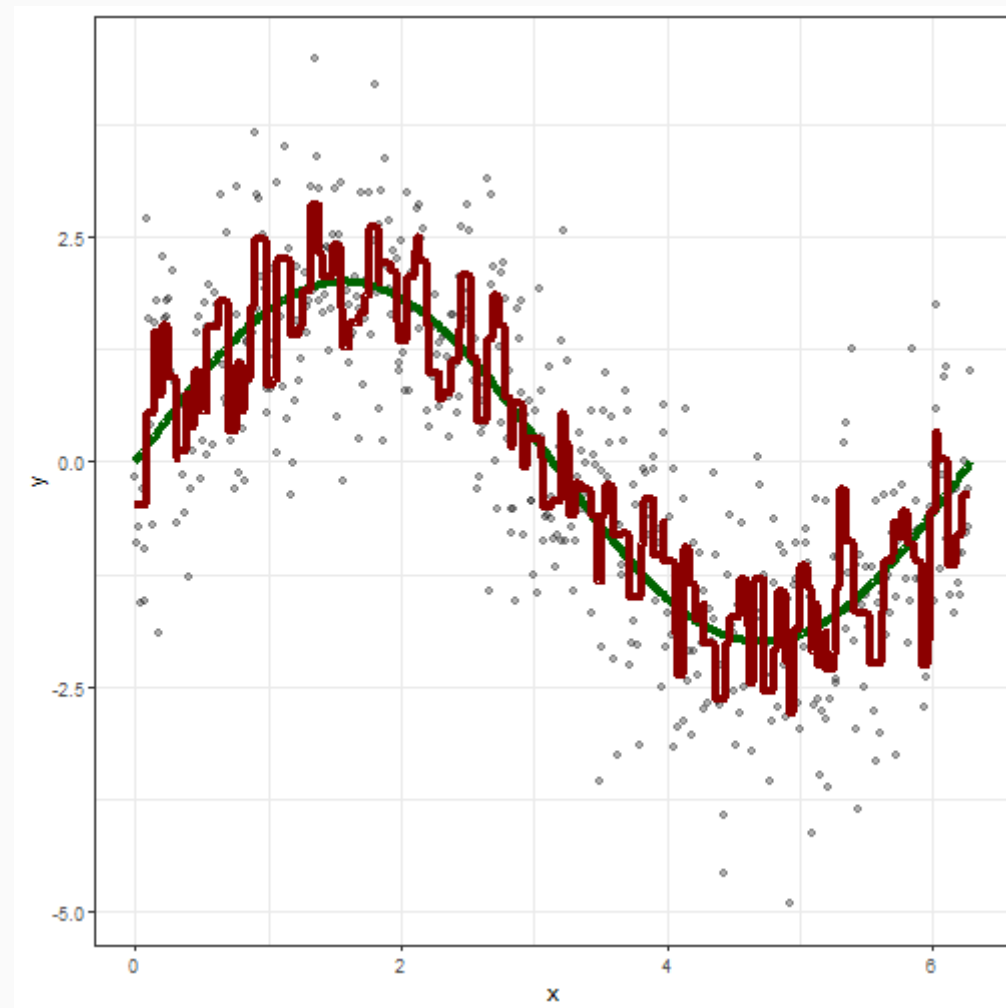
# Combining the predictions of both trees

```
# Predictions for the first tree
pred_b1 <- fit_b1 %>% predict(dfr)
# Predictions for the first tree
pred_b2 <- fit_b2 %>% predict(dfr)

# Average the predictions
pred <- rowMeans(cbind(pred_b1,
                        pred_b2))
```

Does it look like the prediction it's getting **less noisy**?

In other words: **is variance reducing?**





## Your turn

**Q:** add a **third tree** to the **bagged ensemble** and inspect the predictions.

1. Generate a **bootstrap sample** of the data (note: don't use the same seed as before because your bootstrap samples will be the same).
2. Fit a **deep tree** to this bootstrap sample.
3. Make predictions for this tree and **average** with the others.

### Q1: bootstrap sample with different seed

```
# Generate the third bootstrapped sample
set.seed(28726)
bsample_3 ← dfr %>% nrow %>%
  sample(replace = TRUE)
# Use the indices to sample the data
dfr_b3 ← dfr %>% dplyr::slice(bsample_3)
```

### Q2: fit a deep tree

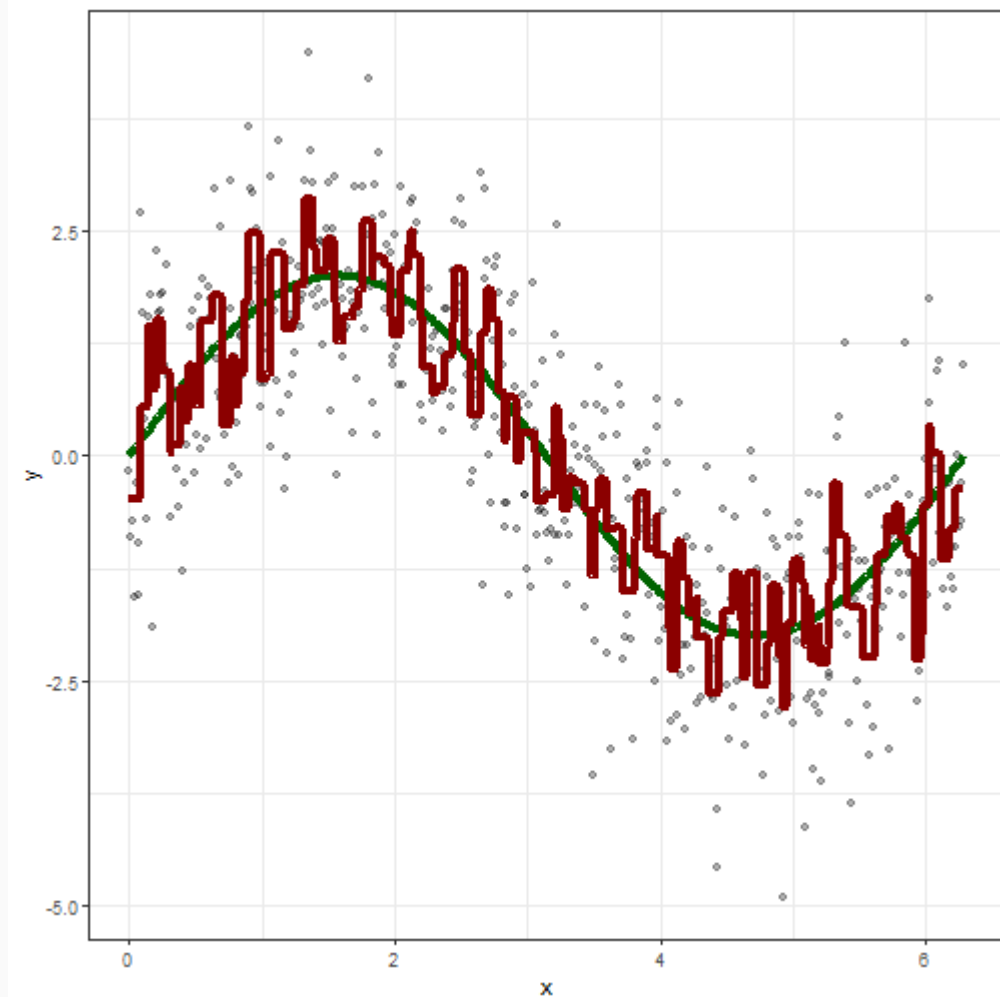
```
# Fit an unpruned tree
fit_b3 ← rpart(formula = y ~ x,
  data = dfr_b3,
  method = 'anova',
  control = rpart.control(
    maxdepth = 20,
    minsplit = 10,
    minbucket = 5,
    cp = 0))
```

### Q3: average the predictions

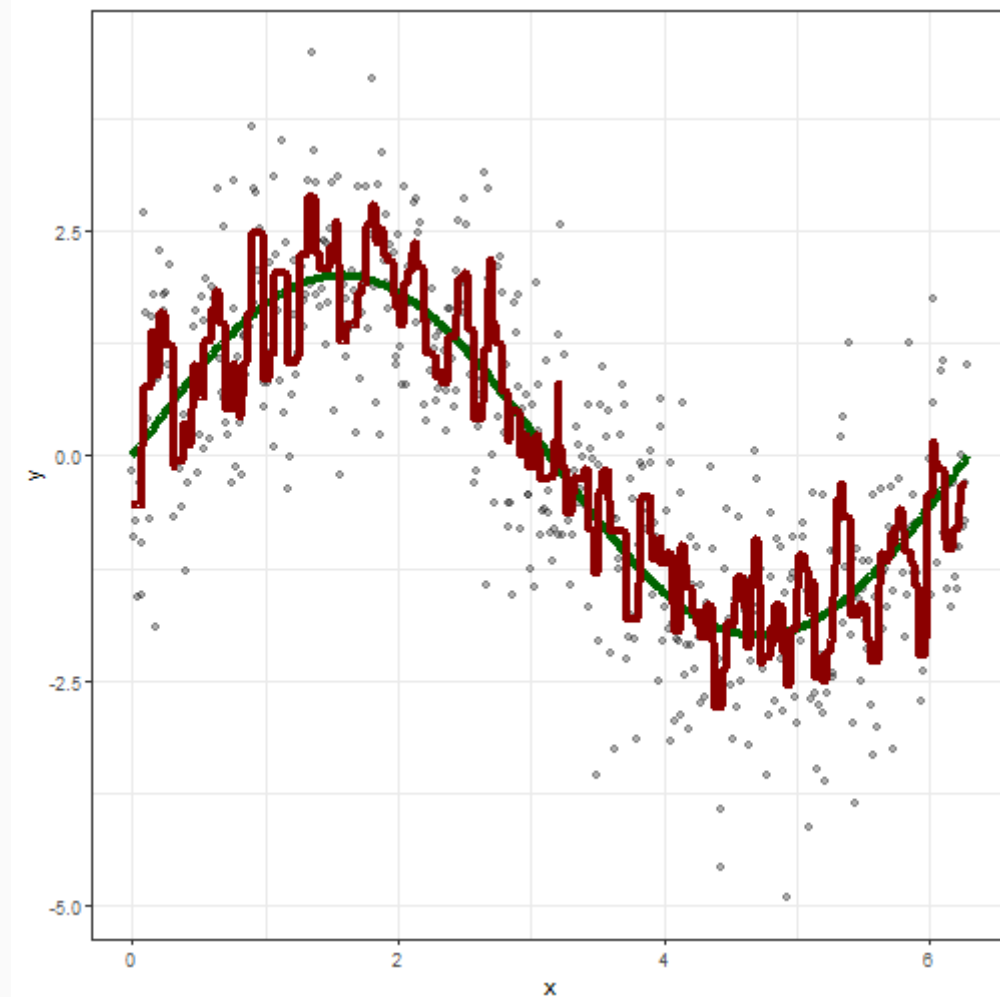
```
# Predictions for the third tree
pred_b3 ← fit_b3 %>% predict(dfr)
# Average the predictions
pred_new ← rowMeans(cbind(pred_b1,
  pred_b2,
  pred_b3))
```



Bagged ensemble with **B = 2**



Bagged ensemble with **B = 3**



**Little** variance reduction might be visible, but we clearly need **a lot more trees**. Let's use the `{ipred}` package for this!

# Using {ipred}

```
bagging(formula, data, control = rpart.control(___),  
        nbagg, ns, coob)
```

- `formula`: a formula as *response ~ feature1 + feature2 + ...*
- `data`: the observation data containing the response and features
- `control`: options to pass to `rpart.control` for the **base learners**
- `nbagg`: the number of bagging iterations **B**, i.e., the number of trees in the ensemble
- `ns`: number of observations to draw for the bootstrap samples (often less than N to save computational time)
- `coob`: a logical indicating whether an **out-of-bag** estimate of the error rate should be computed.

# Out-of-bag (OOB) error

Bootstrap samples are constructed **with** replacement.

Some observations are not present in a bootstrap sample:

- they are called the **out-of-bag** observations
- use those to calculate the out-of-bag (OOB) error
- measures **hold-out** error like cross-validation.

Advantage of OOB over cross-validation?

- the OOB error comes **for free** with bagging.

# Out-of-bag (OOB) error

Bootstrap samples are constructed **with** replacement.

Some observations are not present in a bootstrap sample:

- they are called the **out-of-bag** observations
- use those to calculate the out-of-bag (OOB) error
- measures **hold-out** error like cross-validation.

Advantage of OOB over cross-validation?

- the OOB error comes **for free** with bagging.

But, is this a **representative** sample?

```
set.seed(12345)
N ← 100000 ; x ← 1:N
mean(x %in% sample(N,
                    replace = TRUE))
## [1] 0.63349
```

Roughly **37%** of observations are OOB when N is large.

Even more when we sample **< N** observations

```
mean(x %in% sample(N,
                    size = 0.75*N,
                    replace = TRUE))
## [1] 0.52837
```

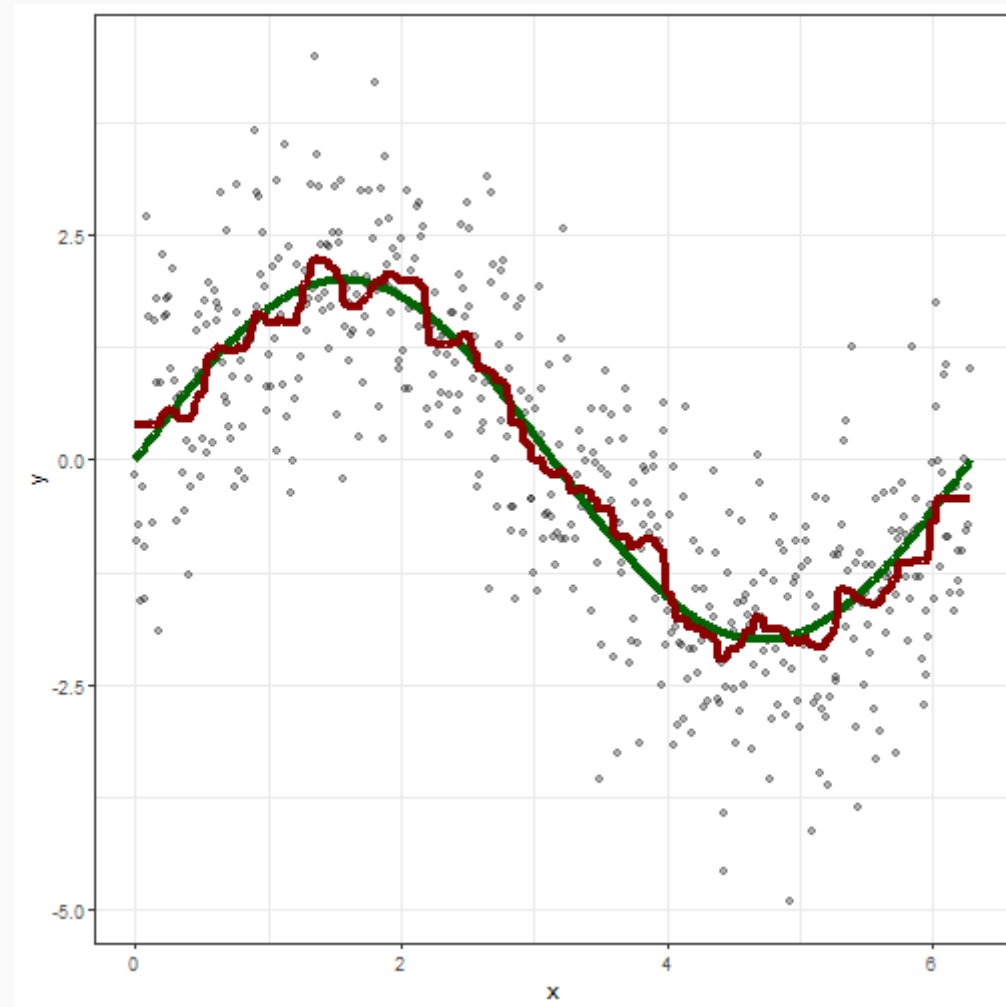
# Bagging properly

```
library(ipred)
set.seed(83946) # reproducibility

# Fit a bagged tree model
fit <- ipred::bagging(formula = y ~ x,
  data = dfr,
  nbagg = 200,
  ns = nrow(dfr),
  coob = TRUE,
  control = rpart.control(
    maxdepth = 20,
    minsplit = 40,
    minbucket = 20,
    cp = 0)
  )

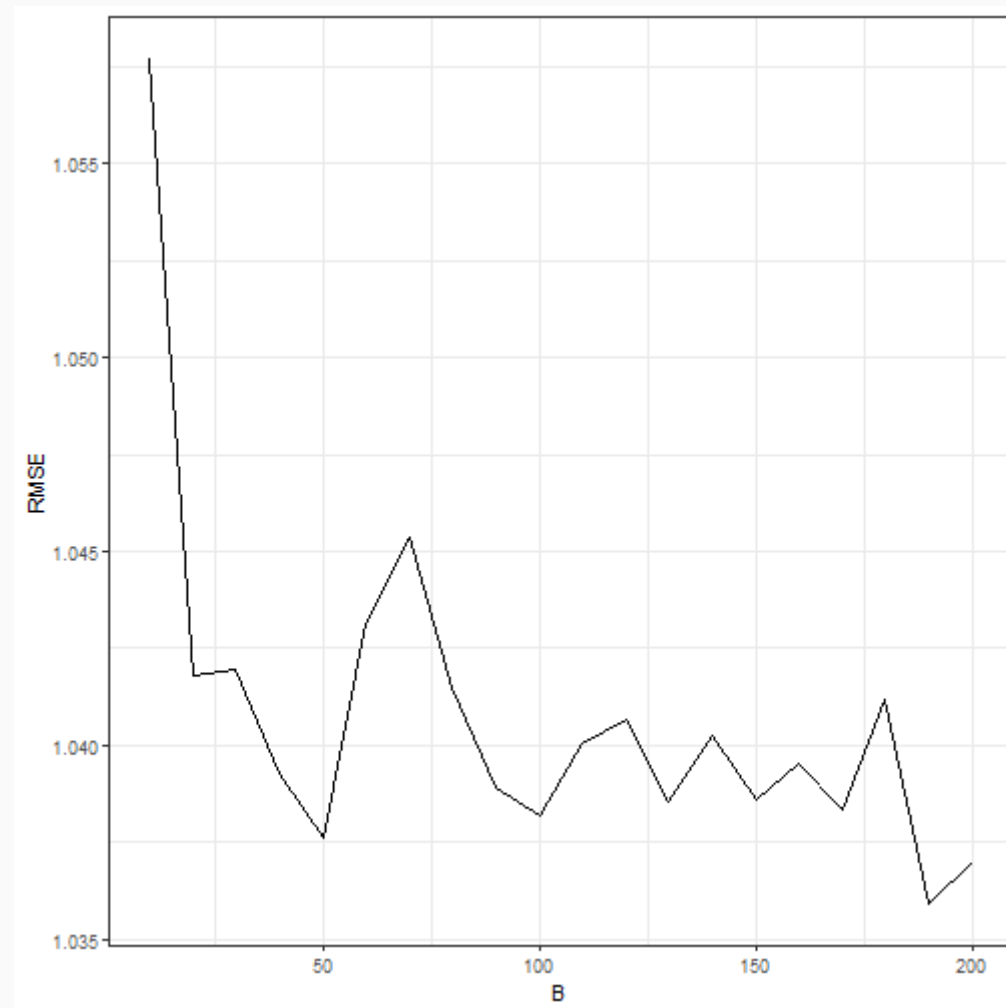
# Predict from this model
pred <- predict(fit, dfr)
```

With 200 trees we can see the **variance reduction**!



# Evolution of the OOB error

```
set.seed(98765) # reproducibility
# Define a grid for B
nbags ← 10*(1:20)
oob ← rep(0, length(nbags))
# Fit a bagged tree model
for(i in 1:length(nbags)){
  fit ← ipred::bagging(formula = y ~ x,
    data = dfr,
    nbagg = nbags[i],
    ns = nrow(dfr),
    coob = TRUE,
    control = rpart.control(
      maxdepth = 20,
      minsplit = 40,
      minbucket = 20,
      cp = 0)
  )
  oob[i] ← fit$err
}
```





## Your turn

Use `{ipred}` to fit a **bagged** tree ensemble for the toy **classification** problem with data `dfc`.

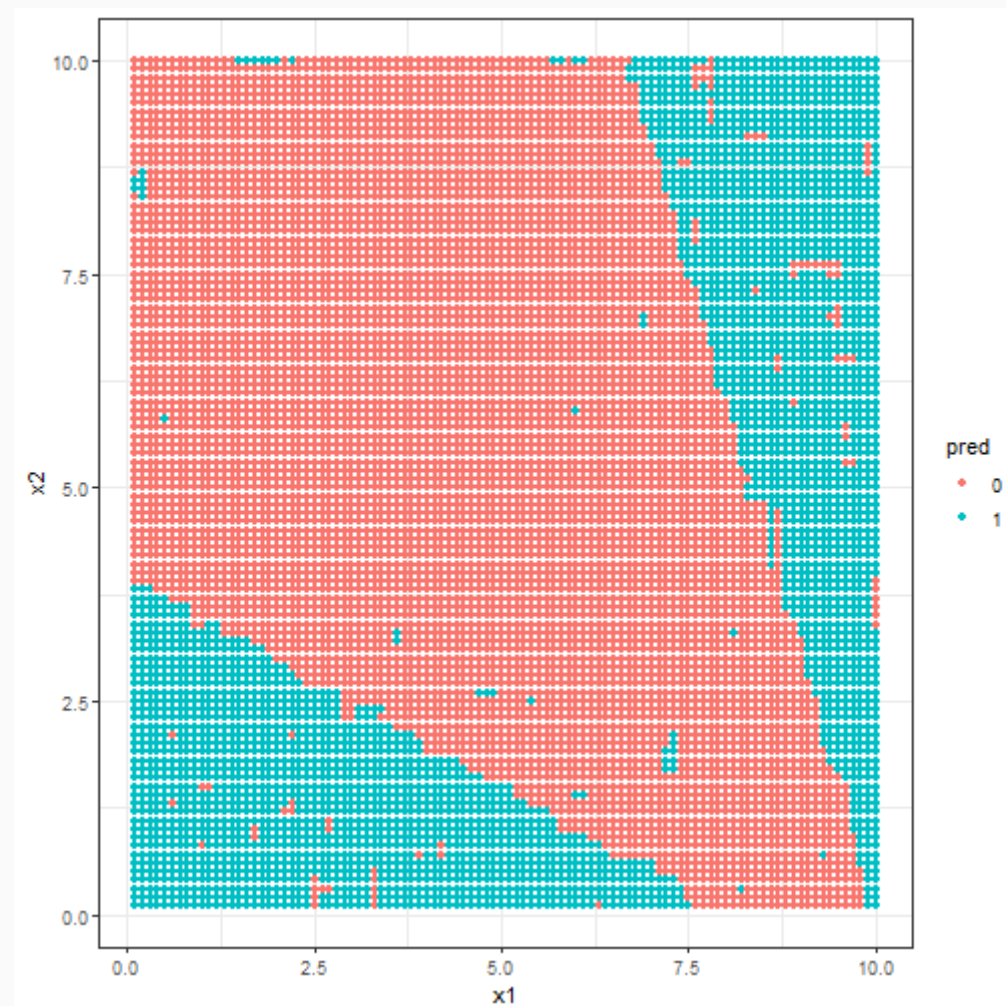
**Q:** experiment with the `nbagg` and `control` parameters to see their effect on the predictions.

**Q:** the following parameter settings seem to produce a decent fit.

```
set.seed(98765) # reproducibility

# Fit a bagged tree model
fit <- ipred::bagging(formula = y ~ x1 + x2,
  data = dfc,
  nbagg = 100,
  ns = nrow(dfc),
  control = rpart.control(
    maxdepth = 20,
    minsplit = 10,
    minbucket = 5,
    cp = 0)
)
```

```
# Predict from this model
pred <- predict(fit,
  newdata = dfc,
  type = 'class',
  aggregation = 'majority'
)
```





# From bagging to random forests

---

# Problem of dominant features



A downside of bagging is that **dominant features** can cause individual trees to have a **similar structure**

- known as **tree correlation**

Remember the **feature importance** results discussed earlier for the MTPL data?

- `bm` is a very dominant variable
- `ageph` was rather important
- `power` also, but to a lesser degree.

Problem?

- bagging gets its predictive performance from **variance reduction**
- however, this reduction  when tree correlation 
- dominant features therefore **hurt** the predictive performance of a bagged ensemble!

Solution?

- **Random forest.**

# Random forest

**Random forest** is a modification on bagging to get an ensemble of **de-correlated** trees.

Process is very similar to bagging, with one small **trick**:

- before each split, select a **subset of features** at random as candidate features for splitting
- this essentially decorrelates the trees in the ensemble, improving predictive performance
- the number of candidates is typically considered a tuning parameter

**Bagging** introduces randomness in the **rows** of the data.

**Random forest** introduces randomness in the **rows** and **columns** of the data.

Many **packages** available, but a couple of popular ones:

- {randomForest}: standard for regression and classification, but not very fast
- {randomForestSRC}: fast OpenMP implementation for survival, regression and classification
- {ranger}: fast C++ implementation for survival, regression and classification.



## Your turn

**Q:** let's investigate the issue of dominant features.

1. Take **two bootstrap samples** from the **MTPL** data.
2. Fit a regression tree of **depth = 3** to each sample.
3. Check the resulting tree structures.

## Q1: two bootstrap samples

```
set.seed(486291) # reproducibility

# Generate the first bootstrapped sample
bsample_1 <- mtpl %>% nrow %>%
  sample(replace = TRUE)

# Generate another bootstrapped sample
bsample_2 <- mtpl %>% nrow %>%
  sample(replace = TRUE)

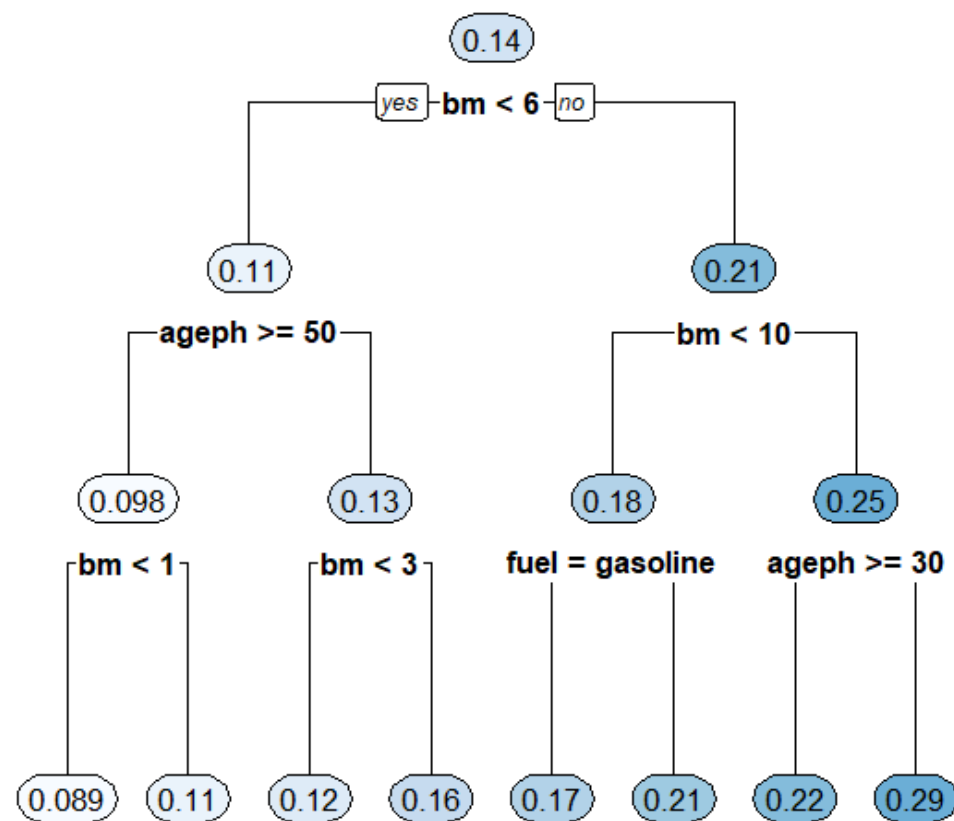
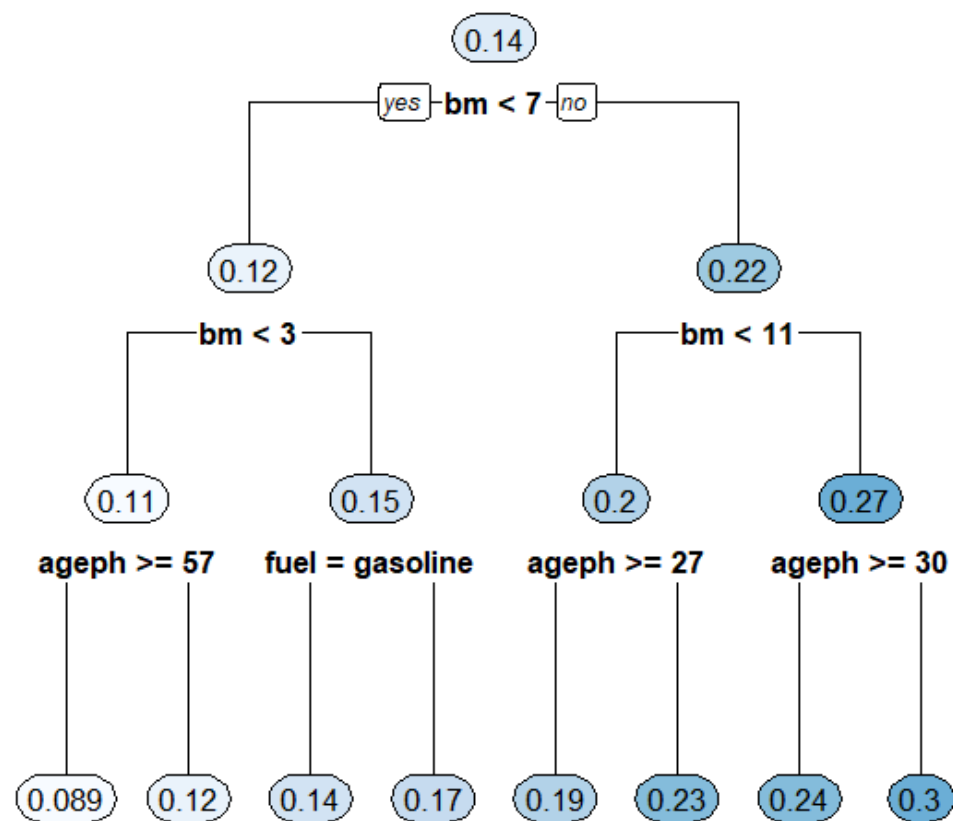
# Use the indices to sample the data
mtpl_b1 <- mtpl %>% dplyr::slice(bsample_1)
mtpl_b2 <- mtpl %>% dplyr::slice(bsample_2)
```

## Q2: Poisson regression tree for each sample

```
fit_b1 <- rpart(formula =
  cbind(expo,nclaims) ~
  ageph + agec + bm + power +
  coverage + fuel + sex + fleet + use,
  data = mtpl_b1,
  method = 'poisson',
  control = rpart.control(
    maxdepth = 3,
    minsplit = 2000,
    minbucket = 1000,
    cp = 0))

fit_b2 <- rpart(formula =
  cbind(expo,nclaims) ~
  ageph + agec + bm + power +
  coverage + fuel + sex + fleet + use,
  data = mtpl_b2,
  method = 'poisson',
  control = rpart.control(
    maxdepth = 3,
    minsplit = 2000,
    minbucket = 1000,
    cp = 0))
```

Q3: the resulting tree structures



# Using {ranger}

```
ranger(formula, data, num.trees, mtry, min.node.size, max.depth,  
       replace, sample.fraction, oob.error, num.threads, seed)
```

- `formula`: a formula as *response ~ feature1 + feature2 + ...*
- `data`: the observation data containing the response and features
- `num.trees`: the number of **trees** in the ensemble
- `mtry`: the number of **candidate** features for splitting
- `min.node.size` and `max.depth`: minimal leaf node size and maximal depth for the individual trees
- `replace` and `sample.fraction`: sample with/without replacement and fraction of observations to sample
- `oob.error`: boolean indication to calculate the **OOB** error
- `num.threads` and `seed`: number of threads and random seed.

# Tuning strategy for random forests

Many **tuning** parameters in a **random forest**:

- number of trees
- number of candidates for splitting
- max tree depth
- minimum leaf node size
- sample fraction.

Construct a full **Cartesian** grid via `expand.grid`:

```
search_grid <- expand.grid(  
  num.trees = c(100,200),  
  mtry = c(3,6,9),  
  min.node.size = c(0.001,0.01)*nrow(mtpl),  
  error = NA  
)
```

```
print(search_grid)  
##      num.trees mtry min.node.size error  
## 1         100    3      163.231    NA  
## 2         200    3      163.231    NA  
## 3         100    6      163.231    NA  
## 4         200    6      163.231    NA  
## 5         100    9      163.231    NA  
## 6         200    9      163.231    NA  
## 7         100    3     1632.310    NA  
## 8         200    3     1632.310    NA  
## 9         100    6     1632.310    NA  
## 10        200    6     1632.310    NA  
## 11         100    9     1632.310    NA  
## 12        200    9     1632.310    NA
```



# Tuning strategy for random forests (cont.)

Perform a **grid search** and track the **OOB error**:

```
library(ranger)
for(i in seq_len(nrow(search_grid))) {
  # fit a random forest for the ith combination
  fit <- ranger(
    formula = nclaims ~
      ageph + agec + bm + power +
      coverage + fuel + sex + fleet + use,
    data = mtpl,
    num.trees = search_grid$num.trees[i],
    mtry = search_grid$mtry[i],
    min.node.size = search_grid$min.node.size[i],
    replace = TRUE,
    sample.fraction = 0.75,
    verbose = FALSE,
    seed = 54321
  )
  # get the OOB error
  search_grid$error[i] <- fit$prediction.error
}
```

```
search_grid %>% arrange(error)
##      num.trees mtry min.node.size      error
## 1         200    3      1632.310 0.1332844
## 2         100    3      1632.310 0.1333003
## 3         200    6      1632.310 0.1333551
## 4         200    9      1632.310 0.1333689
## 5         100    6      1632.310 0.1333754
## 6         100    9      1632.310 0.1333862
## 7         200    3       163.231 0.1337361
## 8         100    3       163.231 0.1338148
## 9         200    6       163.231 0.1341431
## 10        100    6       163.231 0.1342326
## 11        200    9       163.231 0.1343189
## 12        100    9       163.231 0.1344154
```

What does the prediction error **measure** actually?

The **Mean Squared Error**, but does that make sense for us?

# Random forests for actuaries



All available random forest packages only support **standard regression** based on the **Mean Squared Error**

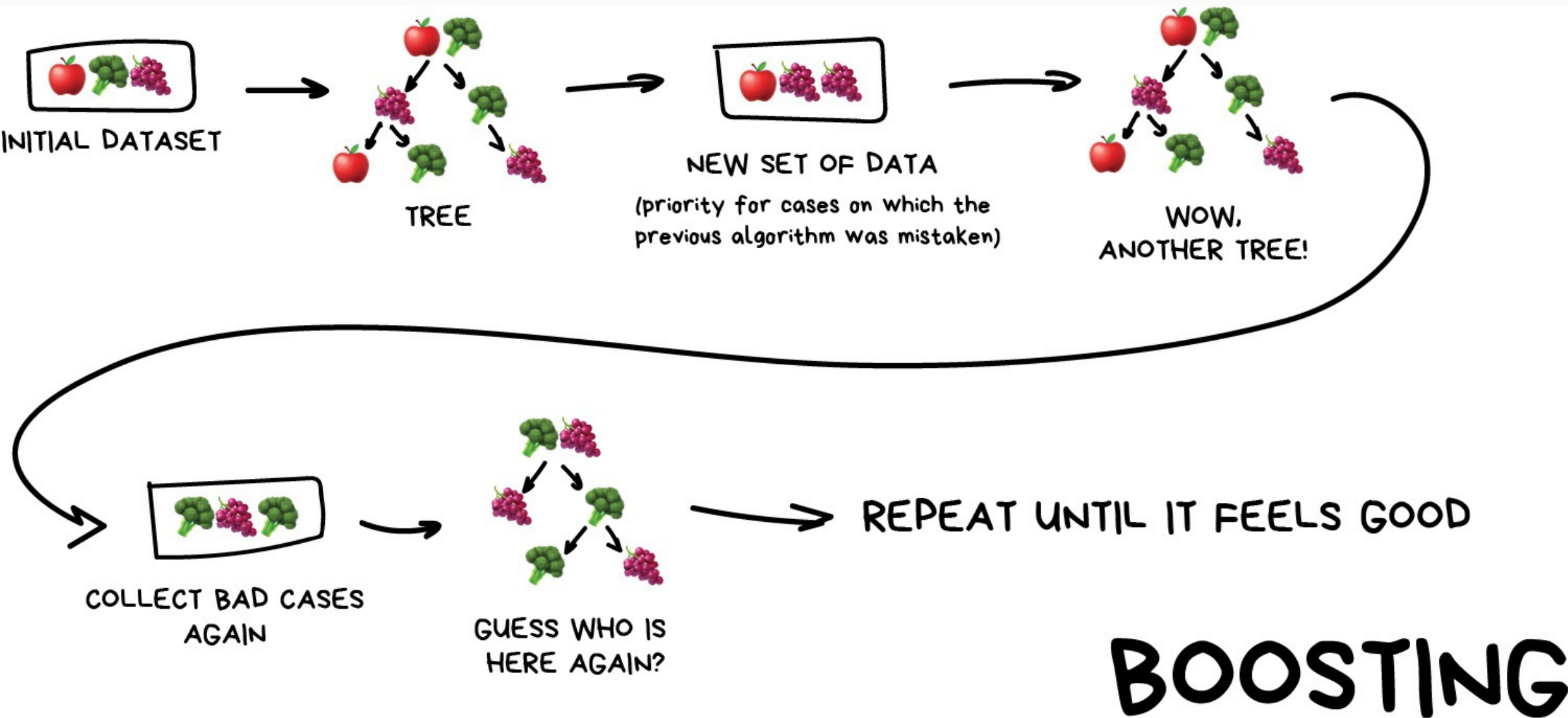
- no Poisson, Gamma or log-normal loss functions available
- bad news for actuaries.

Possible solution: the {distRforest} package on Roel's [GitHub](#) 

- based on {rpart} which supports **Poisson** regression (as we have seen before)
- extended to support **Gamma** and **log-normal** deviance as loss function
- extended to support **random forest** generation
- this package is used in [Henckaerts et al. \(2020\)](#).

# (Stochastic) Gradient Boosting Machines

---



# Boosting vs. bagging

Similar to bagging, boosting is a **general technique** to create an **ensemble** of any type of base learner.

However, there are some **key differences** between both approaches:

With **bagging**:

- **Strong base learners**
  - low bias, high variance
  - for example: deep trees
- **Variance reduction** in ensemble through **averaging**
- **Parallel** approach
  - trees not using information from each other
  - performance thanks to **averaging**
  - low risk for overfitting.

With **boosting**:

- **Weak base learners**
  - low variance, high bias
  - for example: stumps
- **Bias reduction** in ensemble through **updates**
- **Sequential** approach
  - current tree uses information from all past trees
  - performance thanks to **rectifying** past mistakes
  - high risk for overfitting.

# GBM: stochastic gradient boosting with trees

We focus on **GBM**:

- stochastic gradient boosting with **decision trees**
- stochastic: **subsampling** in the rows (and columns) of the data
- gradient: optimizing the loss function via **gradient descent**.

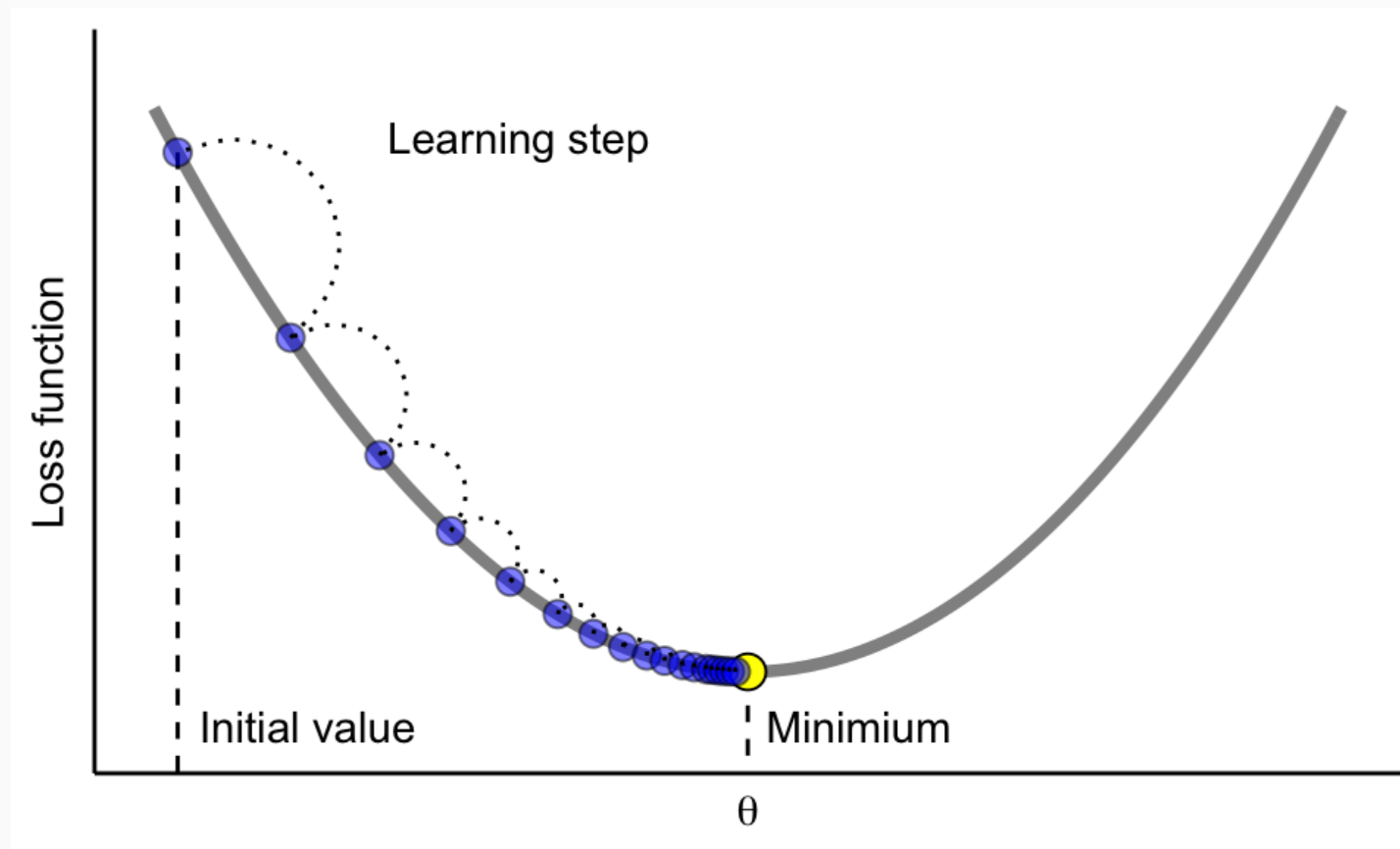


Figure 12.3 from Boehmke & Greenwell [Hands-on machine learning with R](#).

# Stochastic gradient descent

The **learning rate** (also called step size) is very important in gradient descent

- too big: likely to **overshoot** the optimal solution
- too small: **slow** process to reach the optimal solution

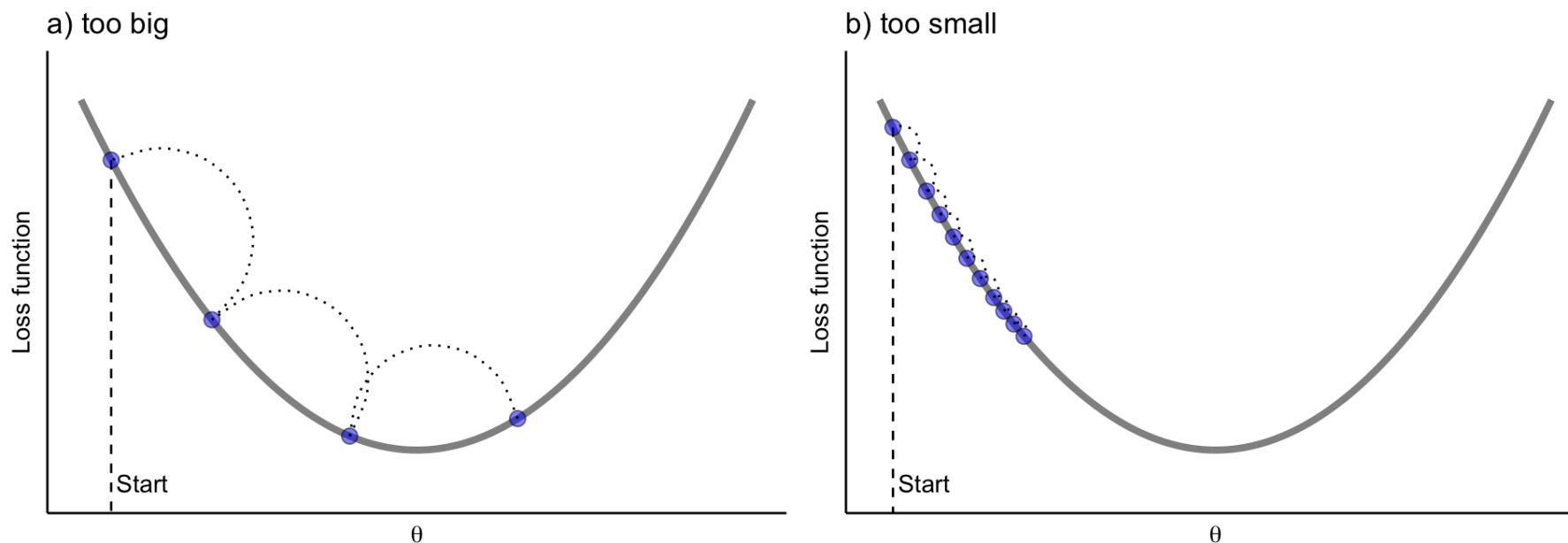


Figure 12.4 from Boehmke & Greenwell [Hands-on machine learning with R](#).

**Subsampling** allows to escape plateaus or local minima for **non-convex** loss functions.

# GBM training process

**Initialize** the model fit with a global average and calculate **pseudo-residuals**.

Do the following **B** times:

- fit a tree of a pre-specified depth to the **pseudo-residuals**
- **update** the model fit and pseudo-residuals with a **shrunk** version
- shrinkage to slow down learning and **prevent** overfitting.

The model fit after B iterations is the **end product**.


Some **popular** packages for stochastic gradient boosting

- {gbm}: standard for regression and classification, but not the fastest
- {gbm3}: faster version of {gbm} via parallel processing, but not backwards compatible
- {xgboost}: efficient implementation with some **extra** elements, for example regularization.



# Using {gbm}

```
gbm(formula, data, distribution, var.monotone, n.trees,  
     interaction.depth, shrinkage, n.minobsinnode, bag.fraction, cv.folds)
```

- `formula`: a formula as *response ~ feature1 + feature2 + ...*  can contain an **offset**!
- `data`: the observation data containing the response and features
- `distribution`: a string specifying which **loss function** to use (gaussian, laplace, tdist, bernoulli, poisson, coxph,...)
- `var.monotone`: vector indicating a monotone increasing (+1), decreasing (-1), or arbitrary (0) relationship
- `n.trees`: the number of **trees** in the ensemble
- `interaction.depth` and `n.minobsinnode`: the maximum tree **depth** and minimum number of leaf node observations
- `shrinkage`: shrinkage parameter applied to each tree in the expansion (also called: **learning rate** or step size)
- `bag.fraction`: fraction of observations to sample for building the next tree
- `cv.folds`: number of cross-validation folds to perform.

# GBM parameters

A lot of parameters at our disposal to **tweak** the GBM.

Some have a **big impact** on the performance and should therefore be **properly tuned**

- `n.trees`: depends very much on the **use case**, ranging from 100's to 10 000's
- `interaction.depth`: **low** values are preferred for boosting to obtain weak base learners
- `shrinkage`: typically set to the lowest possible value that is **computationally** feasible.

**Rule of thumb:** if `shrinkage` ↓ then `ntrees` ↑.

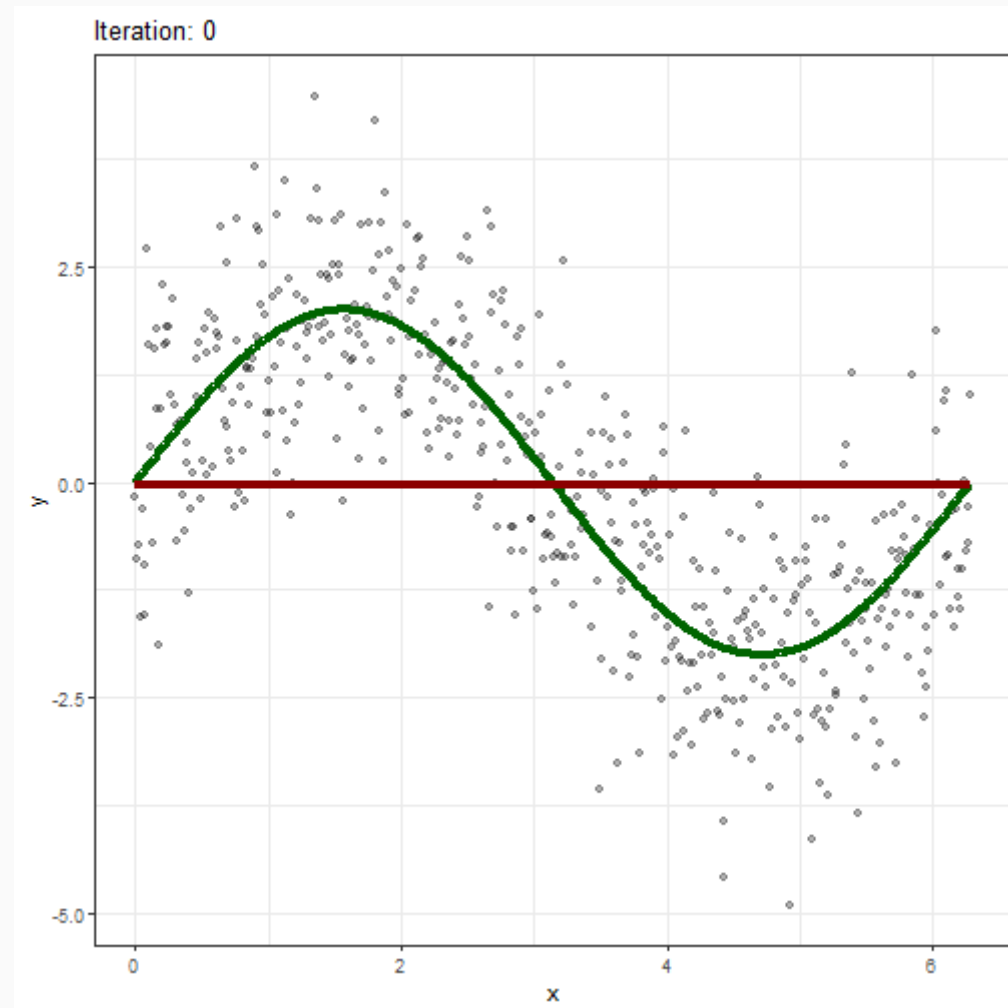
Let's have a look at the **impact** of these tuning parameters.

# GBM parameters (cont.)

Fit a GBM of 10 **stumps**, **without** applying shrinkage:

```
library(gbm)
# Fit the GBM
fit <- gbm(formula = y ~ x,
            data = dfr,
            distribution = 'gaussian',
            n.trees = 10,
            interaction.depth = 1,
            shrinkage = 1
            )

# Predict from the GBM
pred <- predict(fit,
                n.trees = fit$n.trees,
                type = 'response')
```



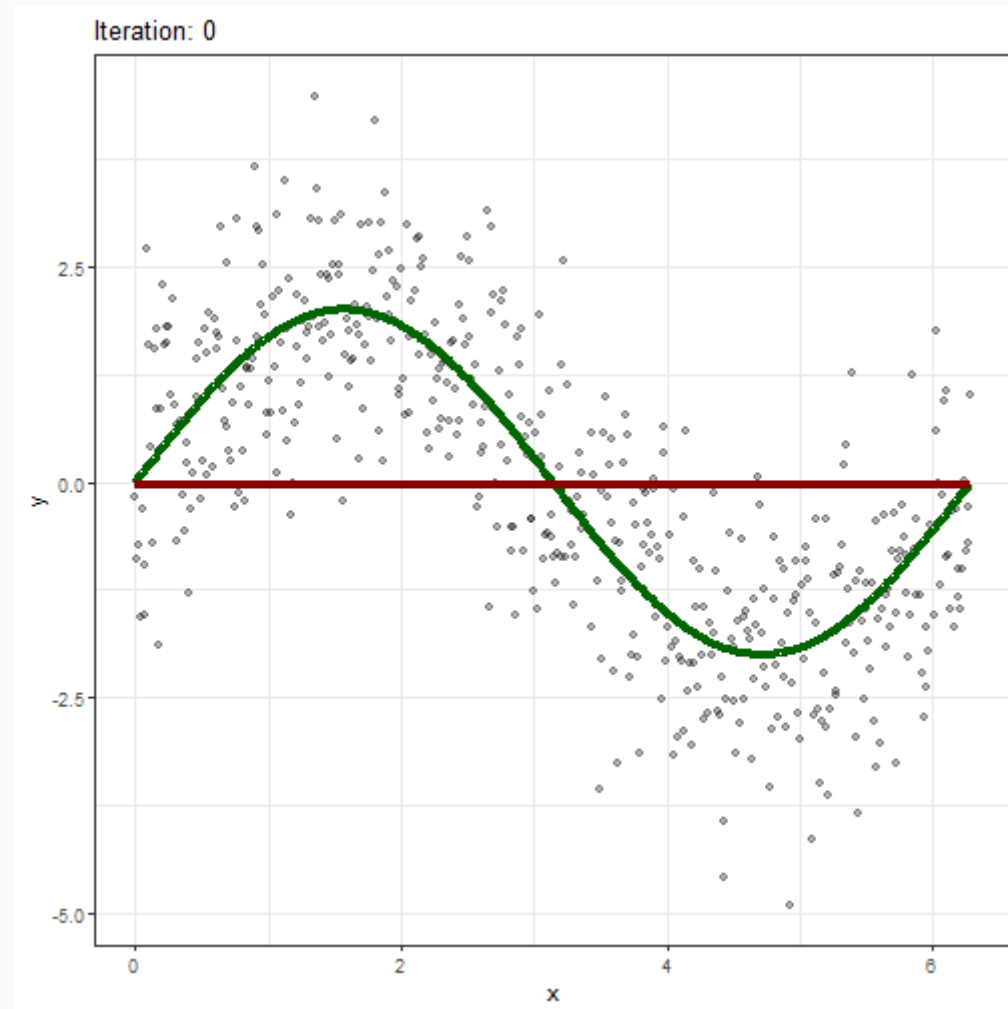
# GBM parameters (cont.)

Fit a GBM of 10 **stumps**, **with** shrinkage:

```
# Fit the GBM
fit <- gbm(formula = y ~ x,
           data = dfr,
           distribution = 'gaussian',
           n.trees = 10,
           interaction.depth = 1,
           shrinkage = 0.1
           )
```

Applying shrinkage **slows down** the learning process:

- **avoids** overfitting
- but we need more trees and **longer** training time.



# GBM parameters (cont.)

Fit a GBM of 10 **shallow** trees, **with** shrinkage:

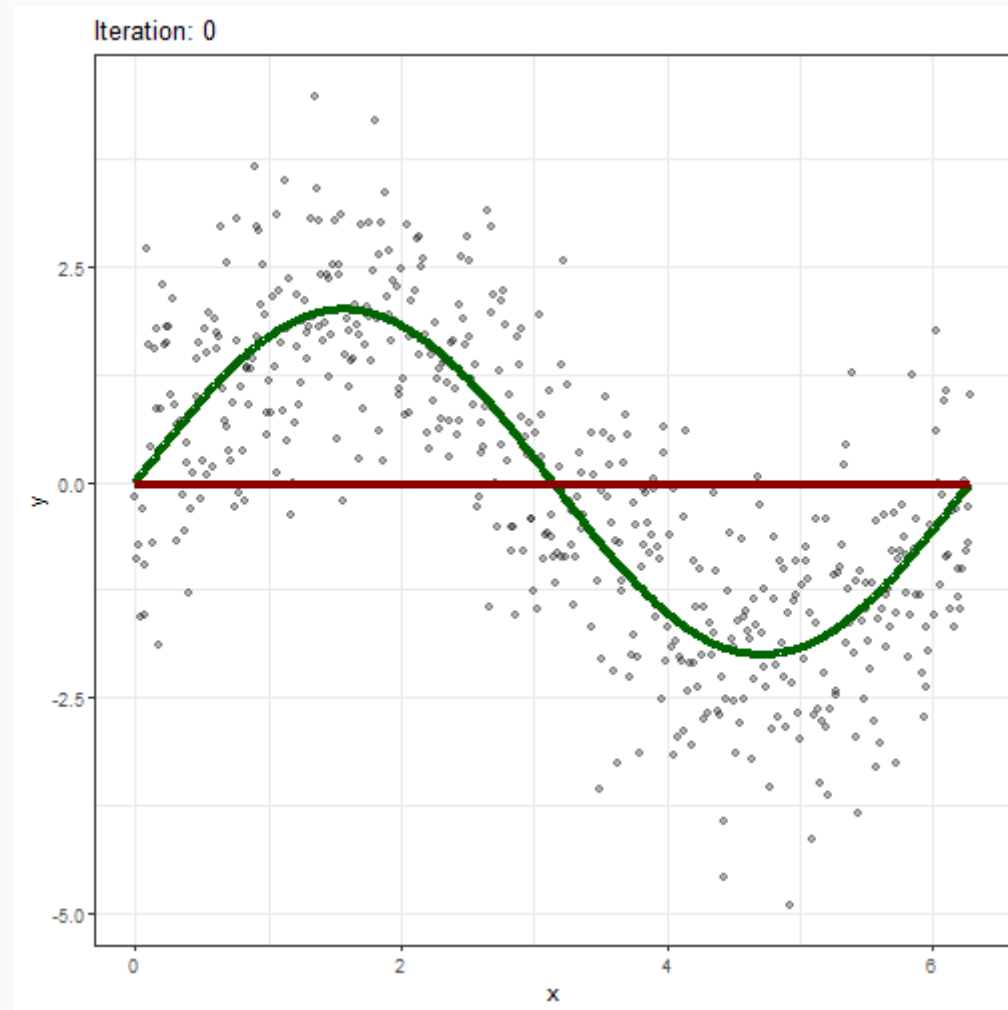
```
# Fit the GBM
fit <- gbm(formula = y ~ x,
  data = dfr,
  distribution = 'gaussian',
  n.trees = 10,
  interaction.depth = 3,
  shrinkage = 0.1
)
```

Increasing tree **depth** allows more versatile splits:

- **faster** learning
- risk of **overfitting** (shrinkage important!)



`interaction.depth > 1` allows for **interactions**!



# GBM parameters (cont.)

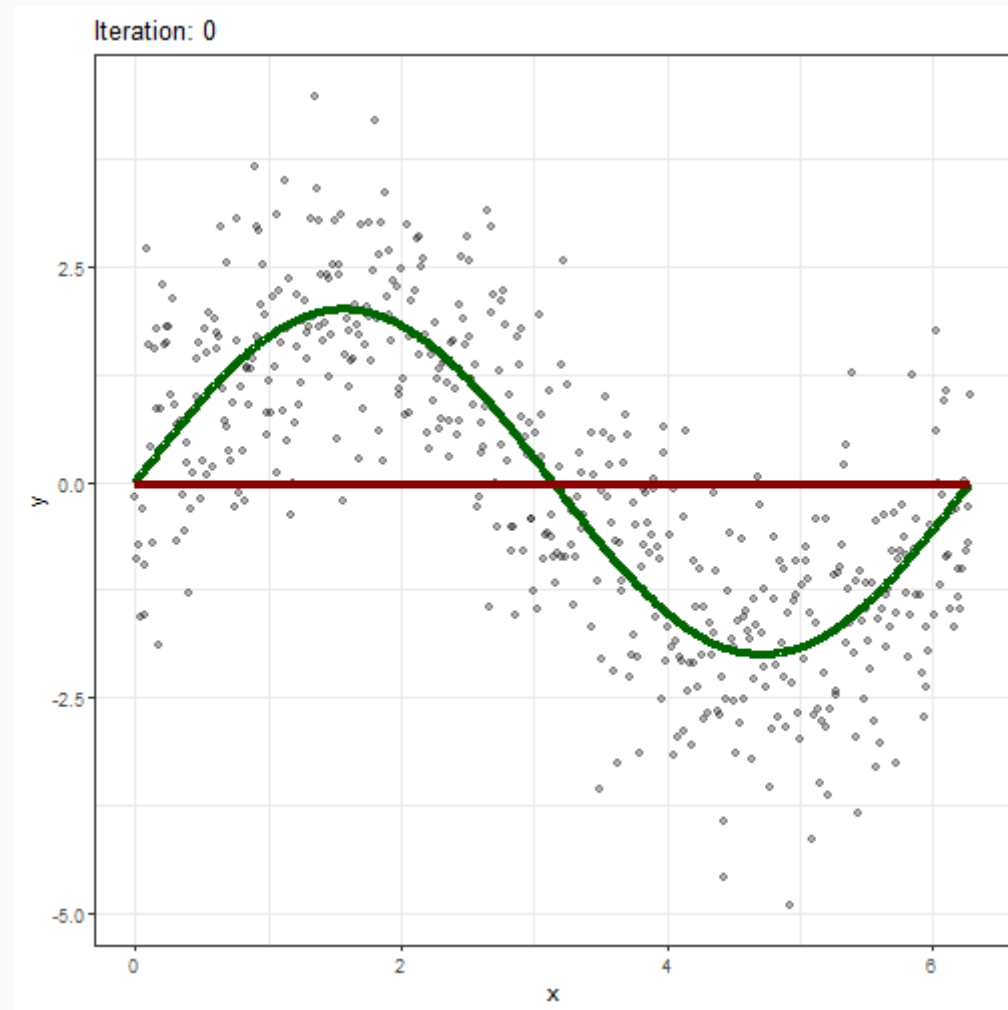
Fit a GBM of 10 **shallow** trees, **without** shrinkage:

```
# Fit the GBM
fit <- gbm(formula = y ~ x,
           data = dfr,
           distribution = 'gaussian',
           n.trees = 10,
           interaction.depth = 3,
           shrinkage = 1
           )
```

The **danger** for overfitting is real!

## Rule of thumb:

- set `shrinkage ≤ 0.01` and adjust `n.trees` accordingly (**computational constraint**).



# Adding trees to the ensemble

Fit a GBM of **300** shallow trees, with shrinkage:

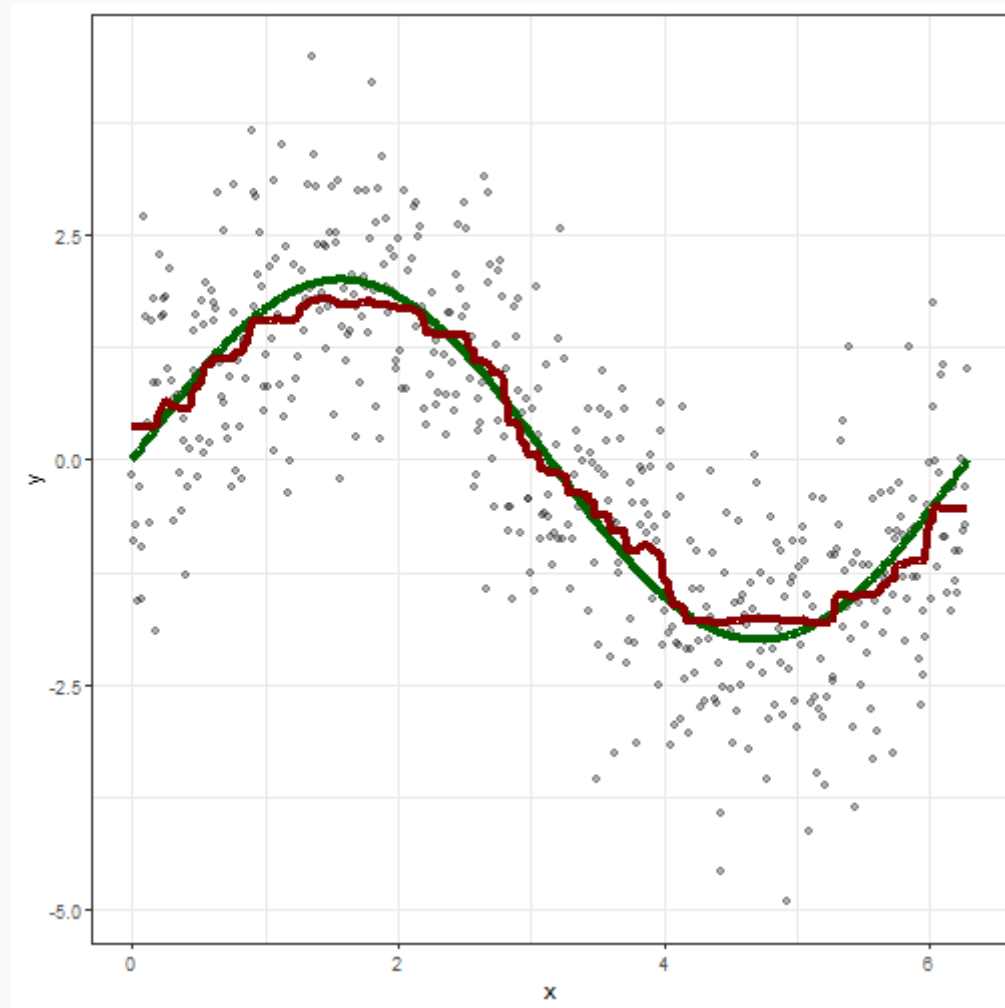
```
# Fit the GBM
fit <- gbm(formula = y ~ x,
  data = dfr,
  distribution = 'gaussian',
  n.trees = 300,
  interaction.depth = 3,
  shrinkage = 0.01
)
```



Look at that nice fit!



Always **beware** of **overfitting** when adding trees!



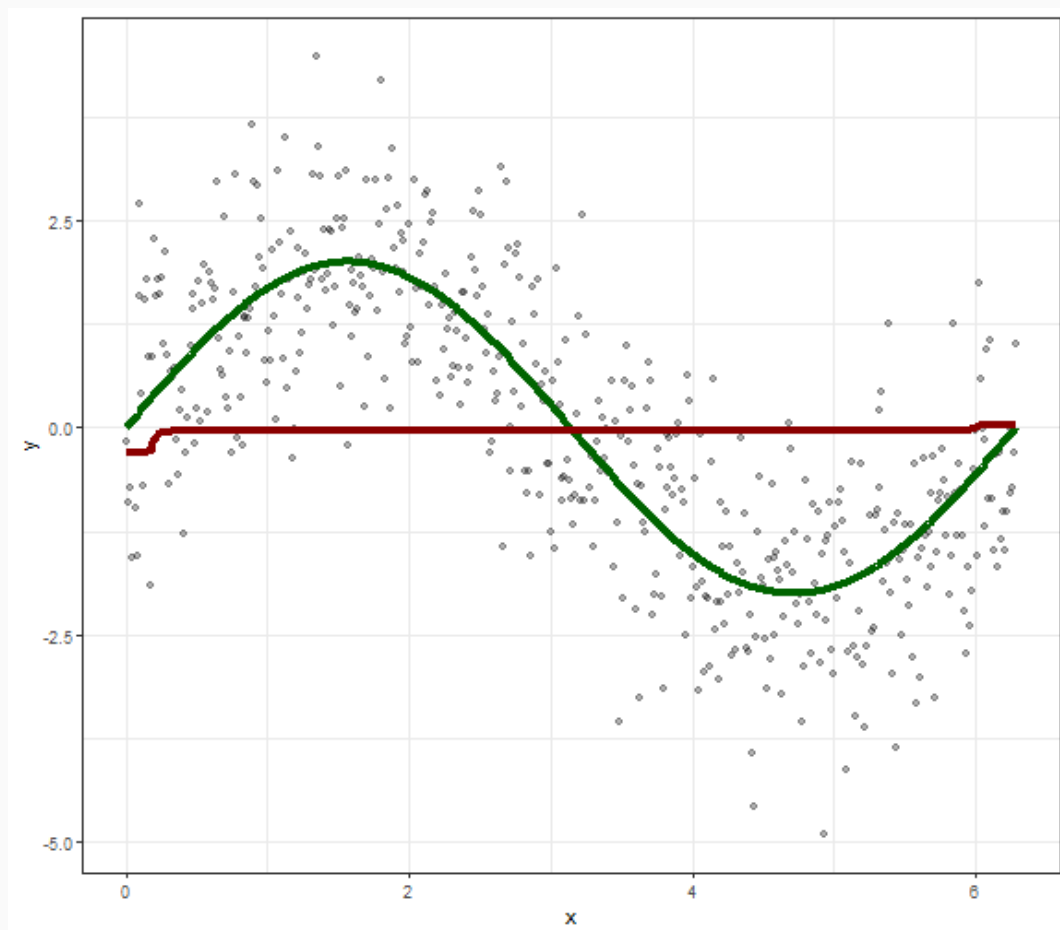


**Q:** use the previous code to **experiment** with your **GBM parameters** of choice (see `?gbm`).

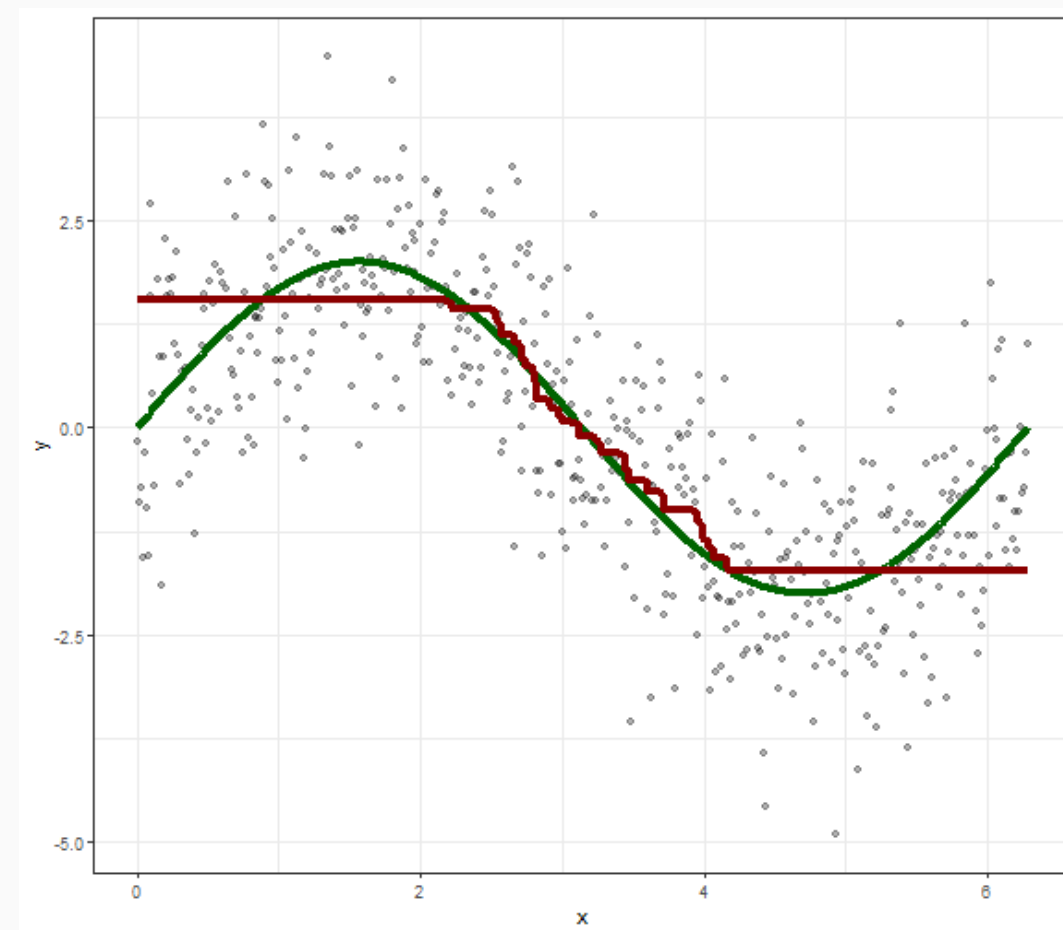
## Your turn



Monotonic increasing fit via `var.monotone = 1`:



Monotonic decreasing fit via `var.monotone = -1`:



# Tuning

---

# Classification with {gbm}

Let's **experiment** with the classification example (`data = dfc`) to get a grip on the tuning of **GBM parameters**

Which `distribution` to specify for **classification**?

Possible **candidates** are:

- `"bernoulli"`: logistic regression for 0-1 outcomes
- `"huberized"`: huberized hinge loss for 0-1 outcomes
- `"adaboost"`: the AdaBoost exponential loss for 0-1 outcomes

**Watch out:** gbm does not take factors as response so you need to **recode y**

- either to a **numeric** in the range [0,1]
- or a **boolean** `TRUE` / `FALSE`

```
dfc <- dfc %>% dplyr::mutate(y_recode = as.integer(as.character(y)))
```

# Different parameter combinations

Set up a grid for the parameters and list to save results:

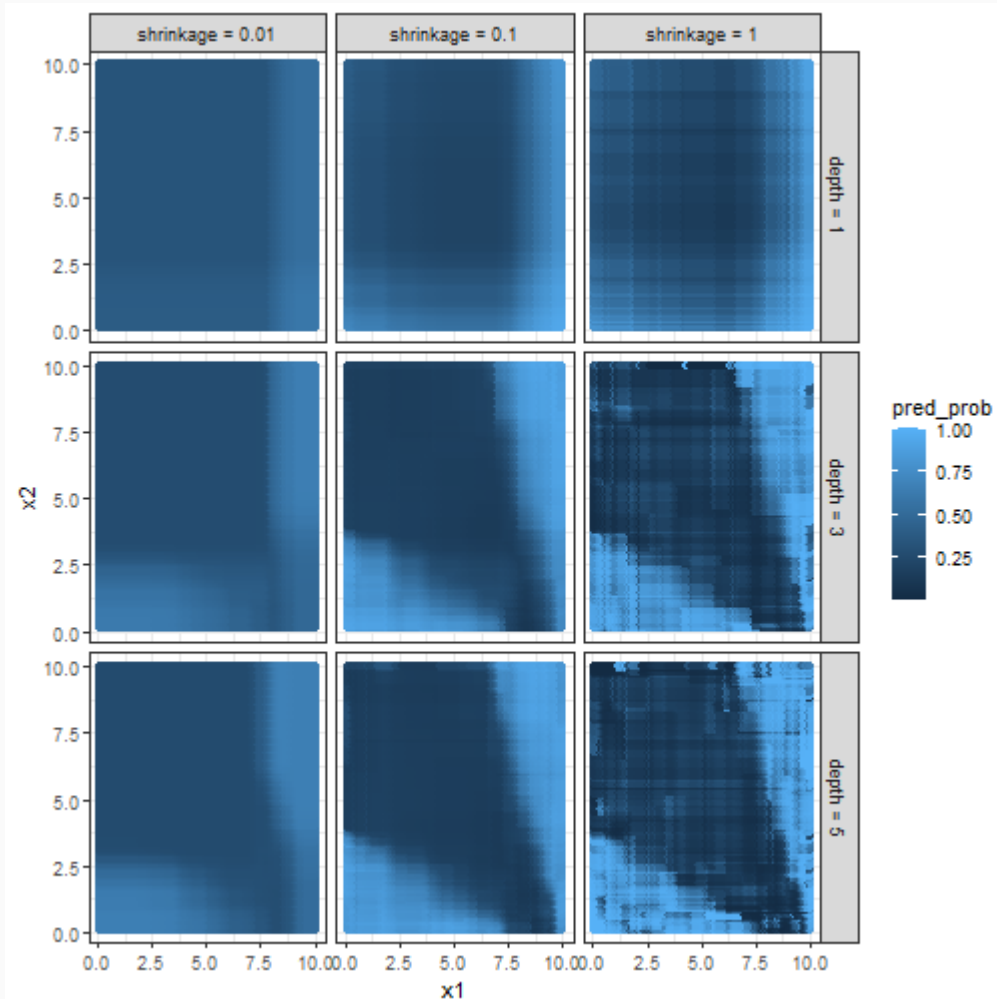
```
ctrl_grid ← expand.grid(depth = c(1,3,5),  
                        shrinkage = c(0.01,0.1,1))  
results ← vector('list', length = nrow(ctrl_grid))
```

Fit different a GBM with 100 trees for each parameter combination:

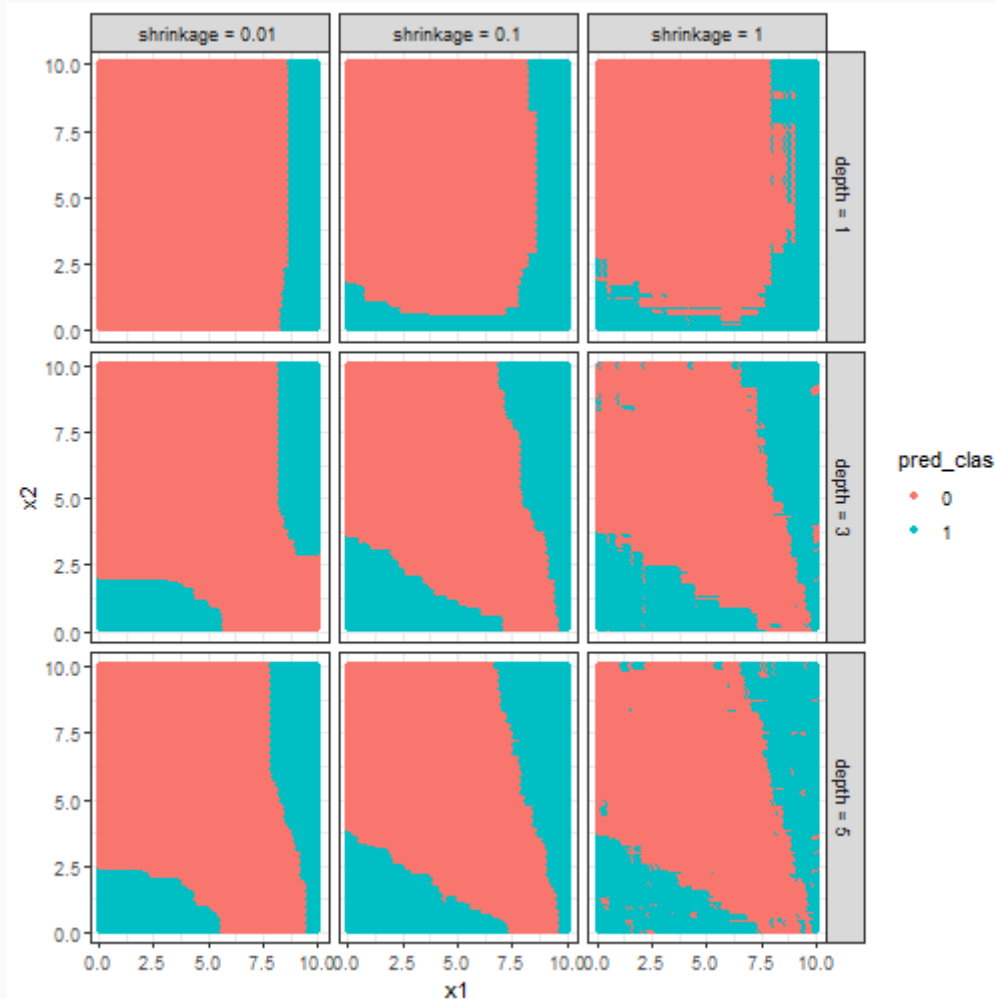
```
for(i in seq_len(nrow(ctrl_grid))) {  
  fit ← gbm(y_recode ~ x1 + x2,  
            data = dfc,  
            distribution = 'bernoulli',  
            n.trees = 100,  
            interaction.depth = ctrl_grid$depth[i],  
            shrinkage = ctrl_grid$shrinkage[i])  
  
  # Save predictions, both the probabilities and the class  
  results[[i]] ← dfc %>% mutate(  
    depth = factor(paste('depth =',ctrl_grid$depth[i]), ordered =TRUE),  
    shrinkage = factor(paste('shrinkage =',ctrl_grid$shrinkage[i]), ordered = TRUE),  
    pred_prob = predict(fit, n.trees = fit$n.trees, type = 'response'),  
    pred_clas = factor(1*(predict(fit, n.trees = fit$n.trees, type = 'response') ≥ 0.5)))  
}
```

# Resulting fits

The predicted **probabilities**



The predicted **classes**





## Your turn

**Q:** complete the code below to find the **optimal combination** of **tuning parameters**.

1. Set up a search grid.
2. Fit a GBM for each combination of parameters.
3. Extract the OOB performance for each GBM.

**Beware:** a fitted `gbm` object returns the **improvements** in OOB error via `$oobag.improve`.

```
my_grid <- expand.grid(____)
my_grid <- my_grid %>% dplyr::mutate(oob_improv = NA)

for(i in seq_len(nrow(my_grid))) {
  fit <- gbm(y_recode ~ x1 + x2,
            data = dfc,
            distribution = 'bernoulli',
            n.trees = ____,
            interaction.depth = ____,
            shrinkage = ____,
            ____ )
  my_grid$oob_improv[i] <- sum(fit$oobag.improve)
}
```

Performing the grid search:

```
my_grid <- expand.grid(depth = c(1,3,5),
                      shrinkage = c(0.01,0.1,1))
my_grid <- my_grid %>% dplyr::mutate(oob_improv = NA)

for(i in seq_len(nrow(my_grid))) {
  fit <- gbm(y_recode ~ x1 + x2,
            data = dfc,
            distribution = 'bernoulli',
            n.trees = 100,
            interaction.depth = my_grid$depth[i],
            shrinkage = my_grid$shrinkage[i])
  my_grid$oob_improv[i] <- sum(fit$oobag.improve)
}
```

The results:

```
my_grid %>% dplyr::arrange(desc(oob_improv))
##   depth shrinkage  oob_improv
## 1     5      0.10  0.179728774
## 2     3      0.10  0.175448387
## 3     5      0.01  0.123874131
## 4     3      0.01  0.100252315
## 5     1      0.10  0.067270554
## 6     1      0.01  0.039404600
## 7     3      1.00  0.036592026
## 8     1      1.00  0.006891027
## 9     5      1.00 -0.094434251
```

Another tuning option is to set `cv.folds > 0` and track the **cross-validation** error via `fit$cv.error`.

That would be a more general approach but also **more time-consuming**.

# Claim frequency and severity modeling with {gbm}

---



# Claim frequency modeling

```
set.seed(76539) # reproducibility
fit <- gbm(formula = nclaims ~
            ageph + agec + bm + power +
            coverage + fuel + sex + fleet + use +
            offset(log(expo)),
            data = mtpl,
            distribution = 'poisson',
            var.monotone = c(0,0,1,0,0,0,0,0,0),
            n.trees = 200,
            interaction.depth = 3,
            n.minobsinnode = 1000,
            shrinkage = 0.1,
            bag.fraction = 0.75,
            cv.folds = 0
            )
```

- Include the log of exposure as an **offset**.
- Specify the **Poisson** distribution for the target.
- Impose a **monotonically increasing** constraint on `bm`.
- Perform **stochastic** gradient boosting with `bag.fraction < 1`.

# Inspecting the individual trees

```
fit %>%
  pretty.gbm.tree(i.tree = 1) %>%
  print(digits = 4)
```

##	SplitVar	SplitCodePred	LeftNode	RightNode	MissingNode	ErrorReduction	Weight
## 0	2	6.500000	1	5	9	164.94	122423
## 1	2	1.500000	2	3	4	27.29	95743
## 2	-1	-0.027378	-1	-1	-1	0.00	66472
## 3	-1	0.004986	-1	-1	-1	0.00	29271
## 4	-1	-0.017484	-1	-1	-1	0.00	95743
## 5	2	10.500000	6	7	8	15.42	26680
## 6	-1	0.035538	-1	-1	-1	0.00	17014
## 7	-1	0.065038	-1	-1	-1	0.00	9666
## 8	-1	0.046226	-1	-1	-1	0.00	26680
## 9	-1	-0.003599	-1	-1	-1	0.00	122423
##	Prediction						
## 0	-0.003599						
## 1	-0.017484						
## 2	-0.027378						
## 3	0.004986						
## 4	-0.017484						
## 5	0.046226						
## 6	0.035538						
## 7	0.065038						

# Feature importance

Applying the `summary` function on a object of class `gbm` shows built-in feature importance results:

```
summary(fit, plotit = FALSE)
##           var      rel.inf
## bm           bm 61.3816154
## ageph        ageph 15.5230257
## power        power  8.4368204
## agec         agec  6.7923989
## fuel         fuel  3.2694791
## sex          sex   2.7324191
## coverage    coverage 1.3318440
## fleet        fleet  0.3382497
## use         use   0.1941476
```

# Partial dependence plot

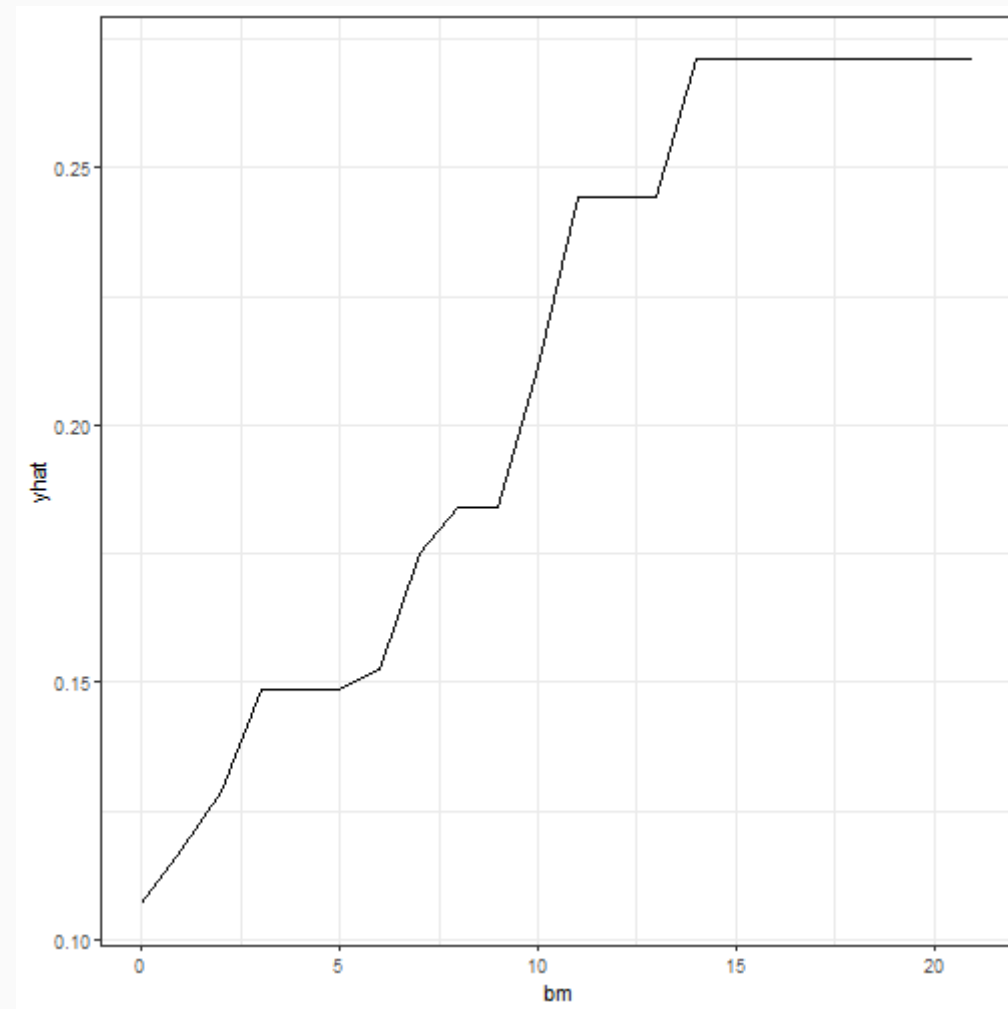
Use the following **helper function** for the pdps:

```
pred.fun <- function(object,newdata){  
  mean(predict(object, newdata,  
              n.trees = object$n.trees,  
              type = 'response'))  
}
```

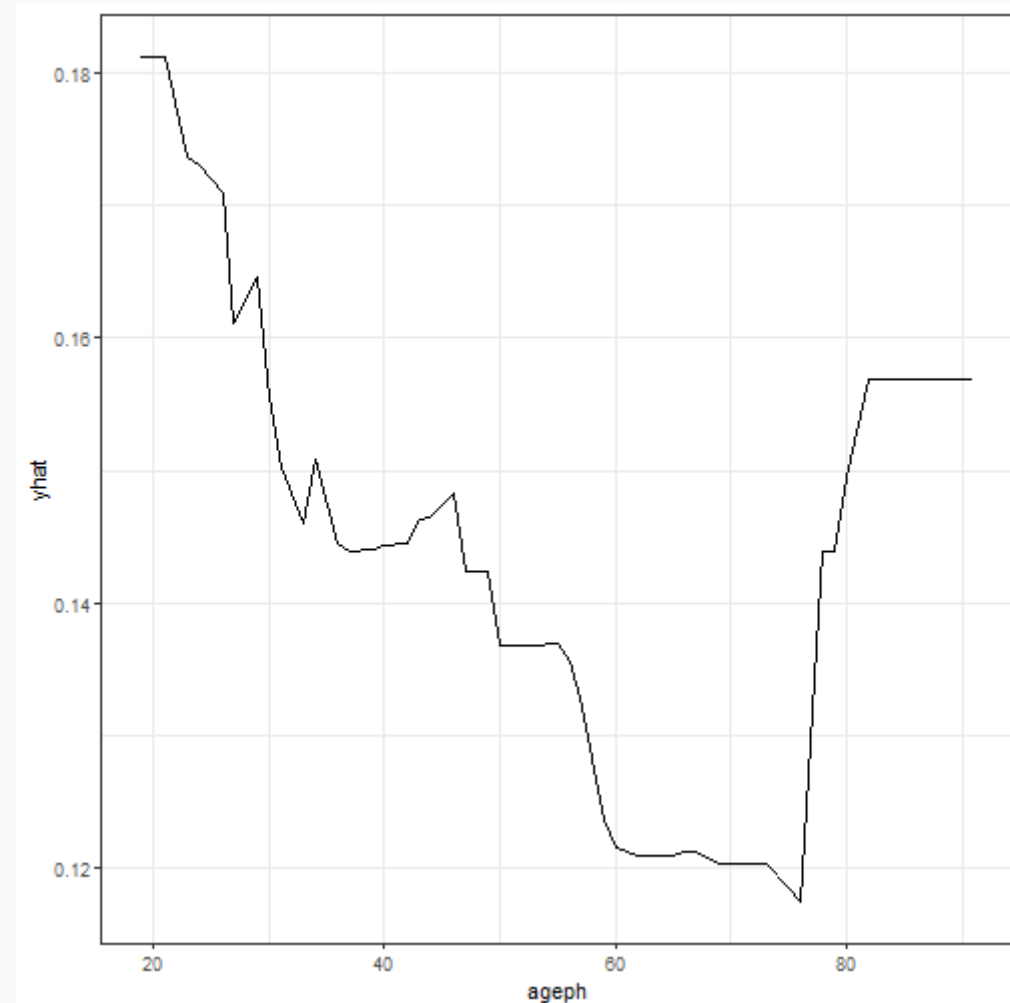
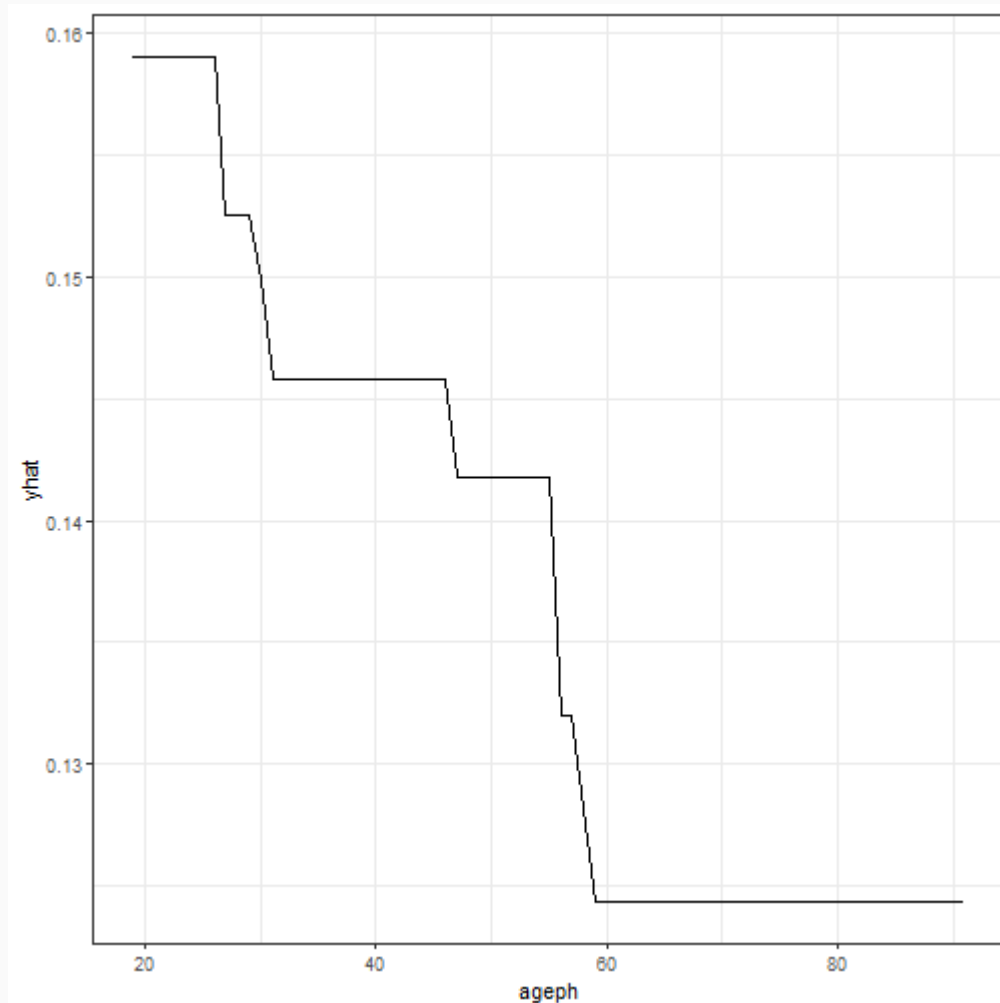
Partial dependence of the **bonus-malus level**:

```
set.seed(48927)  
pdp_ids <- mtlp %>% nrow %>% sample(size = 5000)  
fit %>%  
  partial(pred.var = 'bm',  
          pred.fun = pred.fun,  
          train = mtlp[pdp_ids,],  
          recursive= FALSE) %>%  
  autoplot()
```

Notice that the monotonic constraint is satisfied.



# The age effect in a single tree and a gbm




# Claim severity modeling

From the `gbm` help on the `distribution` argument:

Currently available options are "gaussian" (squared error), "laplace" (absolute loss), "tdist" (t-distribution loss), "bernoulli" (logistic regression for 0-1 outcomes), "huberized" (huberized hinge loss for 0-1 outcomes), "adaboost" (the AdaBoost exponential loss for 0-1 outcomes), "poisson" (count outcomes), "coxph" (right censored observations), "quantile", or "pairwise" (ranking measure using the LambdaMart algorithm)

Which to choose for **claim severity**?

Possible solution: the `{gbm}` version on Harry Southworth's [GitHub](#) 

```
install.packages("devtools")
devtools::install_github("harrysouthworth/gbm")
```

# XGBoost

---

# XGBoost

XGBoost stands for e**X**treme **G**radient **B**oosting.

Optimized gradient boosting library: efficient, flexible and portable across multiple languages.

XGBoost follows the same general boosting approach as GBM, but adds some **extra elements**:

- **regularization**: extra protection against overfitting (see Lasso and glmnet on Day 1)
- **early stopping**: stop model tuning when improvement slows down
- **parallel processing**: can deliver huge speed gains
- different **base learners**: boosted GLMs are a possibility
- multiple **languages**: implemented in R, Python, C++, Java, Scala and Julia

XGBoost also allows to **subsample columns** in the data, much like the random forest did

- GBM only allowed subsampling of rows
- XGBoost therefore **unites** boosting and random forest to some extent.

Very **flexible** method with many many parameters, full list can be found [here](#).



# Using {xgboost}

```
xgboost(data, nrounds, early_stopping_rounds, params)
```

- `data`: training data, preferably an `xgb.DMatrix` (also accepts `matrix`, `dgCMatrix`, or name of a local data file)
- `nrounds`: max number of boosting **iterations**
- `early_stopping_rounds`: training with a validation set will **stop** if the performance doesn't improve for k rounds
- `params`: the list of **parameters**
  - `booster`: gbtrees, gblines or dart
  - `objective`: reg:squarederror, binary:logistic, count:poisson, survival:cox, reg:gamma, reg:tweedie, ...
  - `eval_metric`: rmse, mae, logloss, auc, poisson-nloglik, gamma-nloglik, gamma-deviance, tweedie-nloglik, ...
  - `base_score`: initial prediction for all observations (global bias)
  - `nthread`: number of parallel threads used to run XGBoost (defaults to max available)
  - `eta`: **learning rate** or step size used in update to prevent overfitting
  - `gamma`: minimum loss reduction required to make a further partition on a leaf node
  - `max_depth` and `min_child_weight`: maximum depth and minimum leaf node observations
  - `subsample` and `colsample_by*`: subsample rows and columns (bytree, bylevel or bynode)
  - `lambda` and `alpha`: L2 and L1 **regularization** term to prevent overfitting
  - `monotone_constraints`: constraint on variable monotonicity

# Supplying the data to XGBoost

```
xgb.DMatrix(data, info = list())
```

- `data`: a `matrix` object
- `info`: a named list of additional information

```
library(xgboost)
mtpl_xgb ← xgb.DMatrix(data = mtpl %>%
  select(ageph, power, bm, agec, coverage, fuel, sex, fleet, use) %>%
  data.matrix,
  info = list(
    'label' = mtpl$nclaims,
    'base_margin' = log(mtpl$expo)))
```

- Features go into the **data** argument (needs to be converted to a matrix)
- The target and offset are specified via `label` and `base_margin` in **info** respectively

This results in an `xgb.DMatrix` object:

```
print(mtpl_xgb)
## xgb.DMatrix  dim: 163231 x 9  info: label base_margin  colnames: yes
```

# A simple XGBoost model

```
set.seed(86493) # reproducibility
```

```
fit <- xgboost(
```

```
  data = mtlpl_xgb,
```

```
  nrounds = 200,
```

```
  early_stopping_rounds = 20,
```

```
  verbose = FALSE,
```

```
  params = list(
```

```
    booster = 'gbtree',
```

```
    objective = 'count:poisson',
```

```
    eval_metric = 'poisson-nloglik',
```

```
    eta = 0.1, nthread = 1,
```

```
    subsample = 0.75, colsample_bynode = 0.5,
```

```
    max_depth = 3, min_child_weight = 1000,
```

```
    gamma = 0, lambda = 1, alpha = 1
```

```
  )
```

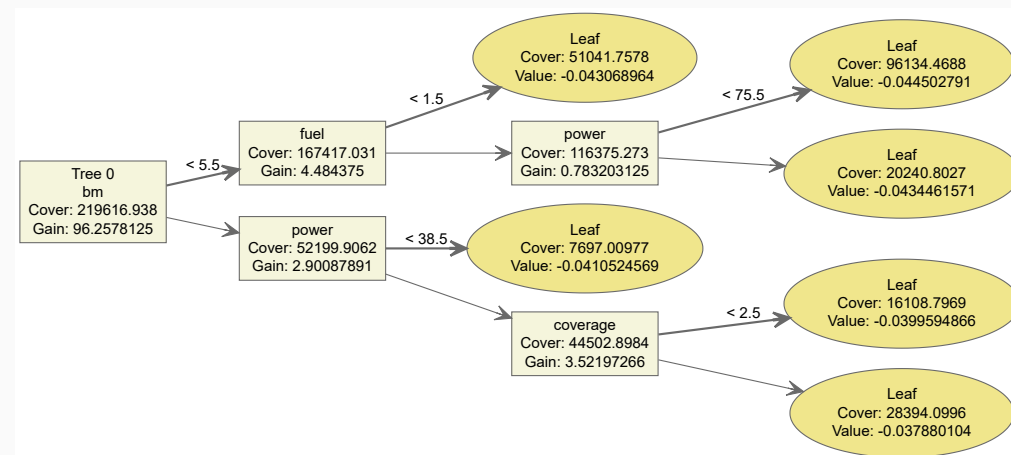
```
)
```

- Fit an XGBoost model to the **xgb.DMatrix** data
- Perform **early stopping** after 20 iterations without improvement
- Use a **decision tree** as base learner
- Choose the **Poisson** distribution for the target
- Stochastic boosting in **rows** and random split candidates in **columns** (like random forest)
- Apply **regularization** comparable to the elastic net penalty in {glmnet}

# Inspecting single trees

- Possible to inspect **single trees** via `xgb.plot.tree`:
  - note that the trees are **0-indexed**
  - 0 returns first tree, 1 returns second tree,...
  - can also supply a vector of indexes

```
xgb.plot.tree(  
    feature_names = colnames(mtpl_xgb),  
    model = fit,  
    trees = 0  
)
```



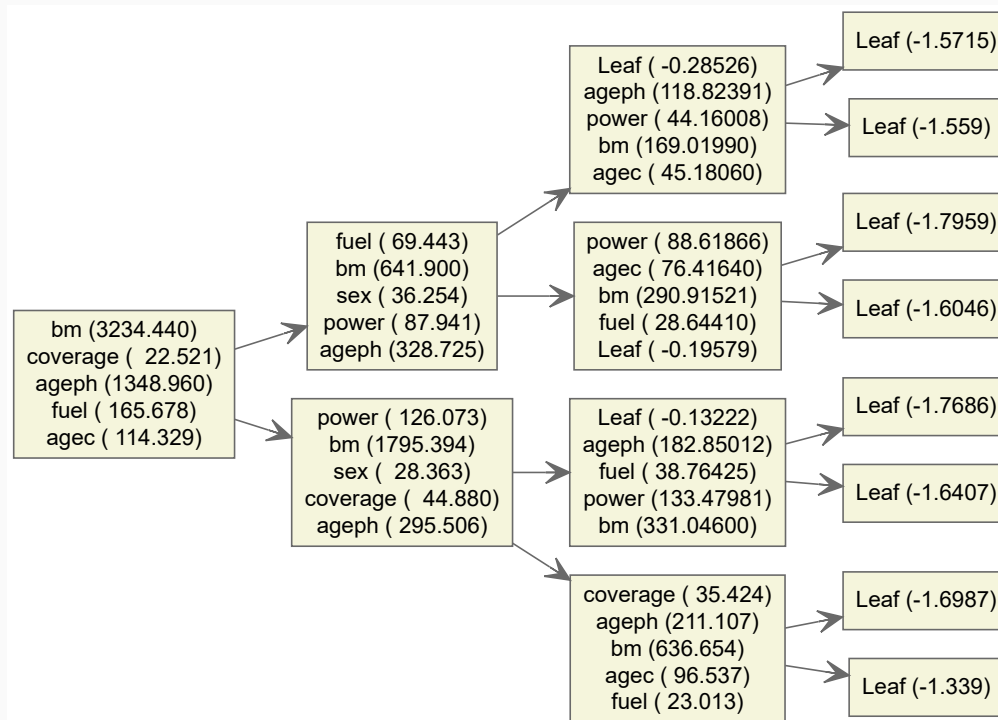
# XGBoost in one tree

- Get a **compressed view** of an XGBoost model via

`xgb.plot.multi.trees:`

- compressing an ensemble of trees into a single **tree-graph** representation
- goal is to improve the interpretability

```
xgb.plot.multi.trees(  
    model = fit,  
    feature_names = colnames(mtpl_xgb)  
)
```



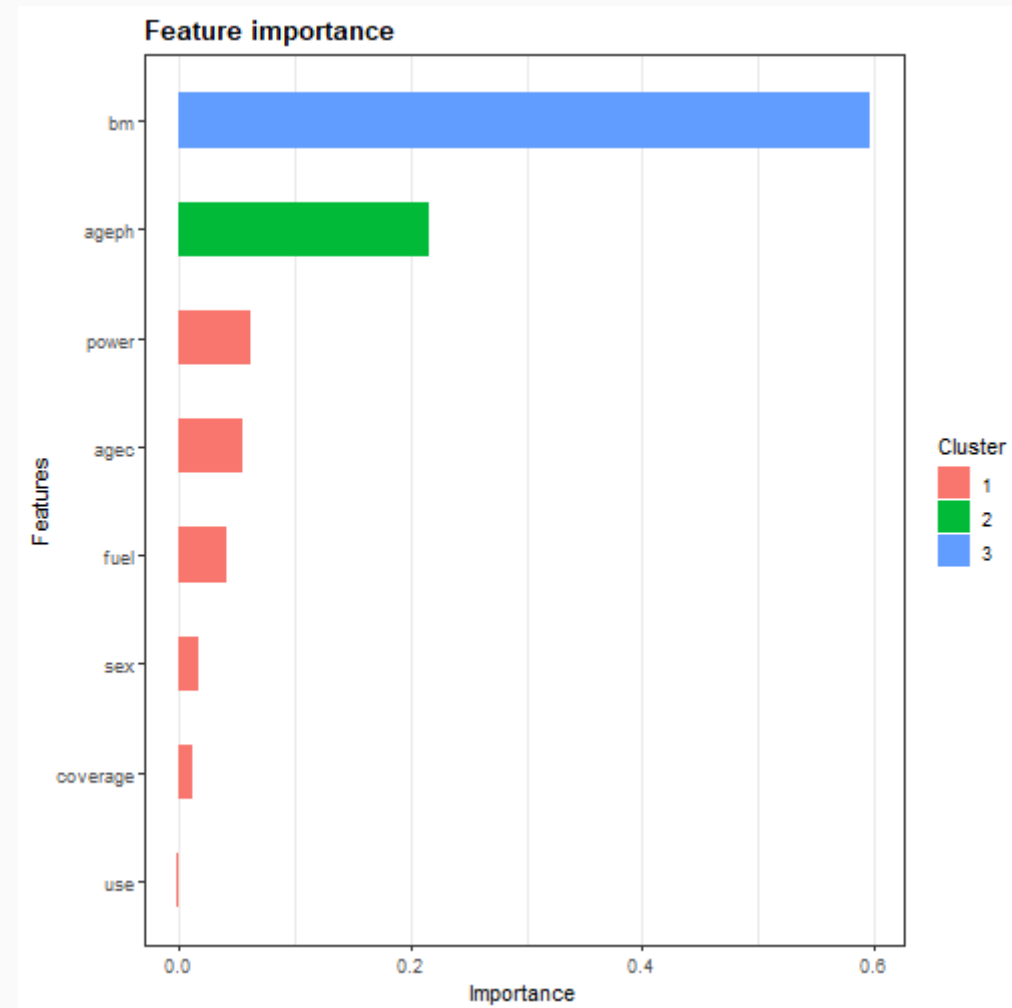
# Further built-in interpretations

- Built-in **feature importance**:

- `xgb.importance`: calculates **data**
- `xgb.ggplot.importance`: **visual** representation

```
xgb.ggplot.importance(  
  importance_matrix = xgb.importance(  
    feature_names = colnames(mtpl_xgb),  
    model = fit  
  )  
)
```

- Packages such as {vip} and {pdp} can also be used on `xgboost` models
  - even a **vignette** dedicated to this



# Cross-validation with XGBoost

## Built-in cross-validation with `xgb.cv`

- same interface as the `xgboost` function
- add `nfolds` to define the **number of folds**
- add `stratified` for **stratification**

```
set.seed(86493) # reproducibility
xval <- xgb.cv(data = mtlp_xgb,
               nrounds = 200,
               early_stopping_rounds = 20,
               verbose = FALSE,
               nfold = 5,
               stratified = TRUE,
               params = list(booster = 'gbtree',
                             objective = 'count:poisson',
                             eval_metric = 'poisson-nloglik',
                             eta = 0.1, nthread = 1,
                             subsample = 0.75, colsample_bynode = 0.5,
                             max_depth = 3, min_child_weight = 1000,
                             gamma = 0, lambda = 1, alpha = 1))
```

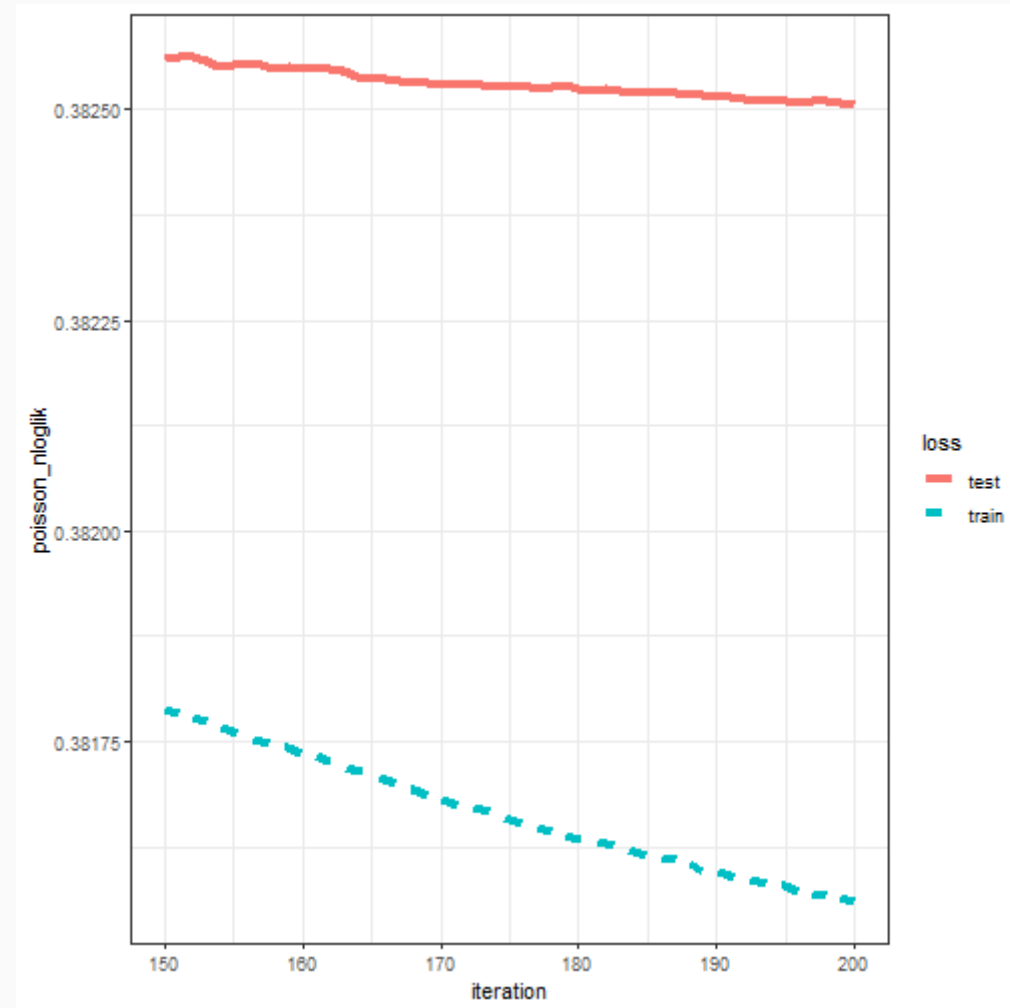
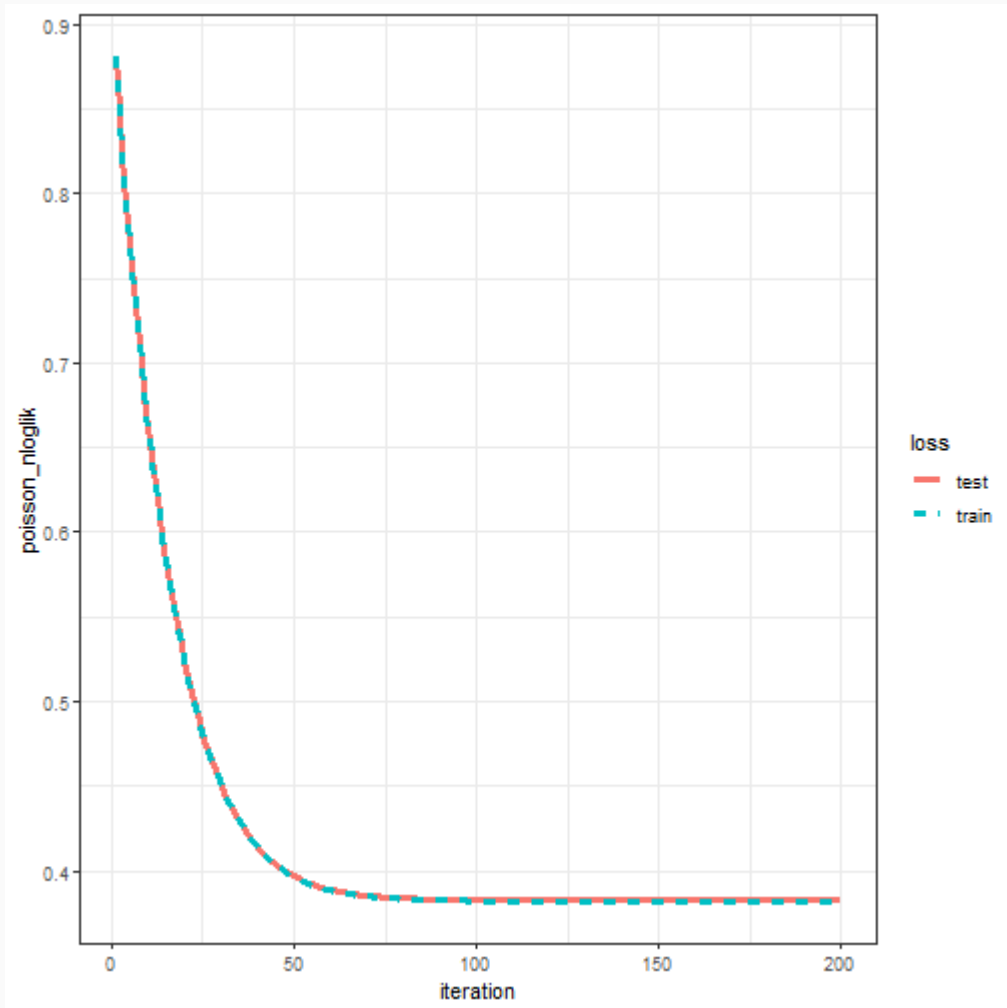
# Cross-validation results

Get the cross-validation **results** via `$evaluation_log`:

```
xval$evaluation_log %>% print(digits = 5)
##      iter train_poisson_nloglik_mean train_poisson_nloglik_std
##  1:      1          0.88048          0.00017649
##  2:      2          0.84983          0.00016304
##  3:      3          0.82167          0.00020066
##  4:      4          0.79477          0.00028473
##  5:      5          0.76935          0.00019933
## ---
## 196:    196          0.38157          0.00097562
## 197:    197          0.38157          0.00097635
## 198:    198          0.38157          0.00097577
## 199:    199          0.38156          0.00097688
## 200:    200          0.38156          0.00097555
##      test_poisson_nloglik_mean test_poisson_nloglik_std
##  1:          0.88052          0.00071933
##  2:          0.85029          0.00083193
##  3:          0.82171          0.00083297
##  4:          0.79471          0.00083522
##  5:          0.76925          0.00086406
## ---
## 196:          0.38251          0.00381020
```



# Cross-validation results





## Your turn

That's a wrap on **tree-based ML**! Now it's your time to experiment.

Below are some **suggestions**, but feel free to **get creative**.


1. Perform a **tuning** exercise for your favorite tree-based algorithm. Beware that tuning can take up a lot of time, so do not overdo this.
2. Apply your favorite algorithm on a classification problem, for example to predict the **occurrence** of a claim.
3. Use a **gamma** deviance to build a **severity** XGBoost model. The `mtp1` data contains the average claim amount in the feature `average`. Remember: if you want to develop a GBM with a gamma loss, you need the implementation available at Harry Southworth's [Github](#).
4. Develop a boosting or random forest model for the **Ames Housing** data and extract **insights** in the form of feature importance and partial dependence plots.
5. Compare the performance of a regression tree, random forest and boosting model. Which model performs **best**?

# Thanks!



Slides created with the R package `xaringan`.

Course material available via

 <https://github.com/katrienantonio/hands-on-machine-learning-R-module-2>