Machine learning in R - Day 2

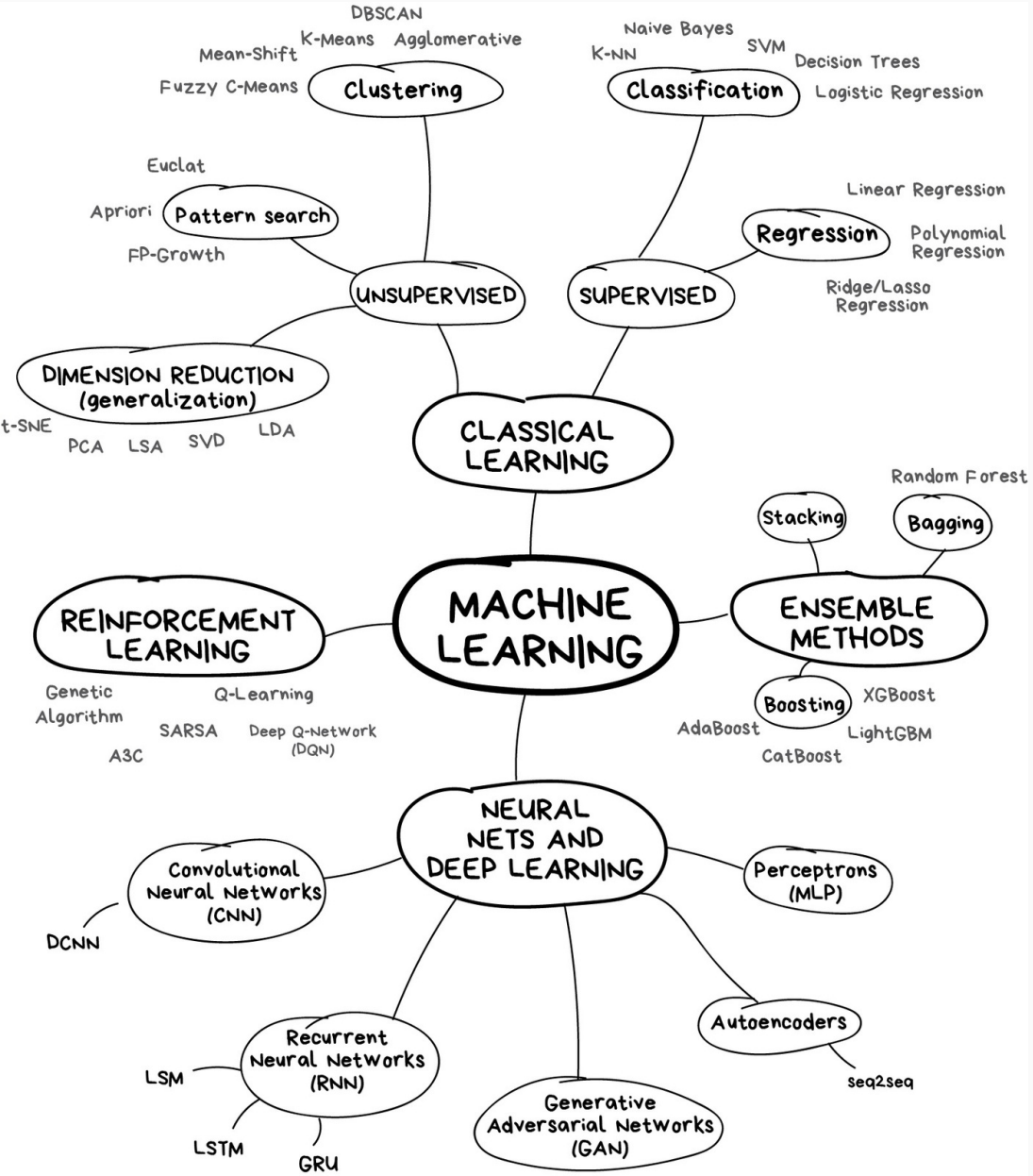
Hands-on workshop at Nationale Nederlanden

Katrien Antonio, Jonas Crevecoeur and Roel Henckaerts

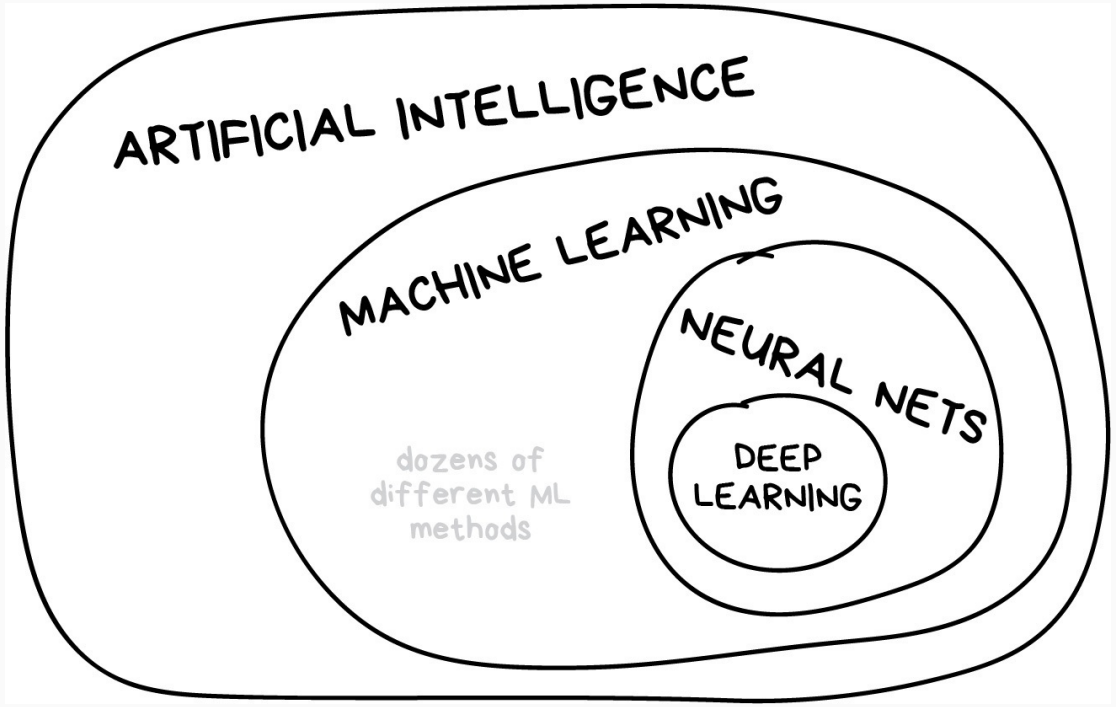
NN ML workshop | 2020-02-12

Today's outline

- Very short introduction
- Decision tree
  - Tree basics
  - Toy example for regression
  - Pruning via cross-validation
  - Toy example for classification
  - Claim frequency prediction
  - Interpretation tools
- Bagging and random forest
  - Bagging basics
  - Dominant features
  - Random forest
  - Tuning with a Cartesian grid search
  - Something for the actuaries
- Gradient boosting machine
  - Boosting basics
  - Important parameters
  - Claim frequency prediction
  - XGBoost
- H2O
  - H2O basics
  - Tuning with a random grid search
  - Stacking



Some roadmaps to explore the ML landscape...



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

Very short introduction

What is tree-based machine learning?

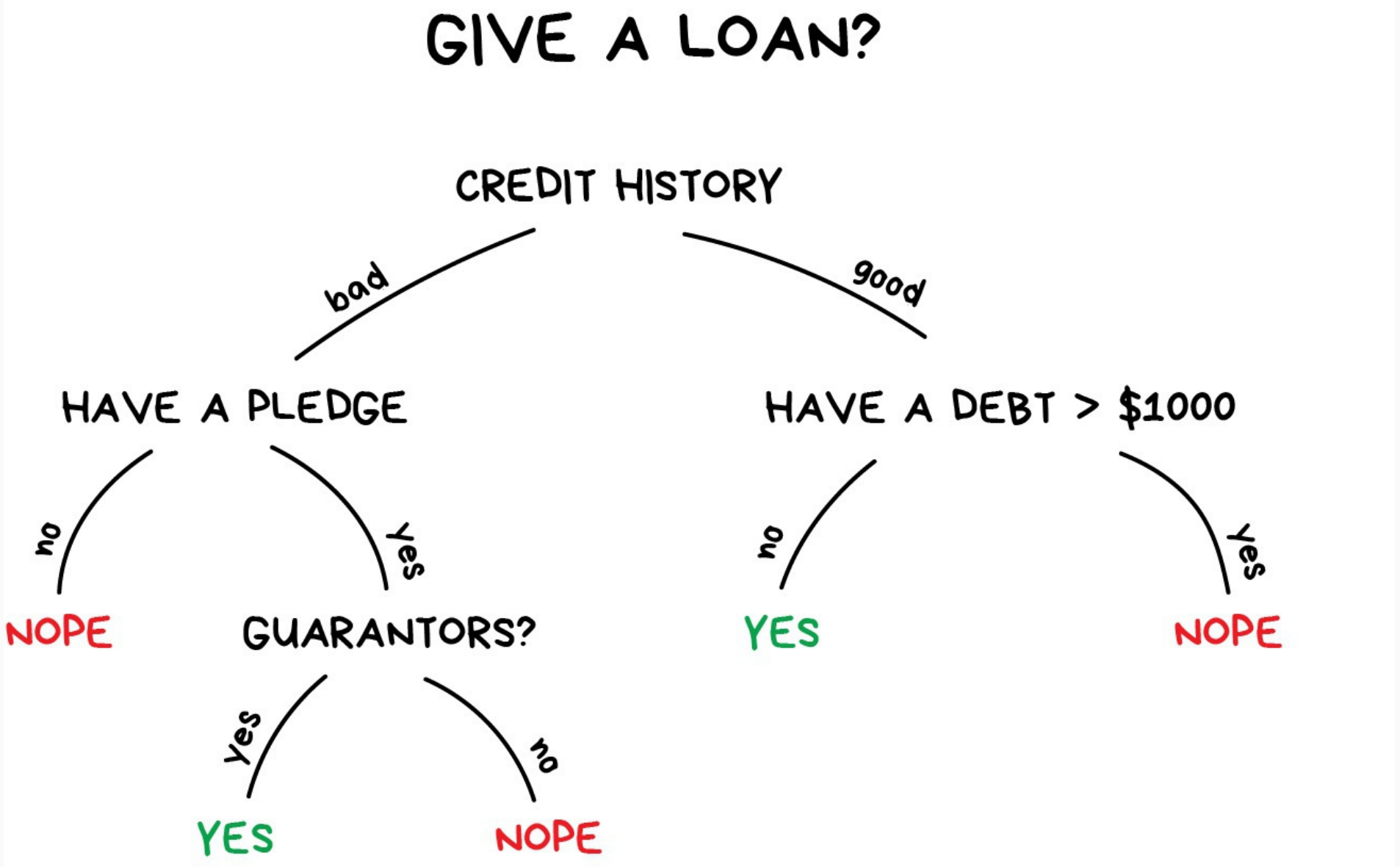
- Machine learning (ML) according to Wikipedia:

> *"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.

- What is a tree?

Decision tree



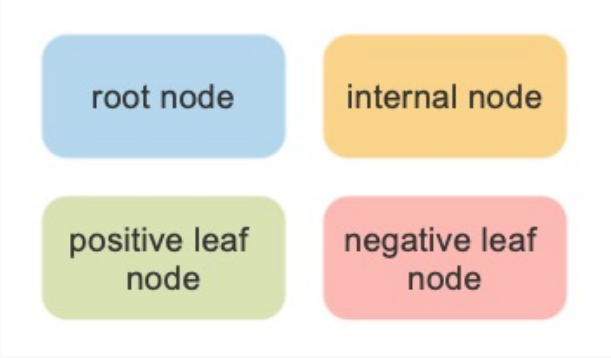
DECISION TREE

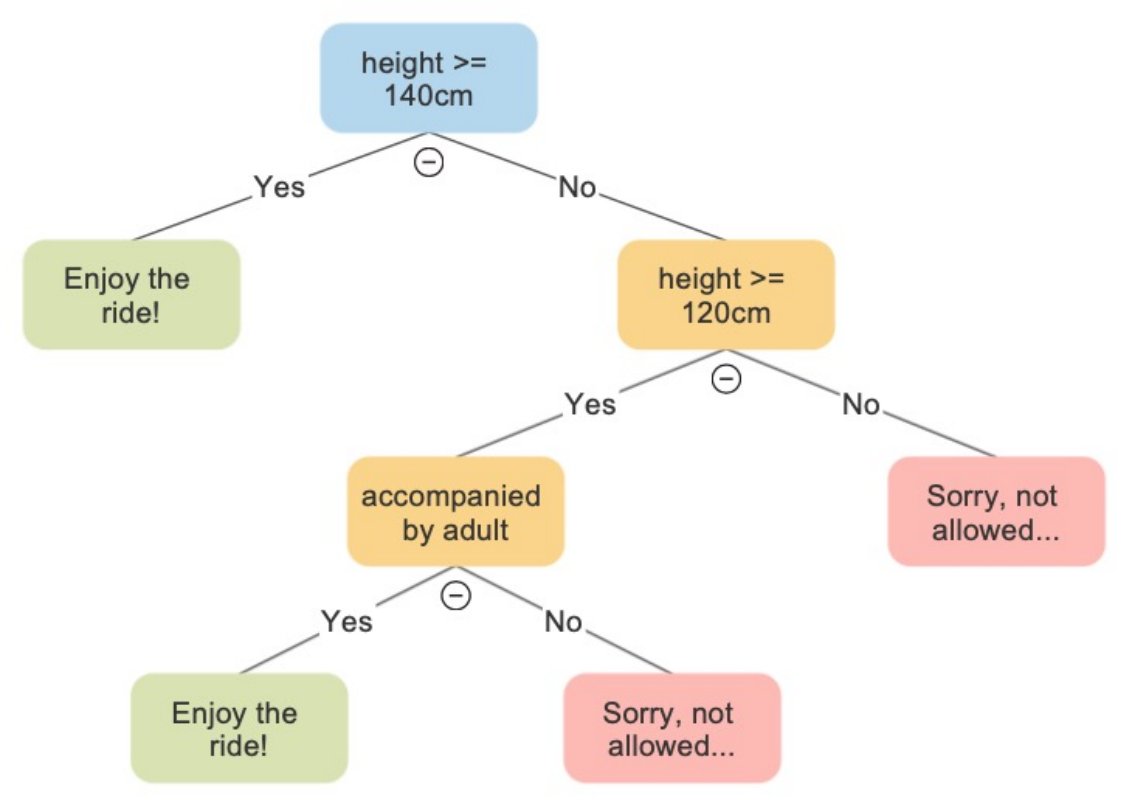
Tree structure and terminology

- Top of the tree contains all available training observations: **root node**
- Partition the data into homogeneous non-overlapping subgroups: **nodes**
- Subgroups formed via simple yes-no questions
- Tree predicts the output in a **leaf node** as follows:
  - average of the response for regression
  - majority voting for classification

Tree structure and terminology

- Top of the tree contains all available training observations: **root node**
- Partition the data into homogeneous non-overlapping subgroups: **nodes**
- Subgroups formed via simple yes-no questions
- Tree 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

- Golden standard is the **c**lassification **a**nd **r**egression **t**ree algorithm: CART
- CART uses binary recursive partitioning to split the data in subgroups
- At each node, search for the best feature to partition the data in two regions: R_1 and R_2 (hence, **binary**)
- Best? Minimize the overall loss between observed responses and leaf node prediction
  - overall loss = loss in R_1 + loss in R_2
  - regression: mean squared/absolute error, deviance,...
  - 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
- CART is implemented in the {rpart} package 

Using {rpart}

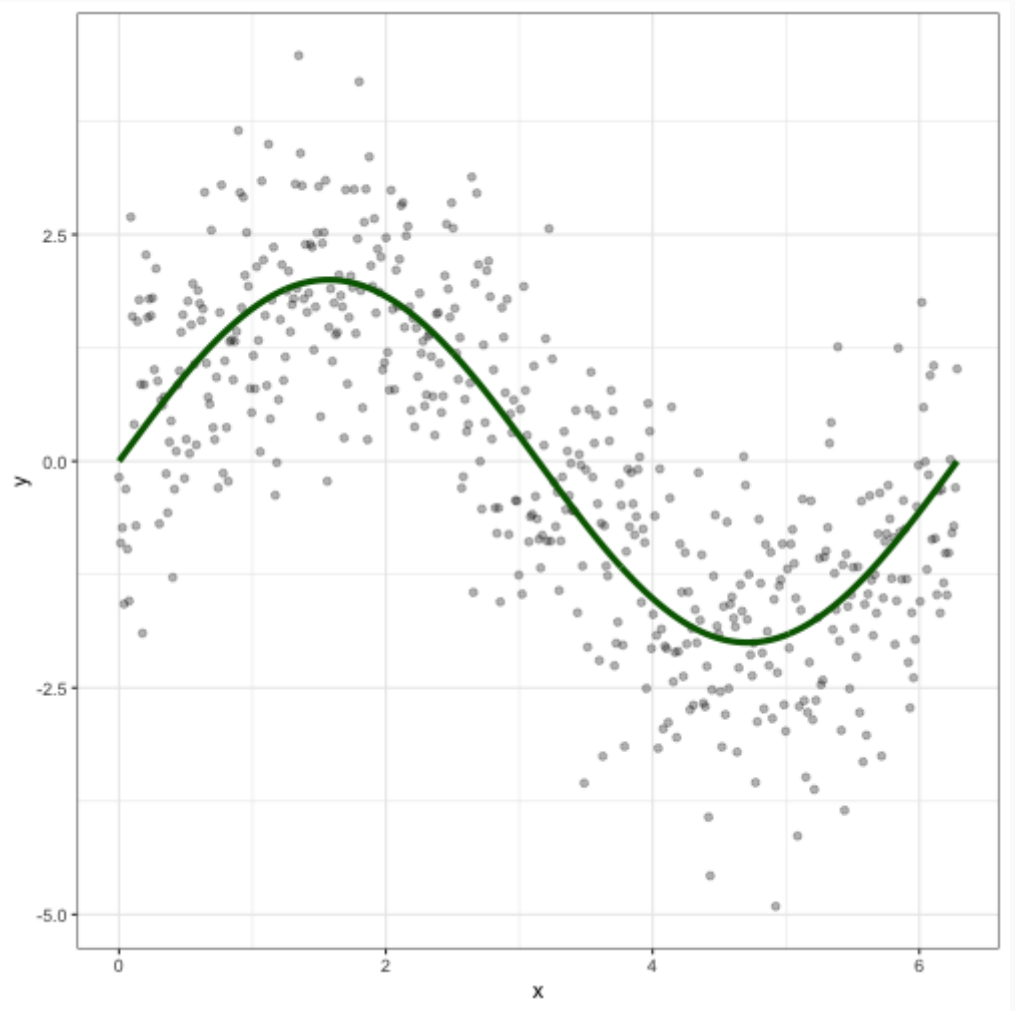
```
rpart(formula, data, method,
      control = rpart.control(cp, maxdepth, minsplit, minbucket))
```

- `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 for regression

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

```
## # A tibble: 500 x 3
##         x      m      y
##     <dbl>  <dbl>  <dbl>
##  1 0      0      -0.179
##  2 0.0126 0.0252 -0.903
##  3 0.0252 0.0504 -0.734
##  4 0.0378 0.0755 -1.58
##  5 0.0504 0.101  -0.307
##  6 0.0630 0.126  -0.970
##  7 0.0755 0.151  -1.54
##  8 0.0881 0.176   2.69
##  9 0.101  0.201   1.60
## 10 0.113  0.226   0.406
## # … with 490 more rows
```

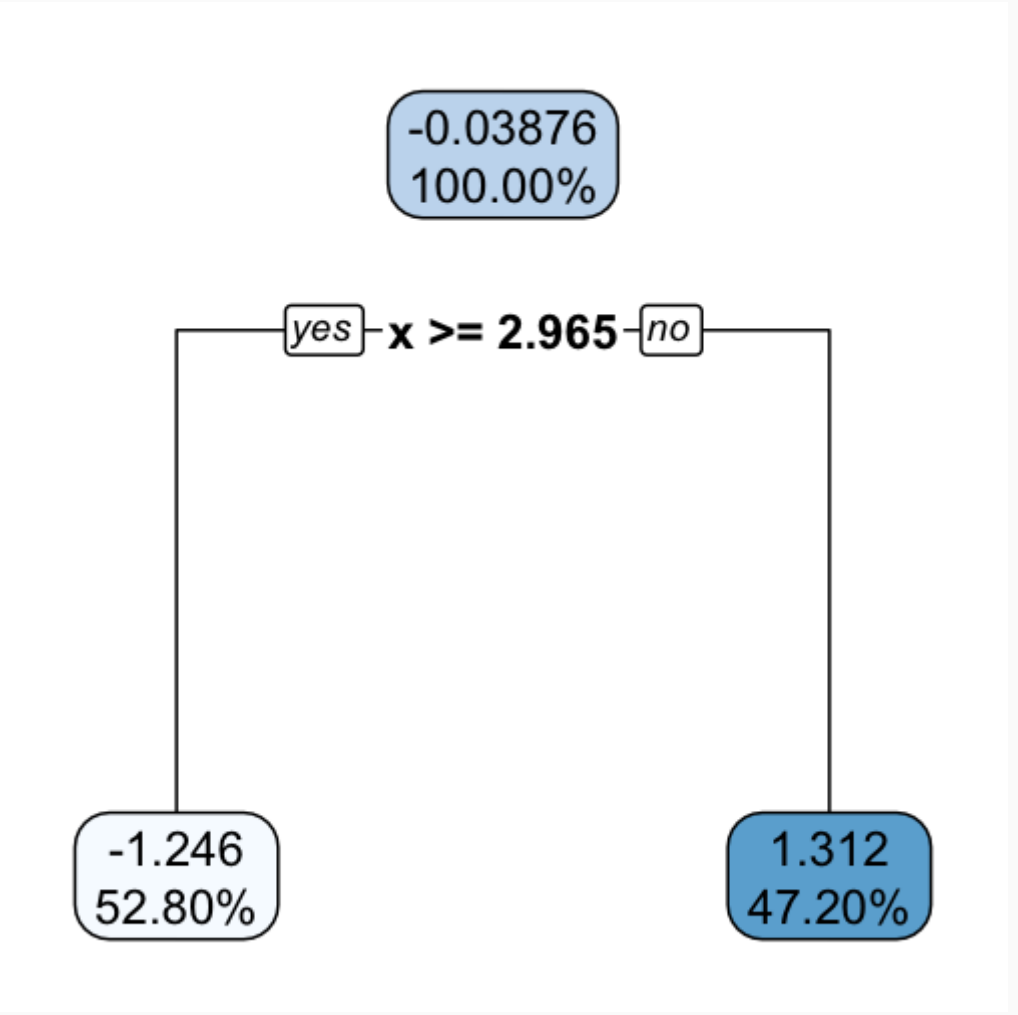


Decision stump (tree with only one split)

```
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 *
```

```
# Nice plots with the rpart.plot package
rpart.plot(fit, digits = 4, cex = 2)
```

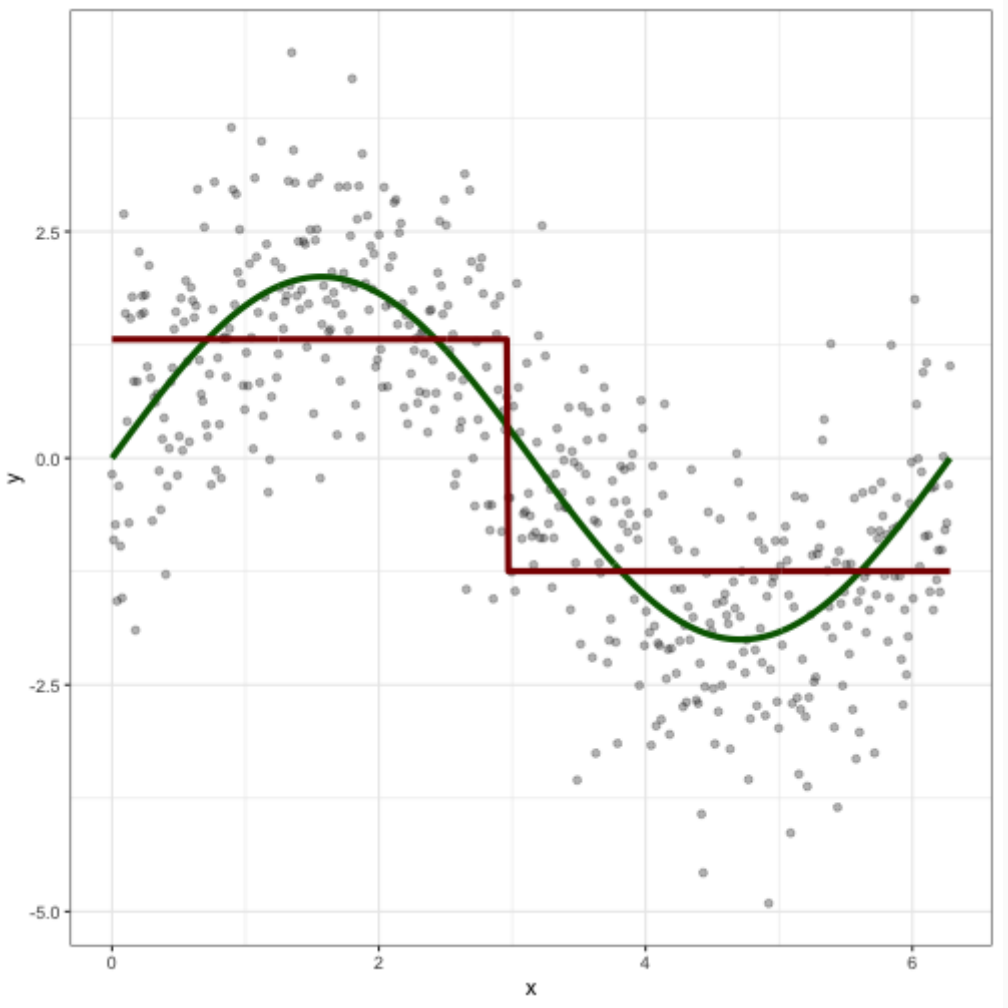


Decision stump (tree with only one split)

```
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 *
```

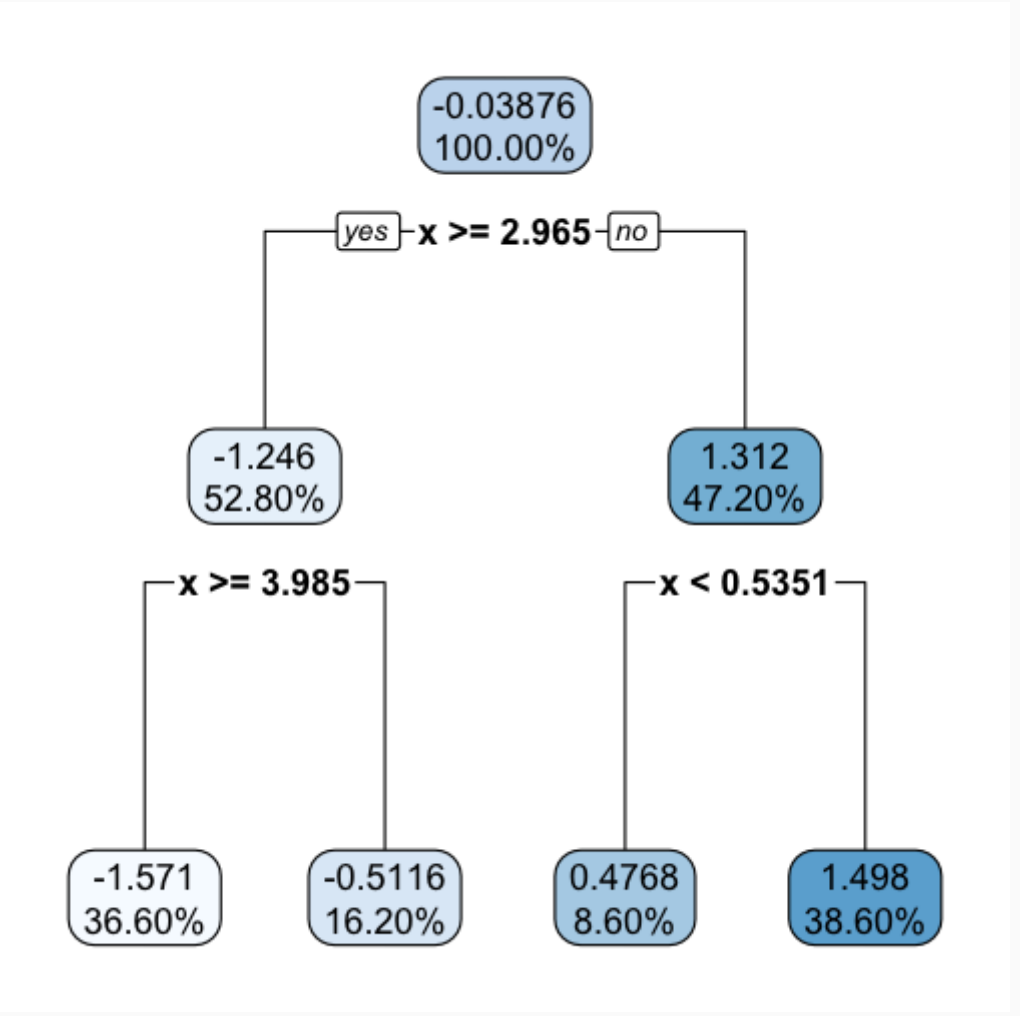
```
# Get predictions via the predict function
pred <- predict(fit, dfr)
```



Let's add some splits

```
fit <- rpart(formula = y ~ x,
             data = dfr,
             method = 'anova',
             control = rpart.control(
               maxdepth = 2
               )
             )
print(fit)
```

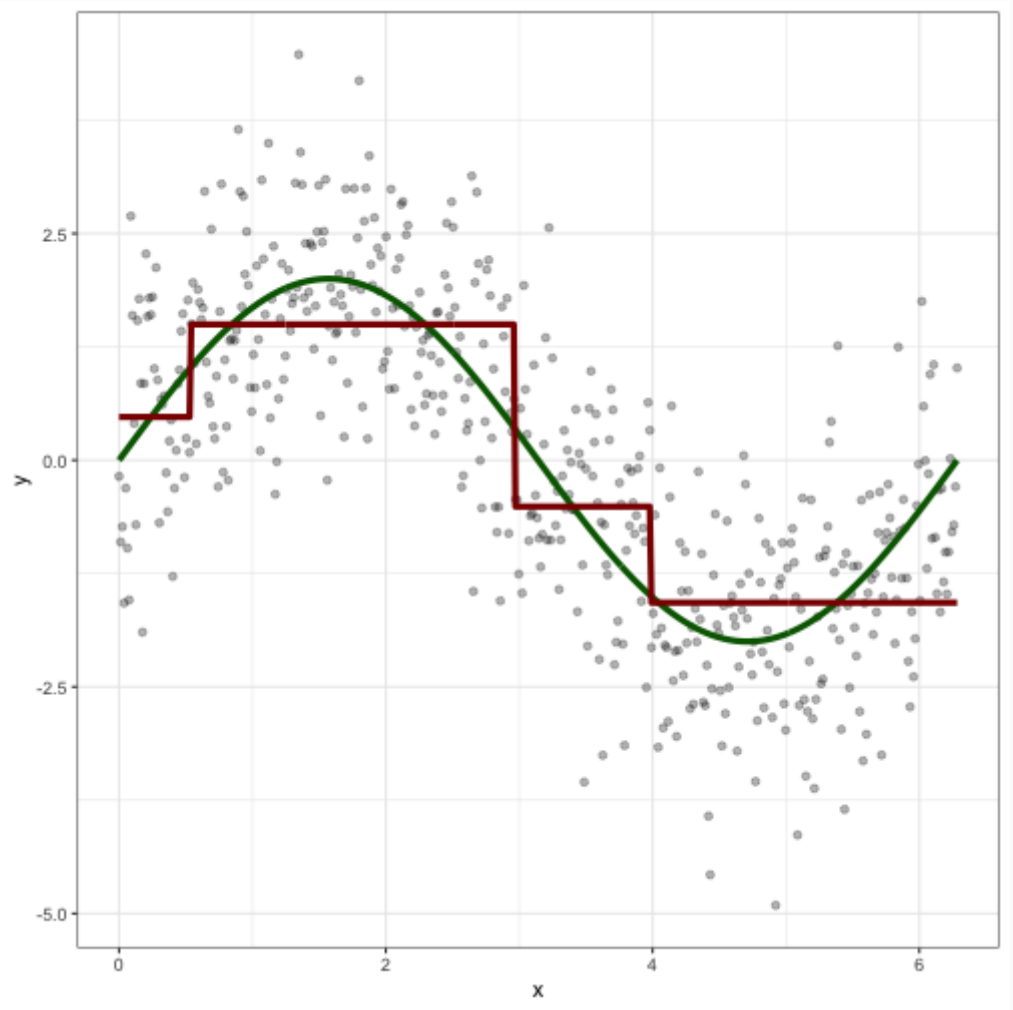
```
## n= 500
##
## node), split, n, deviance, yval
##       * denotes terminal node
##
## 1) root 500 1498.45700 -0.03876172
##   2) x>=2.965311 264  384.33360 -1.24604800
##     4) x>=3.985227 183  228.44490 -1.57111200 *
##     5) x< 3.985227 81   92.86428 -0.51164310 *
##   3) x< 2.965311 236  298.88880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



Let's add some 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 *
```



Your turn

Let's get familiar with the structure of a decision tree.
Choose your favorite tree and 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 two levels:

```
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

Q1: calculate the prediction

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

# Prediction
pred <- obs$y %>%  mean
pred
```

```
## [1] 0.4768002
```

Q2: calculate the deviance

```
# Deviance
dev <- (obs$y - pred)^2 %>% sum
dev
```

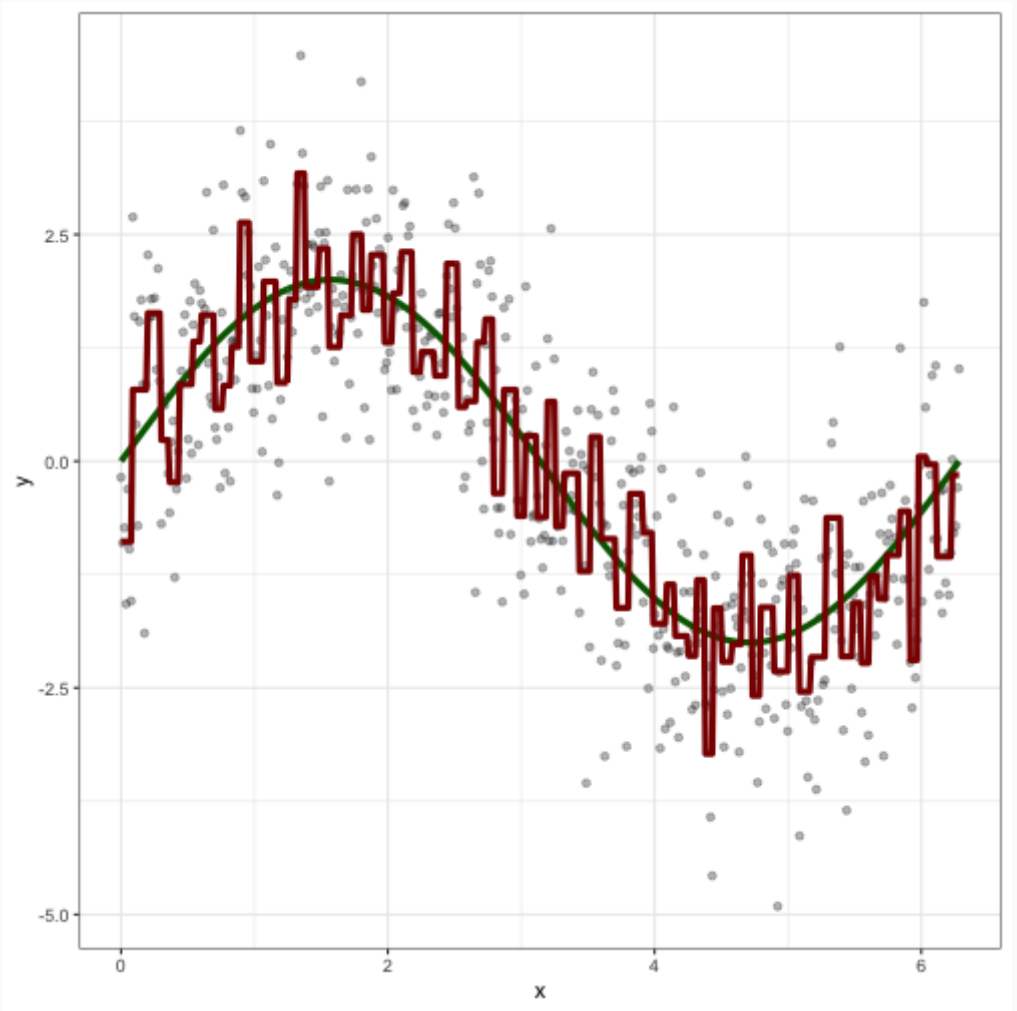
```
## [1] 55.23637
```

Let's build a very deep tree

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

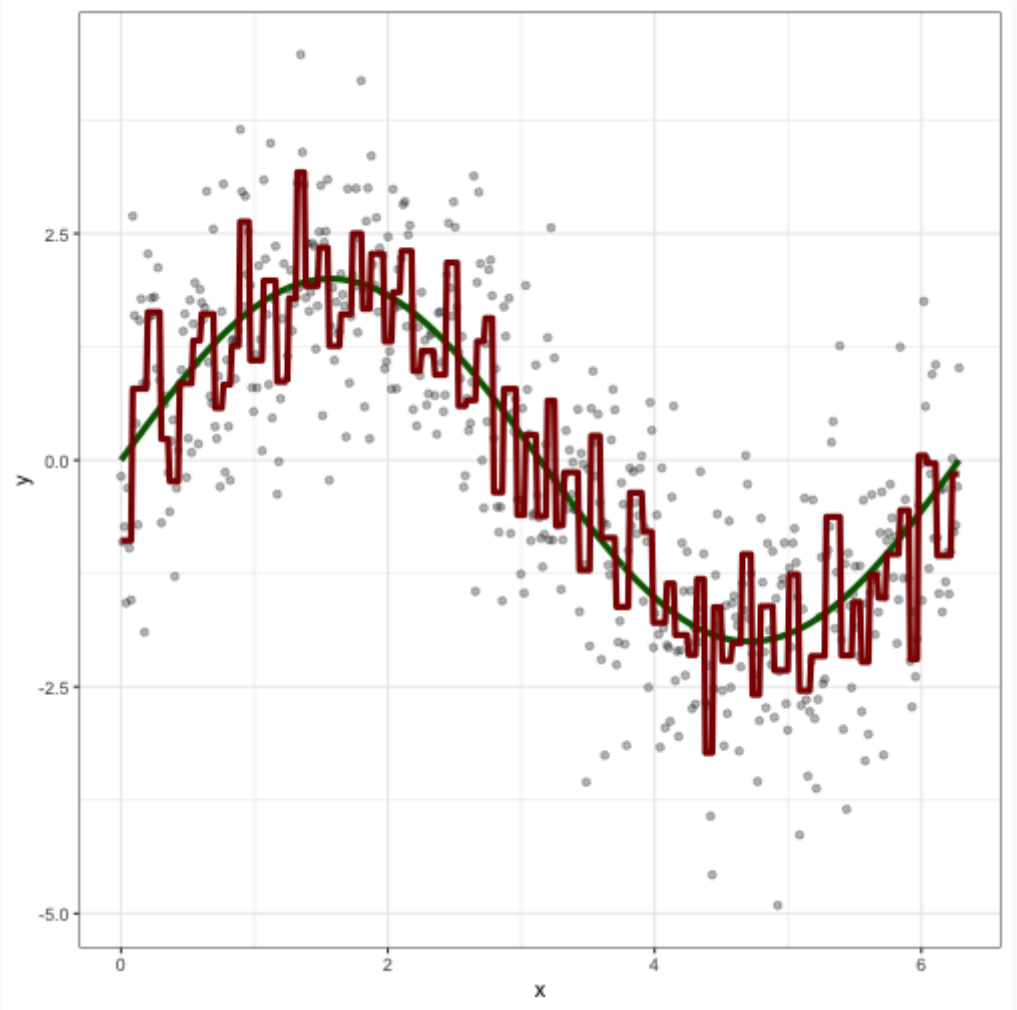
Note on the `cp` parameter:

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



Let's build a very deep tree

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



☹ Clearly dealing with **overfitting**

How deep should a tree be?

- Remember the bias-variance tradeoff:
  - **shallow** tree: bias ⬆ and variance ⬇

    ---> underfit
  - **deep** tree: bias ⬇ and variance ⬆

    ---> overfit
  - need to find the right balance ⚖

- Typical approach to get the right fit:
  1. fit an overly complex **deep tree**
  2. prune the tree to find the **optimal subtree**

How deep should a tree be?

- Remember the bias-variance tradeoff:
  - **shallow** tree: bias ⬆ and variance ⬇

    ---> underfit
  - **deep** tree: bias ⬇ and variance ⬆

    ---> overfit
  - need to find the right balance ⚖

- Typical approach to get the right fit:
  1. fit an overly complex **deep tree**
  2. prune the tree to find the **optimal subtree**

How to prune?

- Minimize a **penalized loss function** during training:

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

  - loss function f_{loss}
  - complexity parameter α
  - number of leaf nodes $|T|$
  - **shallow** tree when α ⬆
  - **deep** tree when α ⬇

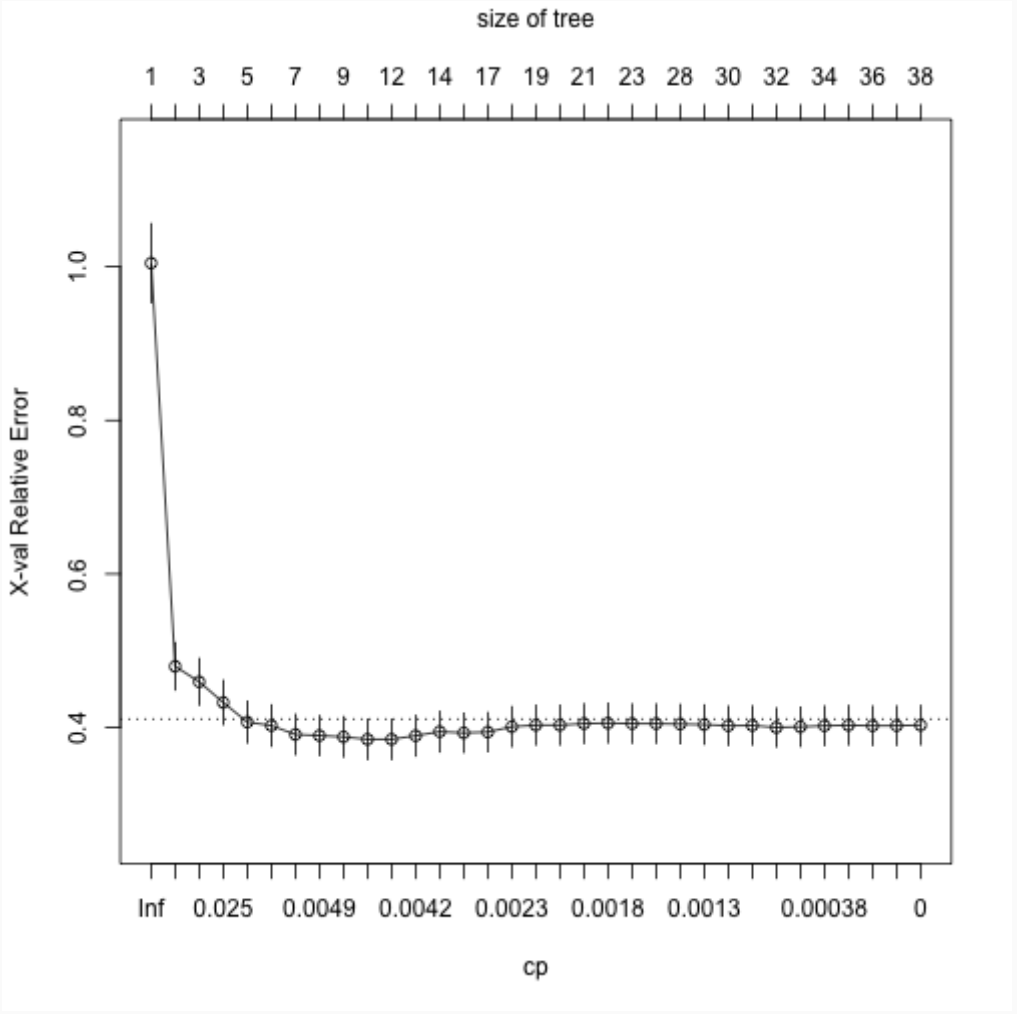
- Perform **cross-validation** on the parameter α
  - `cp` is the complexity parameter in {rpart}

- Same idea as the **Lasso** and **glmnet** on Day 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

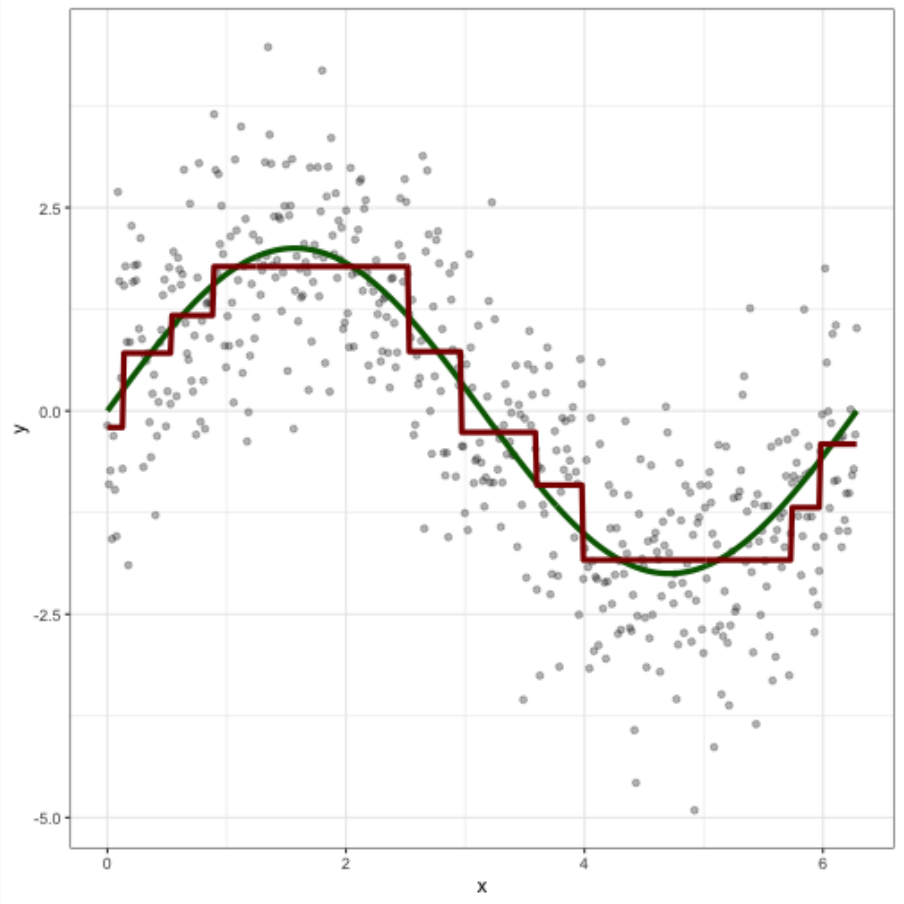
```
set.seed(87654) # reproducibility
fit <- rpart(formula = y ~ x,
             data = dfr,
             method = 'anova',
             control = rpart.control(
               maxdepth = 10,
               minsplit = 20,
               minbucket = 10,
               cp = 0,
               xval = 5
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

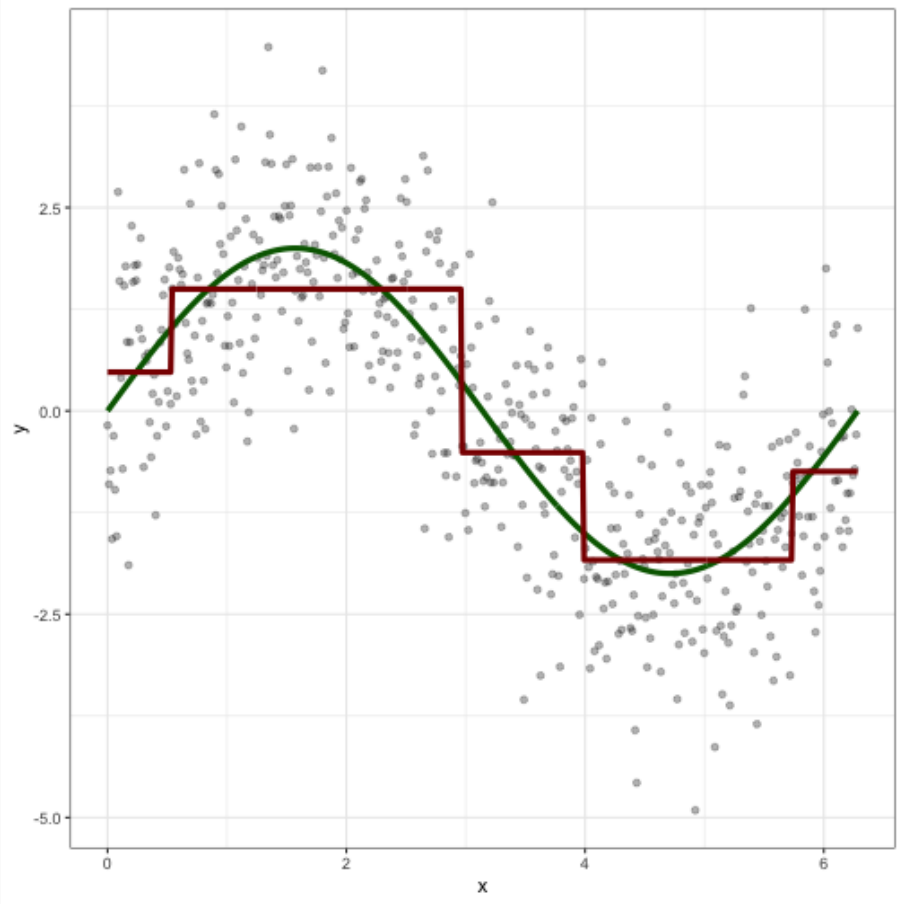
```
##            CP nsplit rel error   xerror      xstd
## 1  0.54404922      0  1.000000 1.004726 0.0514072
## 2  0.04205955      1  0.455951 0.479691 0.0306899
## 3  0.02638545      2  0.413891 0.459565 0.0303987
## 4  0.02446313      3  0.387506 0.432619 0.0288631
## 5  0.01686947      4  0.363043 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

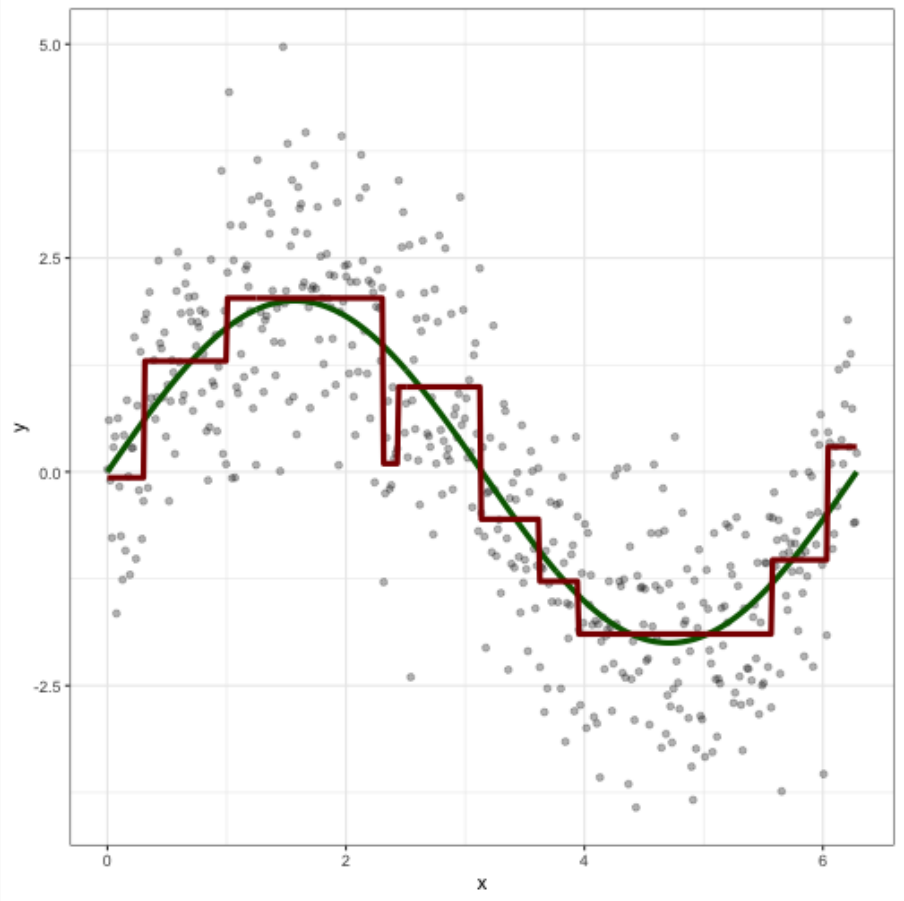
Trees are often associated with high variance, meaning that the resulting model can be very **sensitive** to the input data.

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 big differences with the trees from before?

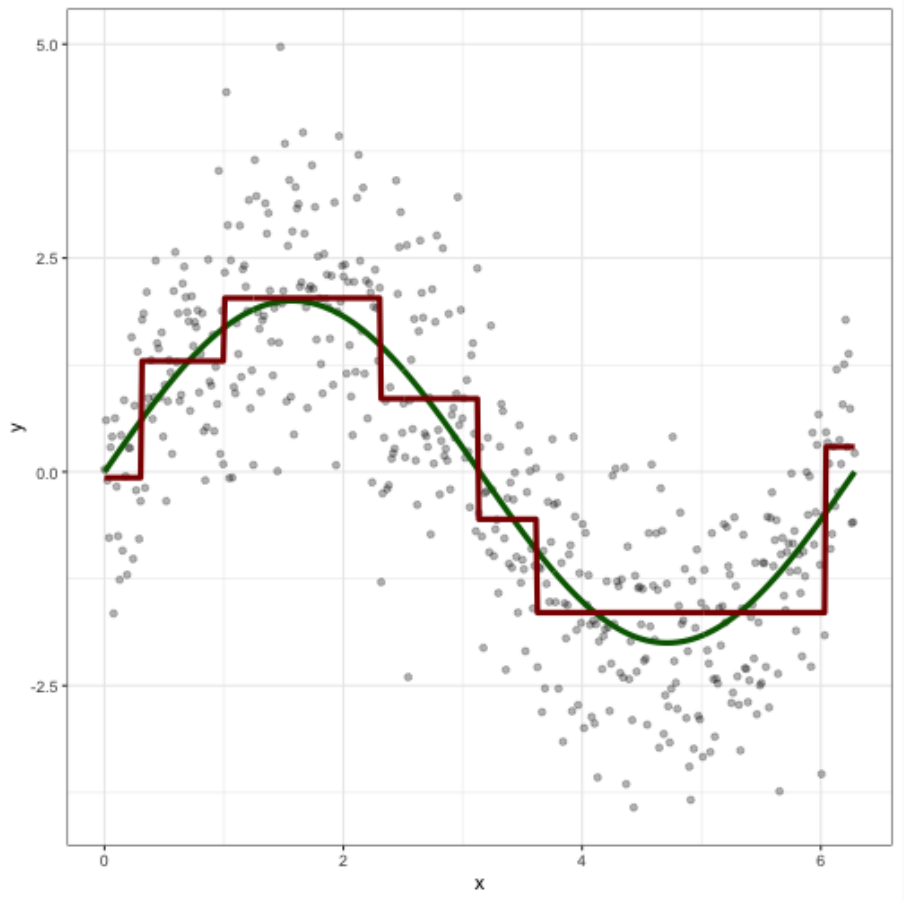
Q1: a brand new data set

```
# 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)
  )
```

Q2a: optimal tree via min CV error



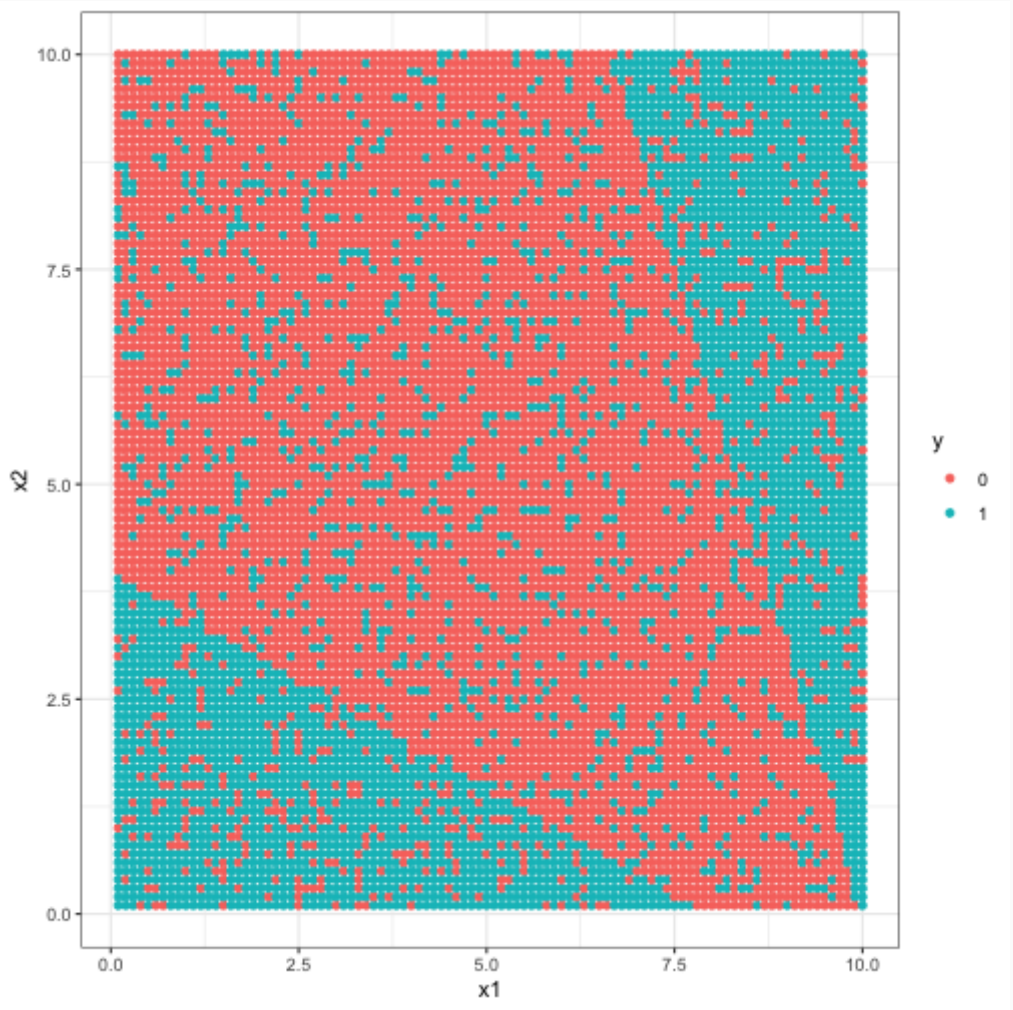
Q2b: optimal tree with via one SE rule



Q3: trees look rather different compared to those from before, even though they try to approximate the same function

Toy example for classification

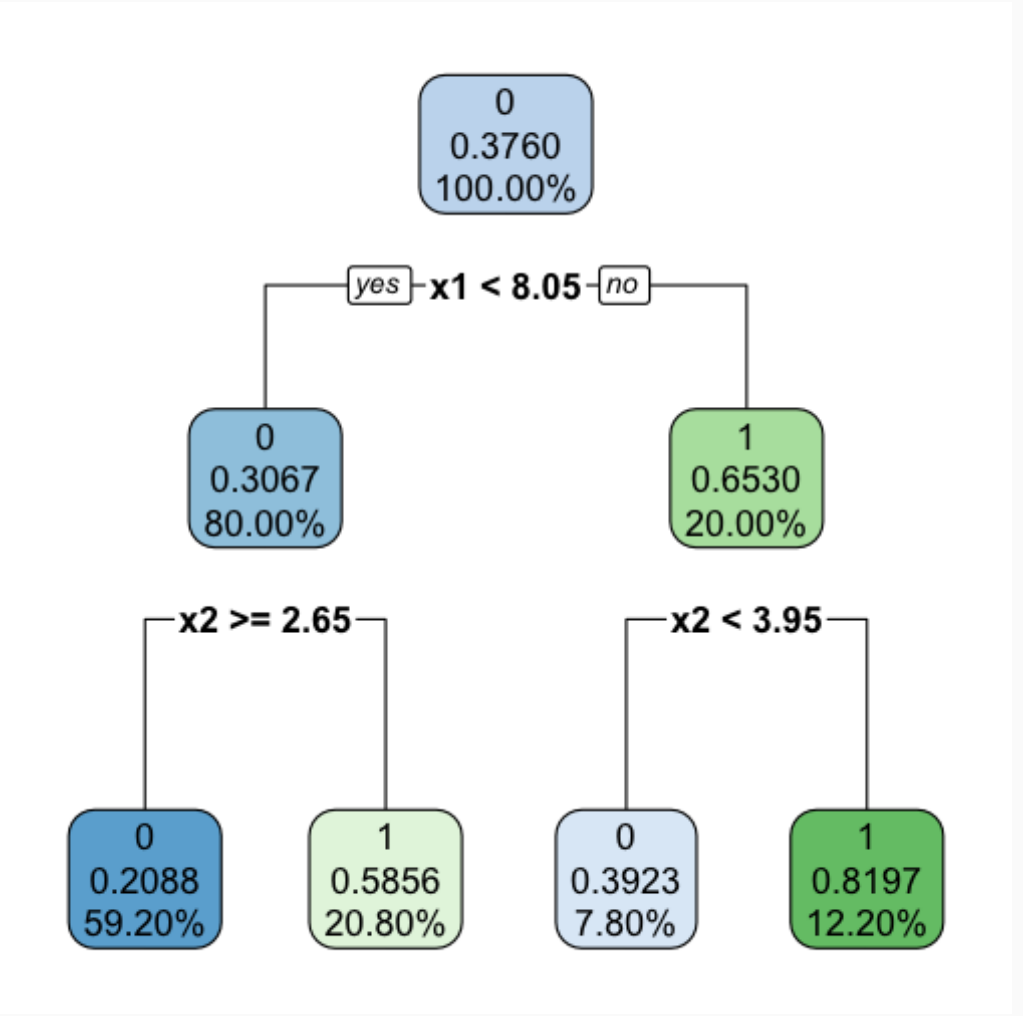
```
set.seed(54321) # reproducibility
dfc <- tibble(
  x1 = rep(seq(0.1,10,by = 0.1), times = 100),
  x2 = rep(seq(0.1,10,by = 0.1), each = 100),
  y = as.factor(
    pmin(1,
         pmax(0,
              round(
      1*(x1+2*x2<8) + 1*(3*x1+x2>30) +
        rnorm(10000,sd = 0.5))
              )
 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



Let's see what a simple tree does

```
fit <- rpart(formula = y ~ x1 + x2,
             data = dfc,
             method = 'class',
             control = rpart.control(
               maxdepth = 2
               )
             )
print(fit)
```

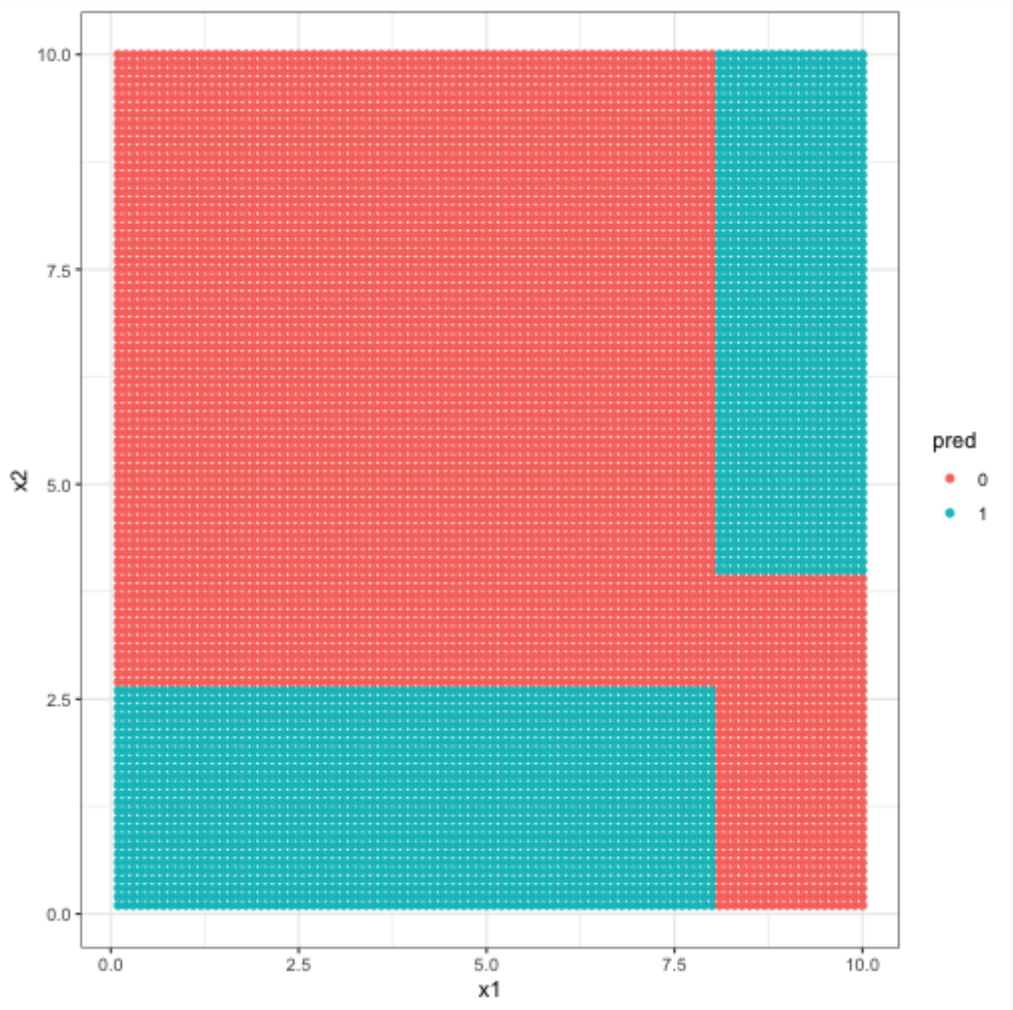
```
## n= 10000
##
## node), split, n, loss, yval, (yprob)
##       * denotes terminal node
##
## 1) root 10000 3760 0 (0.6240000 0.3760000)
##   2) x1< 8.05 8000 2454 0 (0.6932500 0.3067500)
##     4) x2>=2.65 5920 1236 0 (0.7912162 0.2087838) *
##     5) x2< 2.65 2080  862 1 (0.4144231 0.5855769) *
##   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) *
```



Let's see what a simple tree does

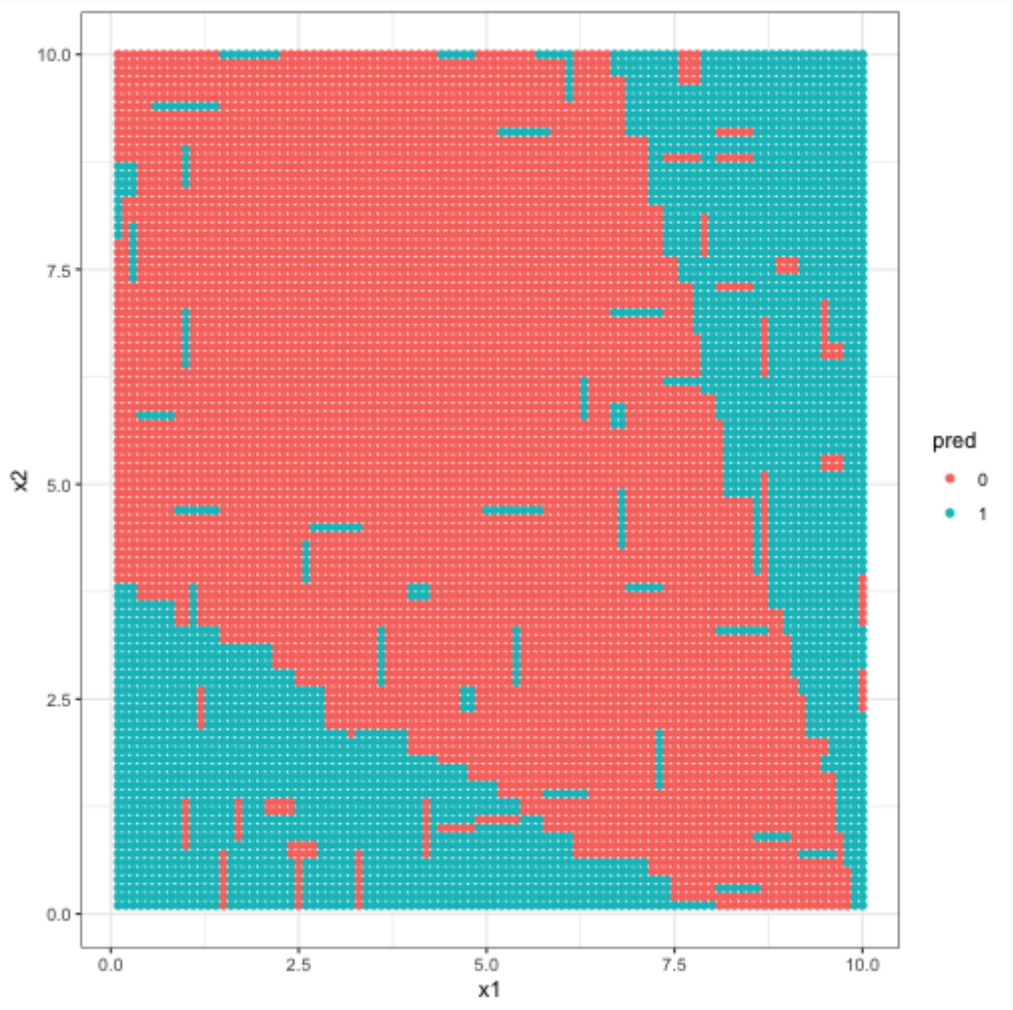
```
fit <- rpart(formula = y ~ x1 + x2,
             data = dfc,
             method = 'class',
             control = rpart.control(
               maxdepth = 2
               )
             )
print(fit)
```

```
## 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) *
```



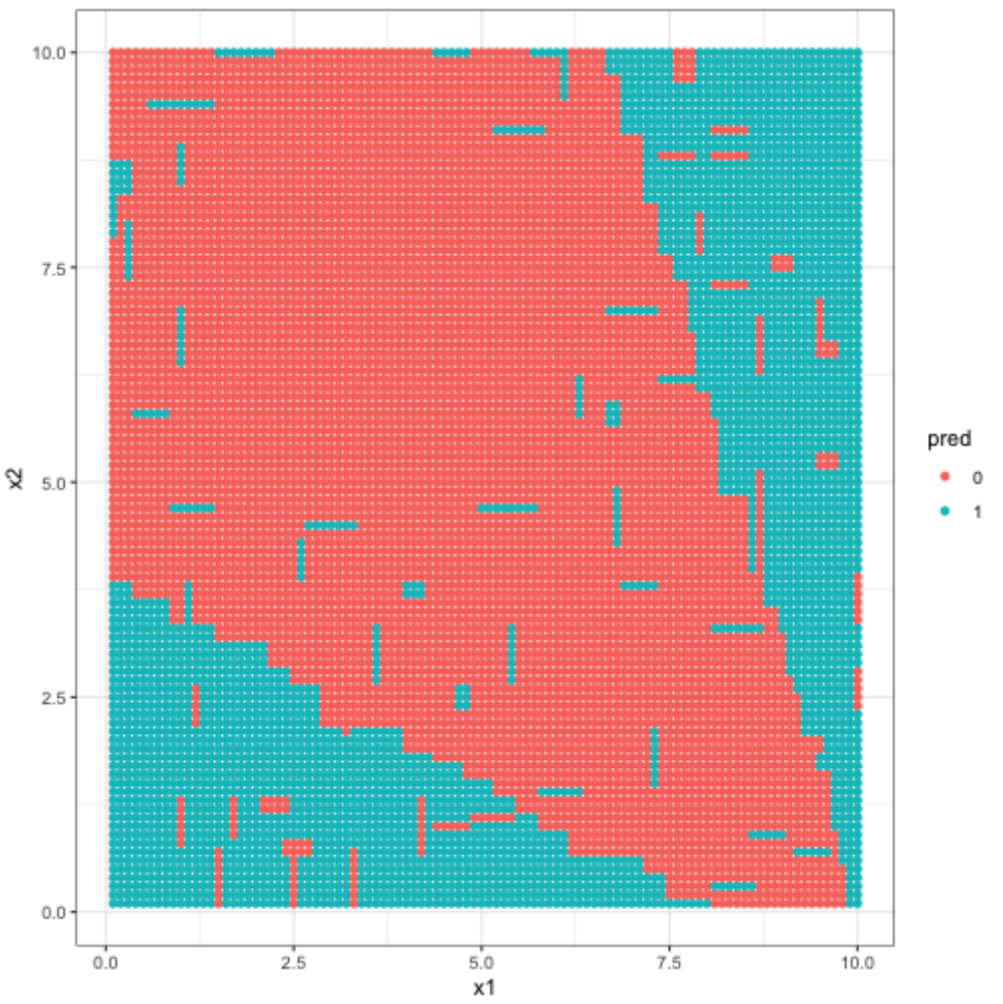
What about an overly complex tree?

```
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 dealing with **overfitting** again

Your turn

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

Perform **cross-validation** on `cp` to find the optimal pruned subtree.

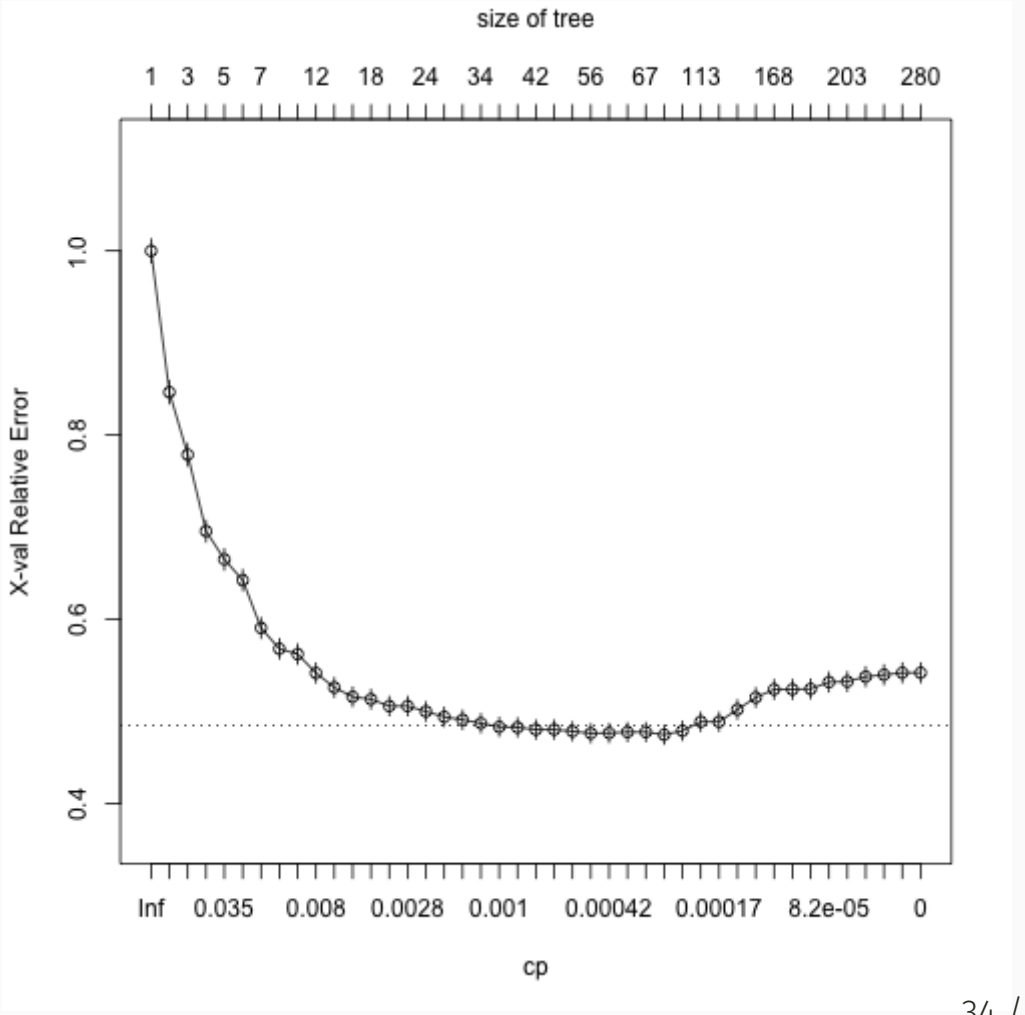
1. Set `xval = 5` in `rpart.control()` (do not forget to set a **seed** beforehand)
2. Graphically inspect the xval results via `plotcp()`
3. Extract the xval results via `$cptable`
4. Apply the min xerror and/or the one se rule to find the **optimal** `cp`
5. Show the resulting classification graphically

Q1: 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
               )
             )
```

Q2: inspect the xval results graphically

```
plotcp(fit)
```



Q3: extract the xval results in a table

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

Q4: 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'])))
```

```
unname(min_xerr)
```

```
## [1] 29
```

```
se_rule
```

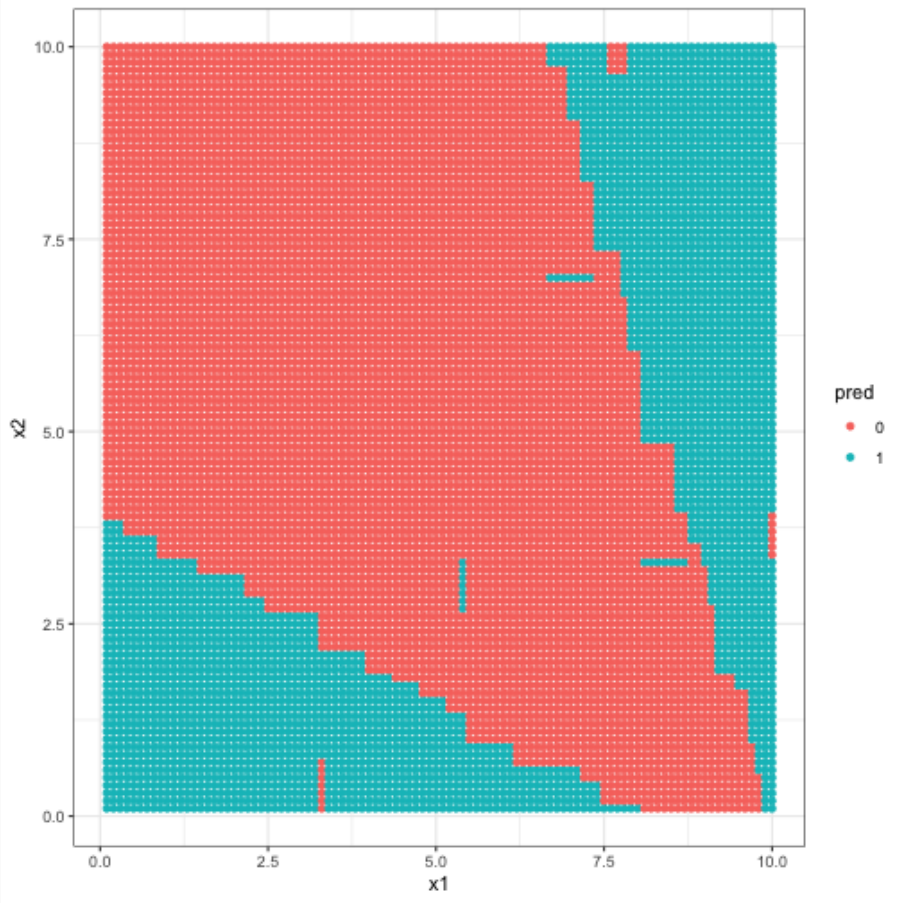
```
## [1] 20
```

```
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
```

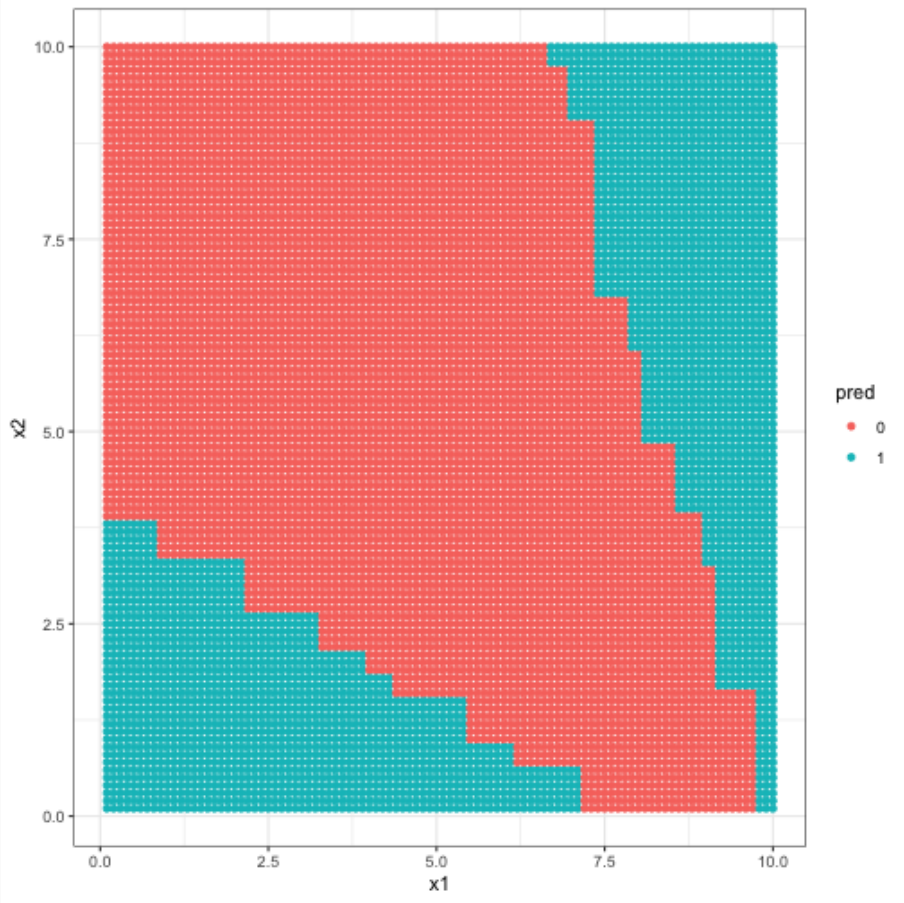
Q5a: optimal subtree via min cv error

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



Q5b: optimal subtree via one se rule

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



Claim frequency prediction with the MTPL data

- Classic approach for **claim frequency**: Poisson GLM
- How to deal with claim counts in a decision tree?
- Use the Poisson deviance as **loss function**:

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

- with expo the exposure measure.

Claim frequency prediction with the MTPL data

- Classic approach for **claim frequency**: Poisson GLM
- How to deal with claim counts in a decision tree?
- Use the Poisson deviance as **loss function**:

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

- with expo the exposure measure.

```
# Read the MTPL data
mtpl <- readRDS(paste0(data_path,'MTPL.rds'))
str(mtpl)
```

```
## 'data.frame':    163212 obs. of  18 variables:
##  $ id       : int  1 ...
##  $ expo     : num  1 ...
##  $ claim    : Factor w/ 2 levels "0","1": 2 ...
##  $ nclaims  : int  1 ...
##  $ amount   : num  1618 ...
##  $ average  : num  1618 ...
##  $ coverage: Factor w/ 3 levels "TPL","TPL+","TPL++"
##  $ ageph    : int  50 ...
##  $ sex      : Factor w/ 2 levels "female","male": 2 .
##  $ bm       : int  5 ...
##  $ power    : int  77 ...
##  $ agec     : int  12 ...
##  $ fuel     : Factor w/ 2 levels "gasoline","diesel":
##  $ use      : Factor w/ 2 levels "private","work": 1
##  $ fleet    : Factor w/ 2 levels "0","1": 1 ...
##  $ postcode: int  1000 ...
##  $ long     : num  4.36 ...
##  $ lat      : num  50.8 ...
```

Splitting the data into a train and test set

A test set is needed for **unbiased model comparison**

The {caret} package has some convenient functions for this:

```
set.seed(54321) # reproducubility

# Create a stratified data partition
train_id <- caret::createDataPartition(
  y = mtpl$nclaims/mtpl$expo,
  p = 0.8,
  groups = 100
  )[[1]]

# Divide the data in training and test set
mtpl_trn <- mtpl[train_id,]
mtpl_tst <- mtpl[-train_id,]
```

We can assess whether **stratification** went as planned:

```
# Proportions of the number of claims in train data
mtpl_trn$nclaims %>% table %>% prop.table %>% round(5)
```

```
## .
##       0       1       2       3       4       5
## 0.88822 0.10119 0.00948 0.00100 0.00010 0.00001
```

```
# Proportions of the number of claims in test data
mtpl_tst$nclaims %>% table %>% prop.table %>% round(5)
```

```
## .
##       0       1       2       3       4       5
## 0.88723 0.10193 0.00974 0.00095 0.00012 0.00003
```

 Proportions in train and test set are **well balanced**

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_trn,
             method = 'poisson',
             control = rpart.control(
               maxdepth = 3,
               cp = 0)
             )
```

```
print(fit)
```

For a Poisson tree in {rpart} you must specify:

Poisson deviance via `method = 'poisson'`

Response as two-column matrix: `cbind(expo,y)`

```
## n= 130571
##
## node), split, n, deviance, yval
##       * denotes terminal node
##
##  1) root 130571 71840.470 0.13890750
##    2) bm< 5.5 97827 48065.800 0.11535110
##      4) bm< 1.5 70827 32785.830 0.10451630
##        8) ageph>=49.5 37158 16129.480 0.09246052 *
##        9) ageph< 49.5 33669 16554.080 0.11832830 *
##      5) bm>=1.5 27000 15050.360 0.14428880
##       10) ageph>=53.5 7555  3878.027 0.12154010 *
##       11) ageph< 53.5 19445 11136.410 0.15341190 *
##    3) bm>=5.5 32744 22433.410 0.21404850
##      6) bm< 9.5 17082 10754.950 0.18397000
##       12) power< 38.5 2309  1210.009 0.13596560 *
##       13) power>=38.5 14773  9513.872 0.19134450 *
##      7) bm>=9.5 15662 11543.710 0.24871680
##       14) power< 39.5 2978  1905.891 0.19503990 *
##       15) power>=39.5 12684  9601.332 0.26108590 *
```

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_trn,
             method = 'poisson',
             control = rpart.control(
               maxdepth = 3,
               cp = 0)
             )
```

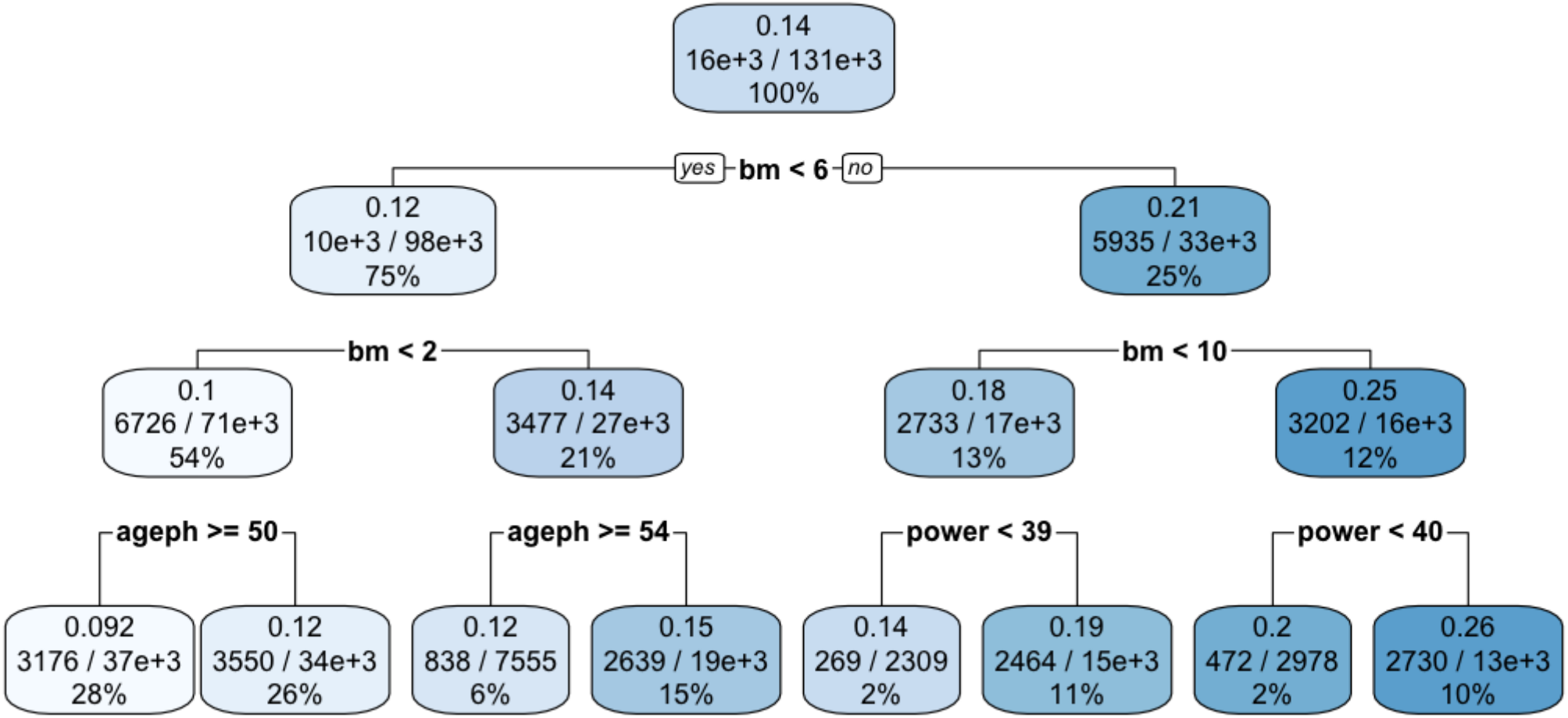
```
print(fit)
```

```
## n= 130571
##
## node), split, n, deviance, yval
##       * denotes terminal node
##
##  1) root 130571 71840.470 0.13890750
##    2) bm< 5.5 97827 48065.800 0.11535110
##      4) bm< 1.5 70827 32785.830 0.10451630
##        8) ageph>=49.5 37158 16129.480 0.09246052 *
##        9) ageph< 49.5 33669 16554.080 0.11832830 *
##      5) bm>=1.5 27000 15050.360 0.14428880
##       10) ageph>=53.5 7555  3878.027 0.12154010 *
##       11) ageph< 53.5 19445 11136.410 0.15341190 *
##    3) bm>=5.5 32744 22433.410 0.21404850
##      6) bm< 9.5 17082 10754.950 0.18397000
##       12) power< 38.5 2309  1210.009 0.13596560 *
##       13) power>=38.5 14773  9513.872 0.19134450 *
##      7) bm>=9.5 15662 11543.710 0.24871680
##       14) power< 39.5 2978  1905.891 0.19503990 *
##       15) power>=39.5 12684  9601.332 0.26108590 *
```

Easier way to **interpret** this tree?

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

Fitting a simple tree to the MTPL data



Your turn

Verify whether the prediction in a leaf node is **what you would expect**.
Take the rightmost node as an example: `bm >= 10` and `power >= 40`.

1. Subset the data accordingly
2. Calculate the expected claim frequency as `sum(nclaims)/sum(expo)`
3. Compare with the {rpart} prediction of 0.2610859

Q1-Q2: subset the data and calculate the claim frequency

```
mtpl_trn %>%
  dplyr::filter(bm >= 10,
                power >= 40) %>%
  dplyr::summarise(claim_freq =
                     sum(nclaims)/sum(expo))
```

```
##   claim_freq
## 1  0.2611701
```

Q3: The prediction and calculation don't match!

Is this due to a rounding error?

Or is there something spooky going on? 

Unraveling the mystery of {rpart}

- Section 8.2 in the vignette on Poisson regression
- **Conceptually**: no events in a leaf node lead to division by zero in the deviance
- Assume **Gamma prior** on the rates: $\text{Gamma}(\mu\,,\sigma)$
  - $\mu = \sum y_i / \sum \text{expo}_i$
  - **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}: $k = 1$
- **Leaf node prediction**:

$$\frac{\alpha + \sum Y_i}{\beta + \sum e_i}, \quad \alpha = 1/k^2, \quad \beta = \alpha/\mu$$

```
k <- 1

alpha <- 1/k^2

mu <- mtpl_trn %>%
  with(sum(nclaims)/sum(expo))

beta <- alpha/mu

mtpl_trn %>%
  dplyr::filter(bm >= 10, power >= 40) %>%
  dplyr::summarise(prediction =
            (alpha + sum(nclaims))/(beta + sum(expo)))
```

```
##   prediction
## 1  0.2610859
```

😊 **Mystery solved!**

Coefficient of variation very low

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

```
## n= 130571
##
## node), split, n, deviance, yval
##       * denotes terminal node
##
##  1) root 130571 71840.470 0.1389075
##    2) bm< 5.5 97827 48441.080 0.1389075
##      4) bm< 1.5 70827 33385.760 0.1389075 *
##      5) bm>=1.5 27000 15055.320 0.1389075
##       10) ageph>=53.5 7555  3893.701 0.1389075 *
##       11) ageph< 53.5 19445 11161.620 0.1389075 *
##    3) bm>=5.5 32744 23399.390 0.1389075 *
```

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_trn,
             method = 'poisson',
             control = rpart.control(
               maxdepth = 3,
               cp = 0),
             parms = list(shrink = 10^5)
             )
```

```
# Remember this number?
mtpl_trn %>%
  dplyr::filter(bm >= 10, power >= 40) %>%
  dplyr::summarise(claim_freq =
                     sum(nclaims)/sum(expo))
```

```
##   claim_freq
## 1  0.2611701
```

```
## n= 130571
##
## node), split, n, deviance, yval
##       * denotes terminal node
##
##  1) root 130571 71840.470 0.13890750
##    2) bm< 5.5 97827 48065.800 0.11534910
##      4) bm< 1.5 70827 32785.830 0.10451250
##        8) ageph>=49.5 37158 16129.480 0.09245078 *
##        9) ageph< 49.5 33669 16554.080 0.11832330 *
##      5) bm>=1.5 27000 15050.360 0.14429040
##       10) ageph>=53.5 7555  3878.027 0.12152200 *
##       11) ageph< 53.5 19445 11136.410 0.15341800 *
##    3) bm>=5.5 32744 22433.410 0.21406810
##      6) bm< 9.5 17082 10754.950 0.18399180
##       12) power< 38.5 2309  1210.009 0.13595490 *
##       13) power>=38.5 14773  9513.872 0.19137380 *
##      7) bm>=9.5 15662 11543.710 0.24877820
##       14) power< 39.5 2978  1905.891 0.19520710 *
##       15) power>=39.5 12684  9601.332 0.26117010 *
```

Your turn

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

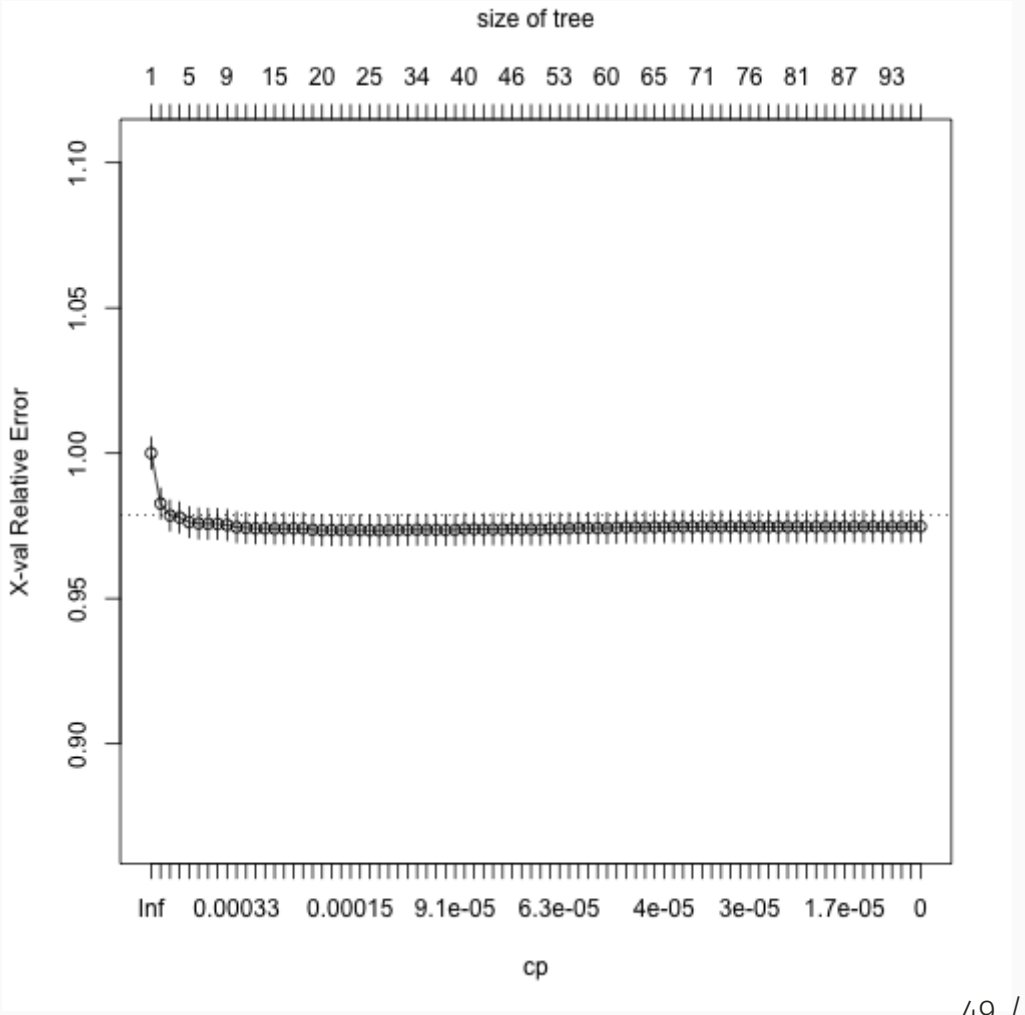
1. Start from an overly complex tree (don't forget your favorite random **seed** upfront)
2. Inspect the cross-validation results
3. Choose the `cp` value minimizing `xerror` for **pruning**
4. Visualize the pruned tree with `rpart.plot`

Q1: fit an overly complex tree

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

Q2: inspect the cross-validation results

```
plotcp(fit)
```



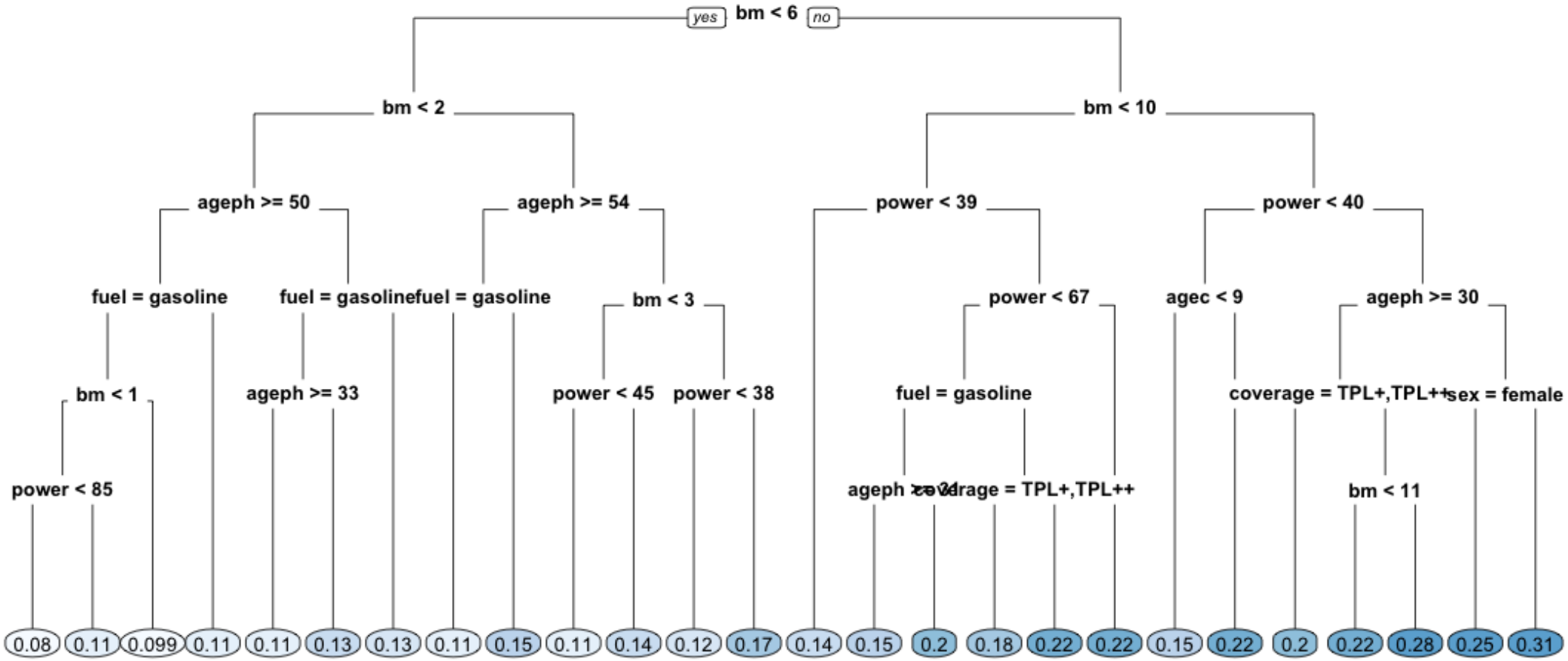
Q3: choose the `cp` value that minimizes `xerror` for pruning

```
# 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
## 1.235731e-04 2.500000e+01 9.693534e-01 9.734269e-01 5.243647e-03
```

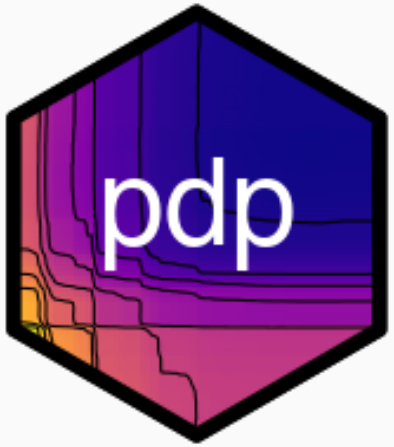
```
# Prune the tree
fit_srt <- prune(fit,
                 cp = cpt[min_xerr, 'CP'])
```

Q4: try to understand how the final model looks like. Can you make sense of it?



Making sense of a tree model

- Interpretability depends on the size of the tree
  - **shallow** tree 😊 but **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} 📦
- Good source on interpretable machine learning: ebook

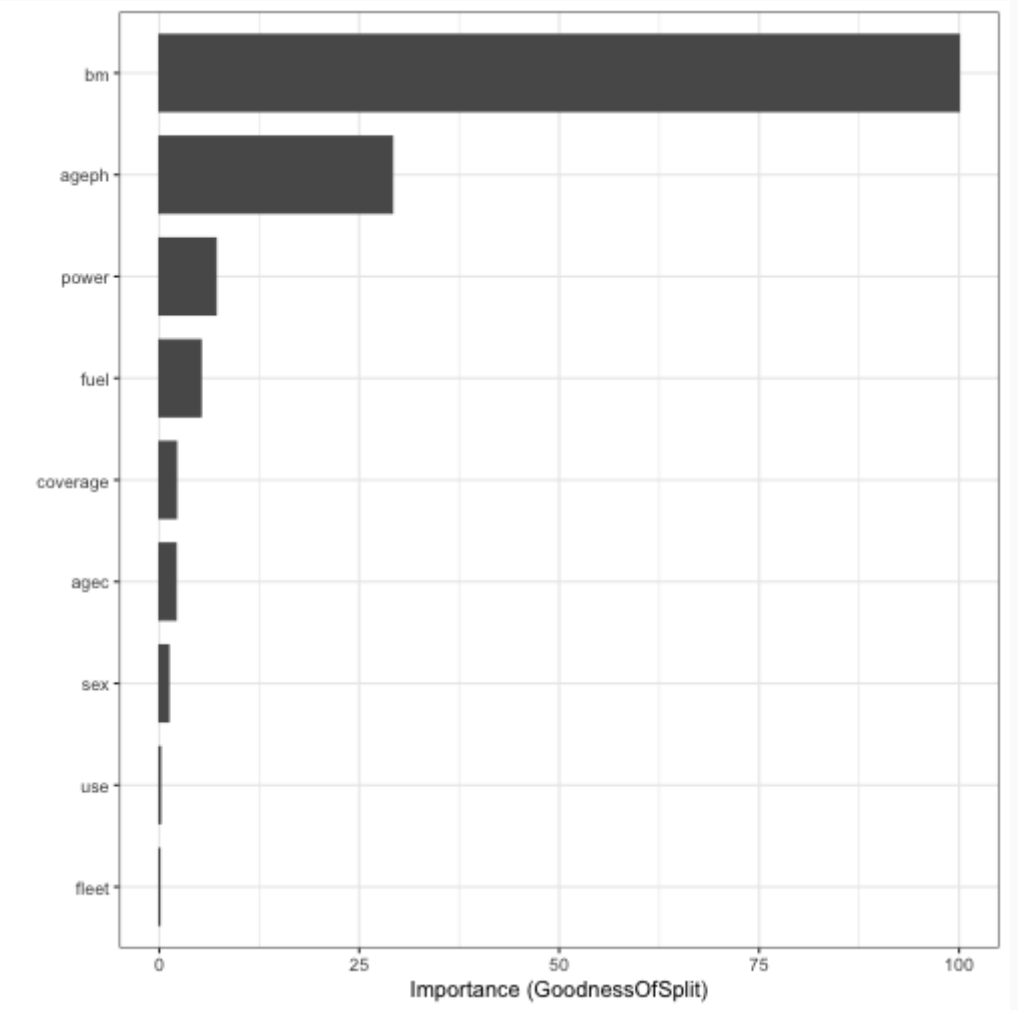


Feature importance

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

```
## # A tibble: 9 x 2
##   Variable Importance
##   <chr>          <dbl>
## 1 bm          1757.
## 2 ageph        512.
## 3 power        124.
## 4 fuel          90.9
## 5 coverage      37.0
## 6 agec          36.1
## 7 sex           20.0
## 8 use            1.95
## 9 fleet          0.446
```

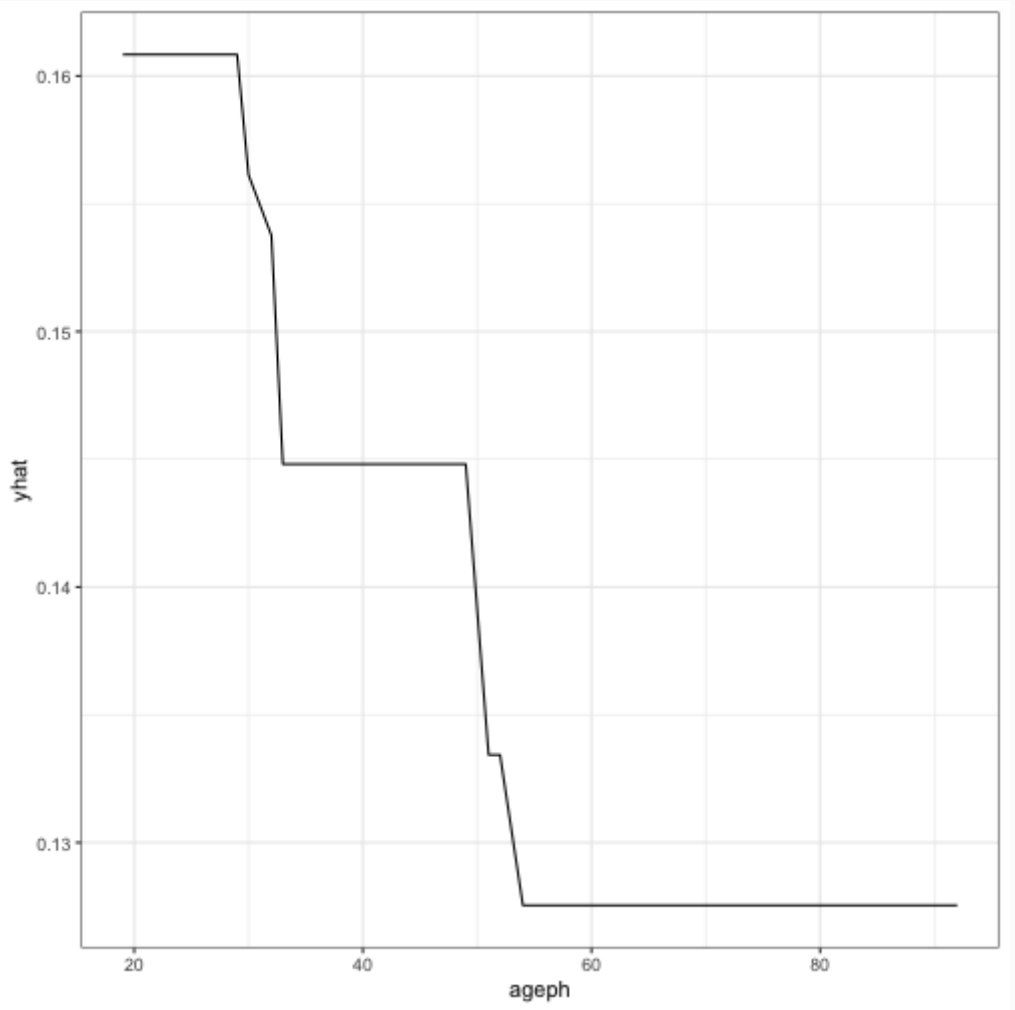
```
# Function vip makes the plot
vip::vip(fit_srt, scale = TRUE)
```



Partial dependence plot

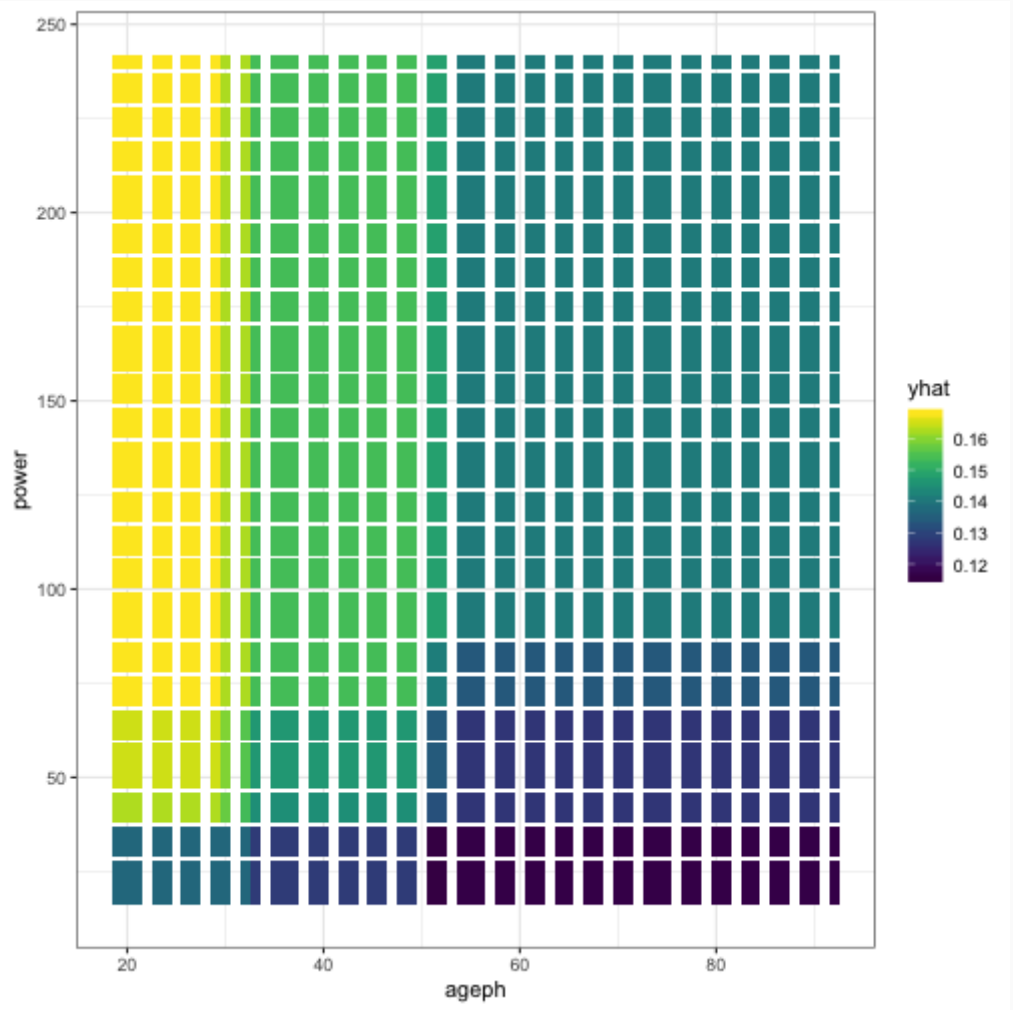
```r
# Need to define this helper function for Poisson
pred.fun <- function(object,newdata){
  mean(predict(object, newdata))
}

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

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



Partial dependence plot in two dimensions

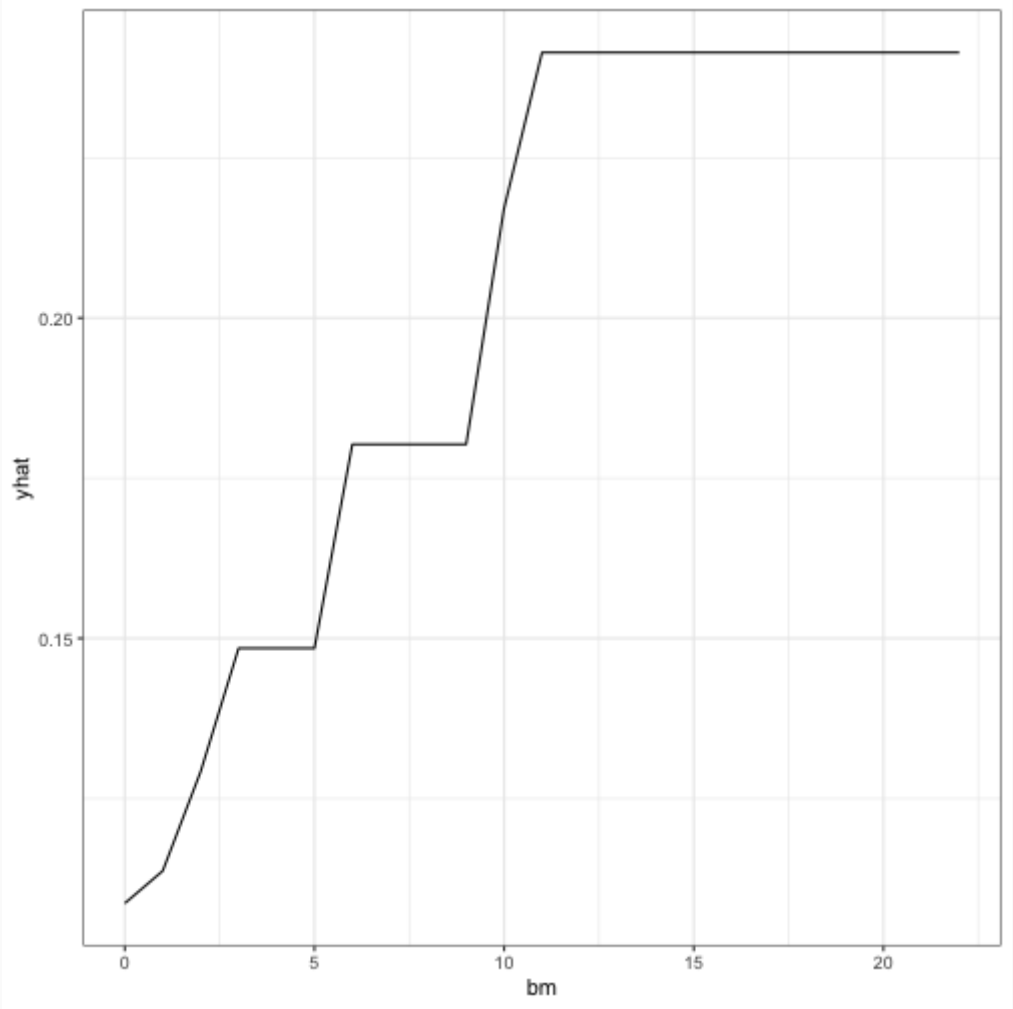
```
# partial: computes the marginal effect
# autoplot: creates the graph using ggplot2
fit_srt %>%
  partial(pred.var = c('ageph','power'),
          pred.fun = pred.fun,
          train = mtpl_trn[pdp_ids,]) %>%
  autoplot()
```



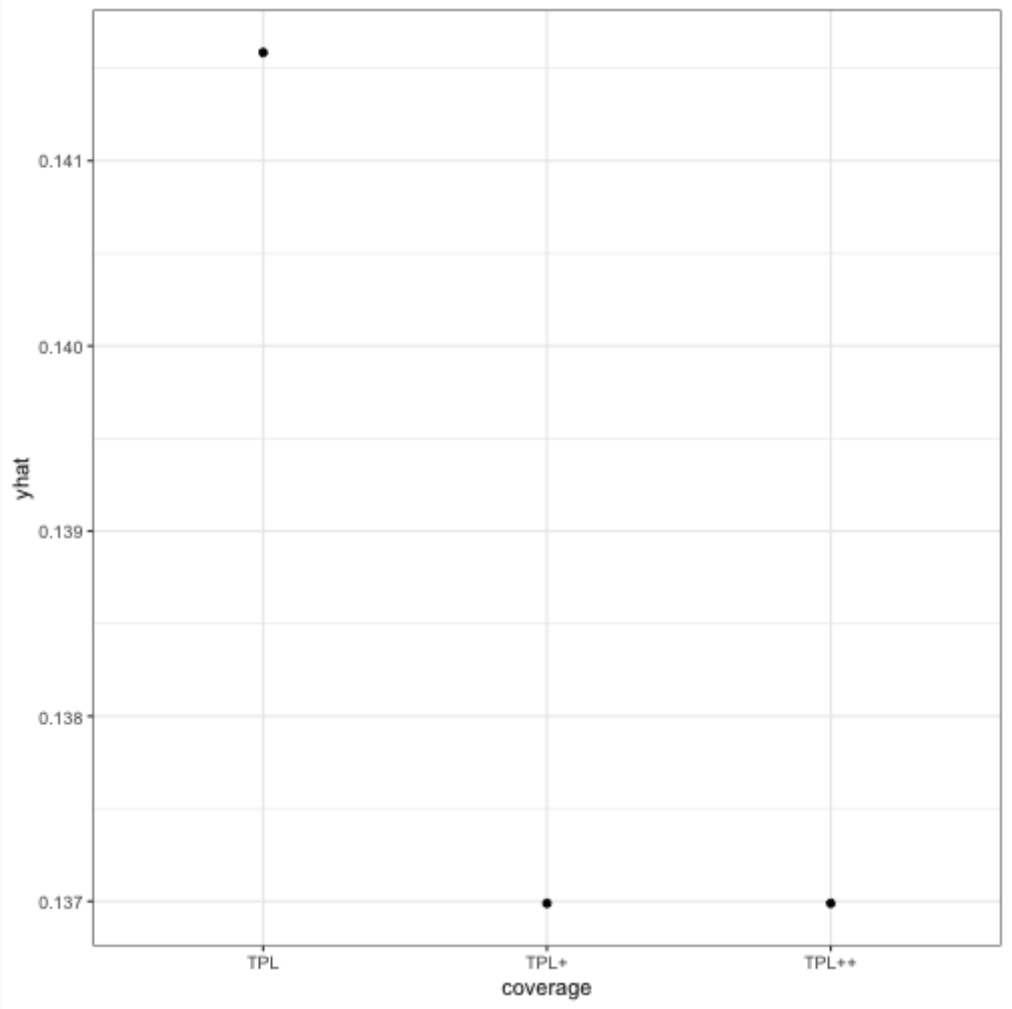
Use partial dependence plots for **other features** to understand your model completely.

Your turn

Level in the bonus-malus scale



Type of coverage



It's a wrap!

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

Disdvantages ☹

- 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**

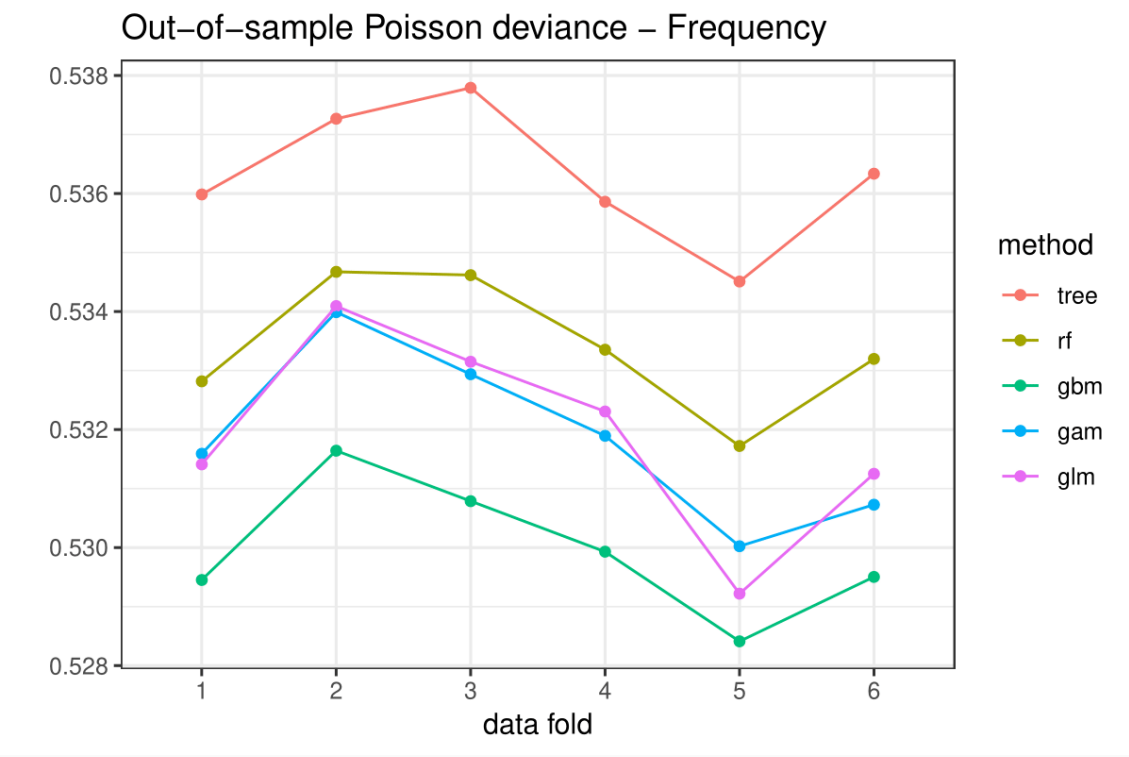
Ensembles of trees

- Remember: **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)

1. Bagging:
    - low **bias** via detailed individual models
    - low **variance** via averaging of those models
    - random forest is a modification on bagging for trees to improve the variance reduction
2. Boosting
    - low **variance** via simple individual models
    - low **bias** by incrementing the model sequentially

My own experience

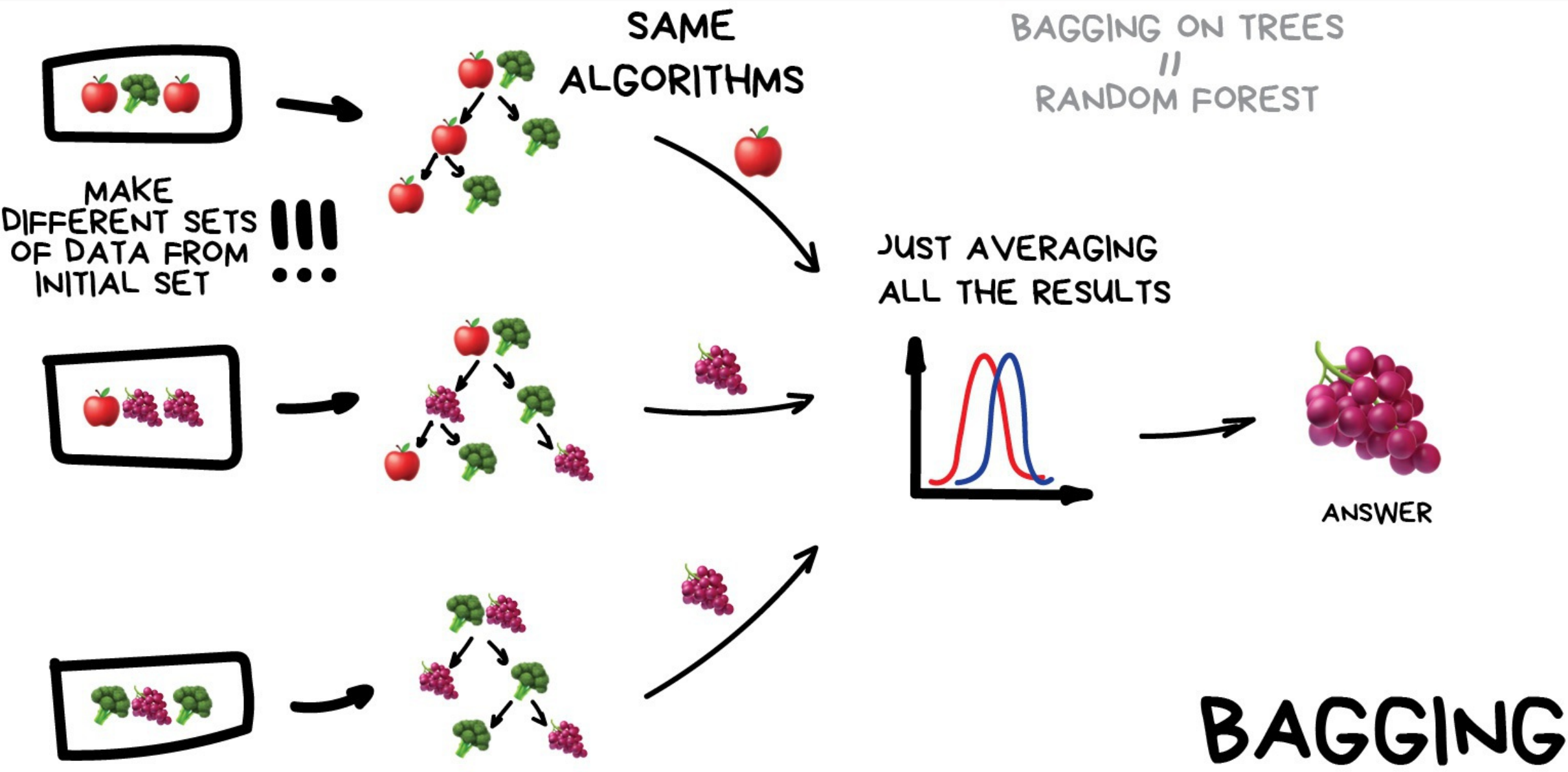
Boosting > Random forest > Bagging > Single tree



From my paper on Boosting insights in insurance tariff plans with tree-based machine learning methods

More details @ Henckaerts et al. (2019, arXiv).

Bagging and random forest


SAME
ALGORITHMS
BAGGING ON TREES
=
RANDOM FOREST
MAKE
DIFFERENT SETS
OF DATA FROM
INITIAL SET
!!!
JUST AVERAGING
ALL THE RESULTS
ANSWER
BAGGING

Bagging

- Bagging stands for **B**ootstrap **AGG**regat**ING**
- 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 decison trees

Bagging

- Bagging stands for **B**ootstrap **AGG**regat**ING**
- 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 decison trees

Bagging with trees?

- Do the following 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
  - **majorty** vote for classification
- Implemented in the {ipred} package 
  - uses {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) %>% print(n = 5)
dfr_b2 %>% dplyr::arrange(x) %>% print(n = 5)
```

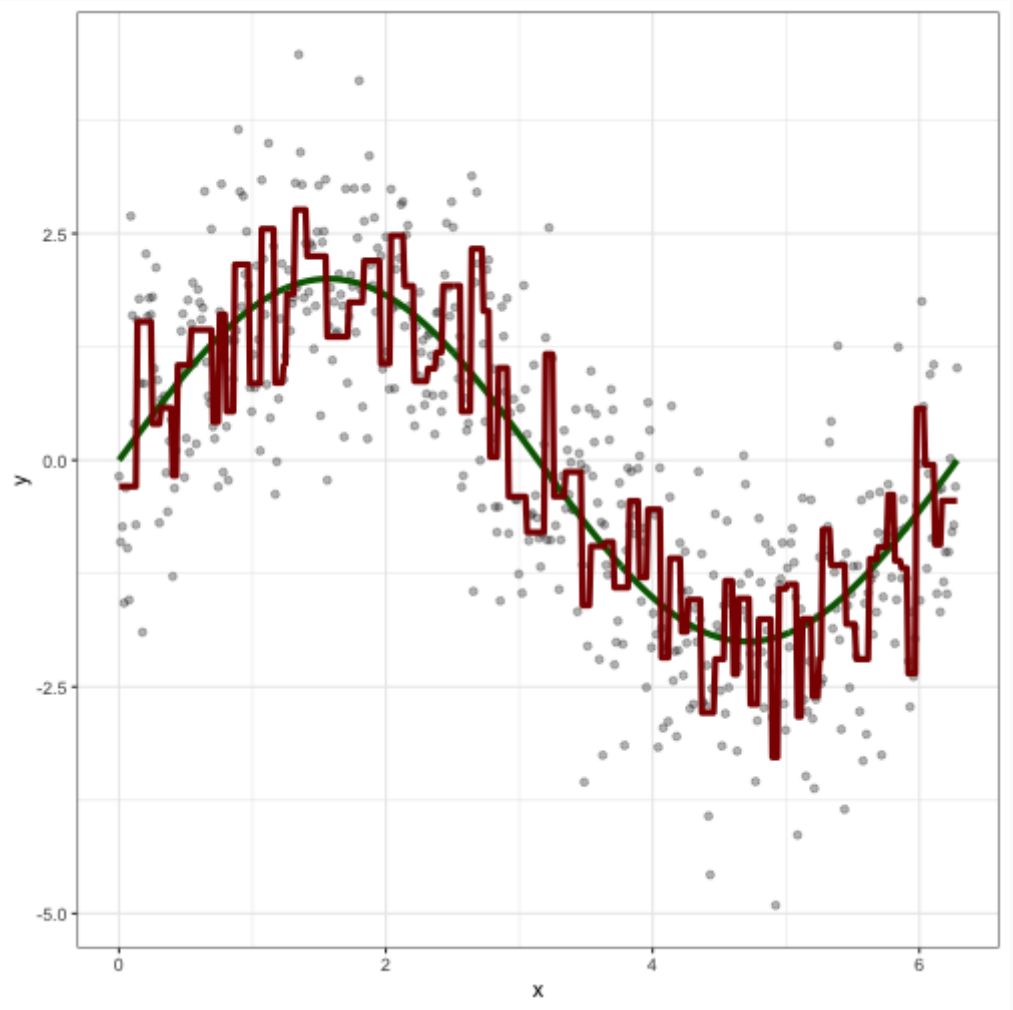
```
## # A tibble: 500 x 3
##         x      m      y
##     <dbl>  <dbl>  <dbl>
## 1 0.0252 0.0504 -0.734
## 2 0.0252 0.0504 -0.734
## 3 0.0378 0.0755 -1.58
## 4 0.0630 0.126  -0.970
## 5 0.101  0.201   1.60
## # … with 495 more rows
```

```
## # A tibble: 500 x 3
##         x      m      y
##     <dbl>  <dbl>  <dbl>
## 1 0      0      -0.179
## 2 0      0      -0.179
## 3 0      0      -0.179
## 4 0.0126 0.0252 -0.903
## 5 0.0630 0.126  -0.970
## # … with 495 more rows
```

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
                  )
                )
```

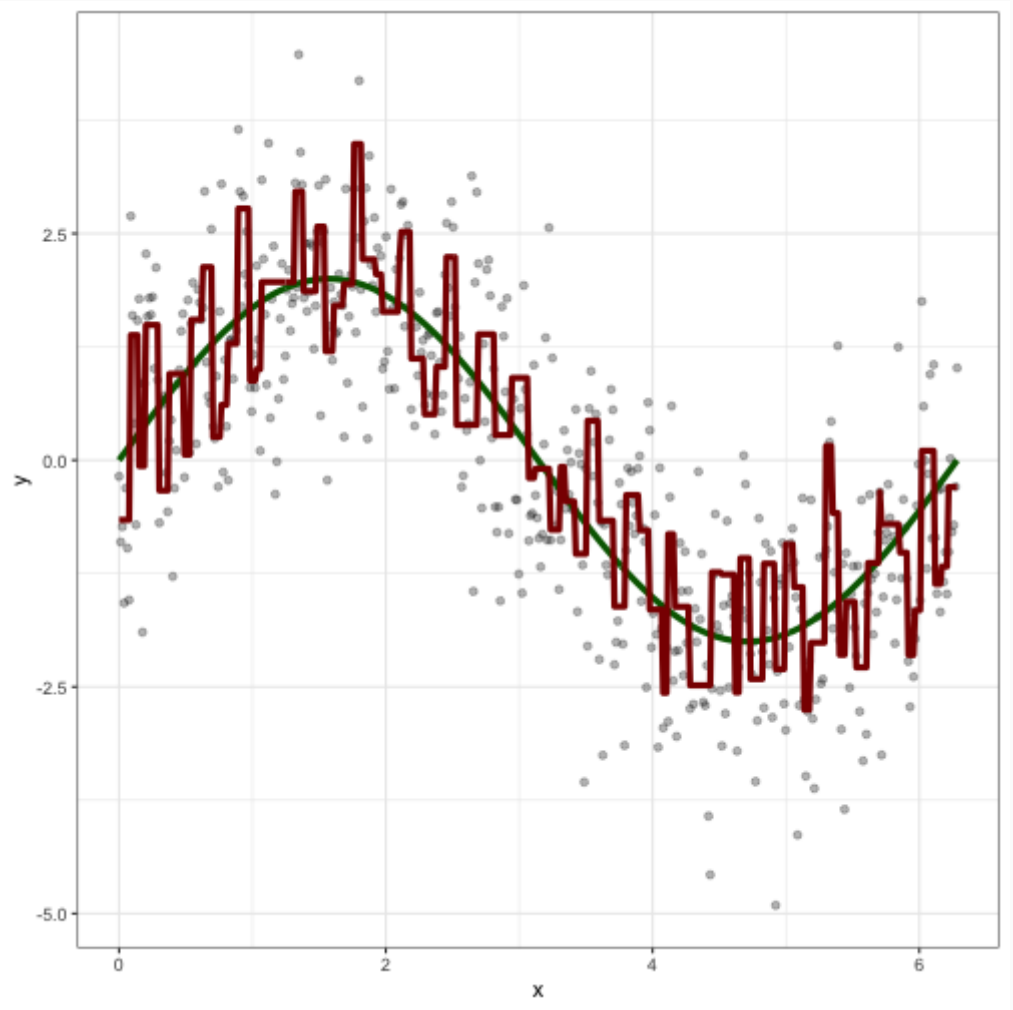
On it's own, this is a **noisy prediction** with very high variance



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

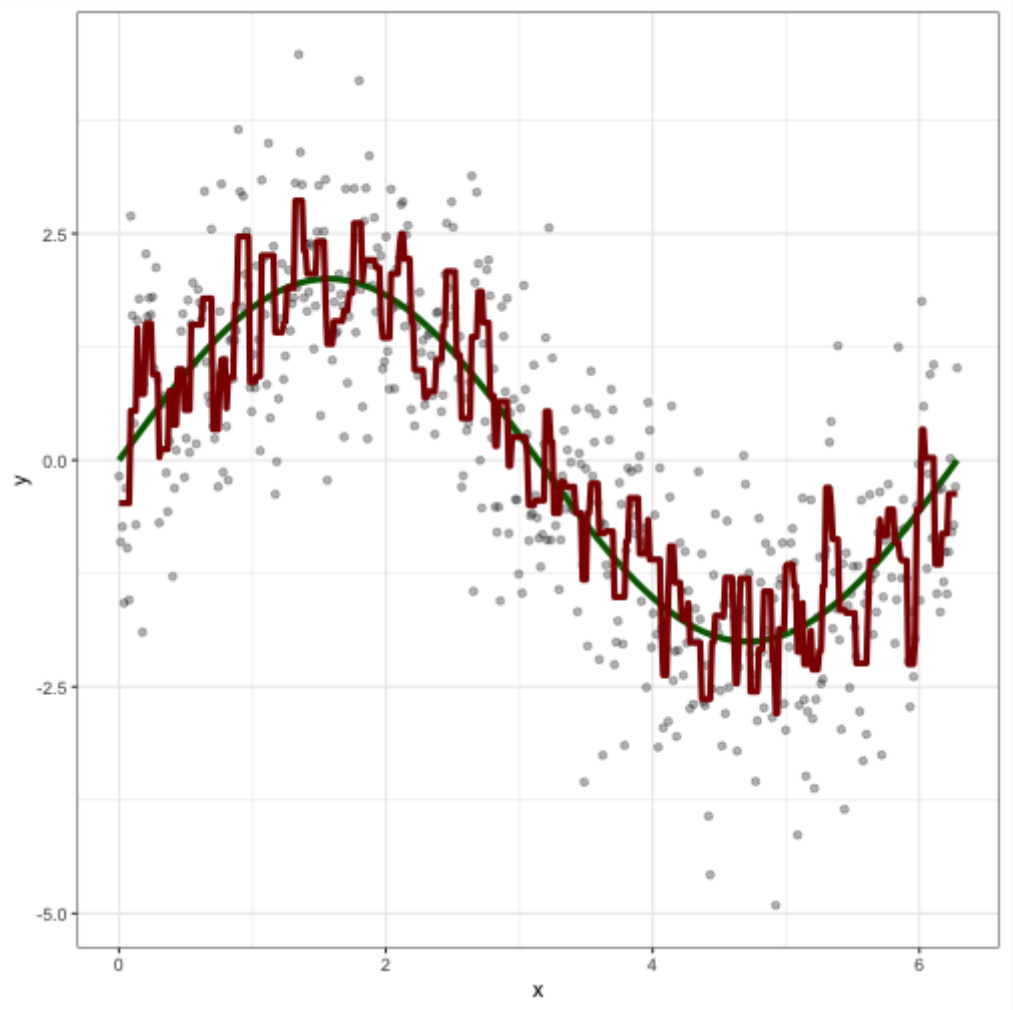


Combining the predictions of both trees

```
# 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

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)
```

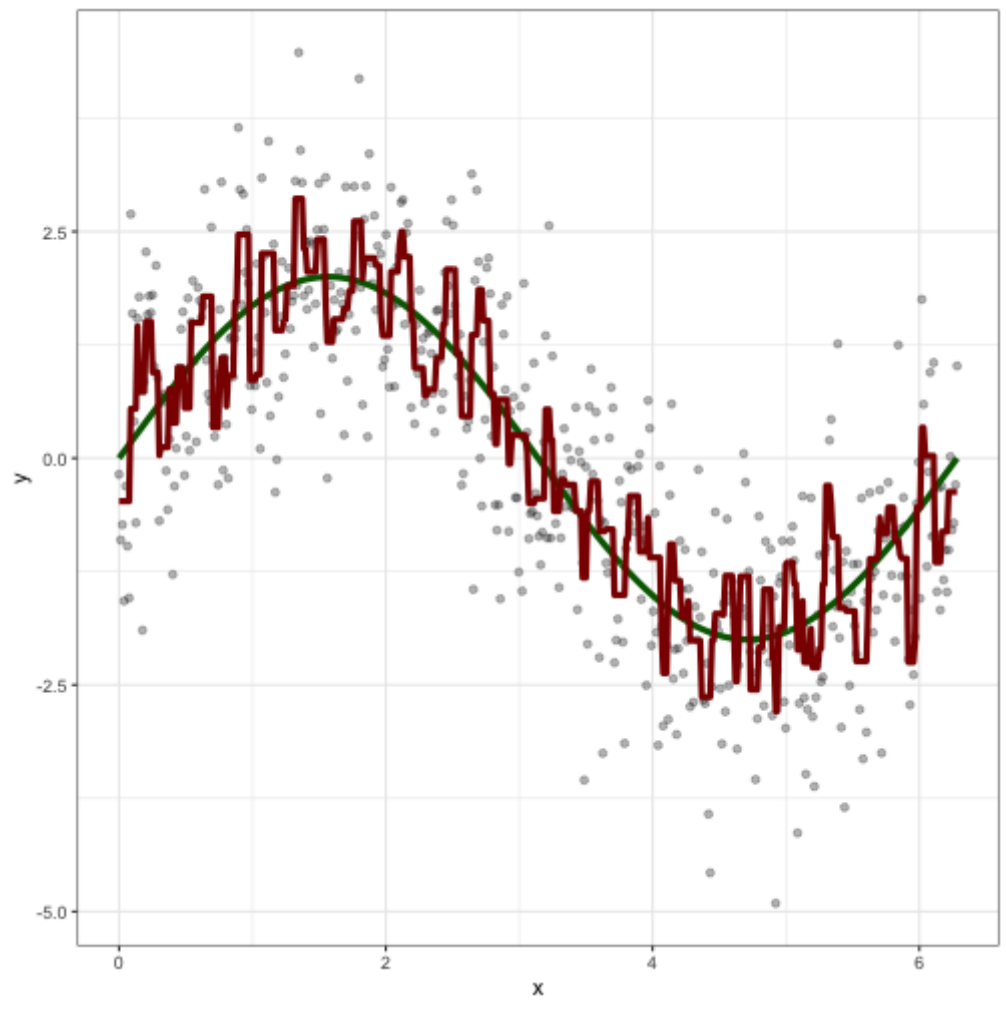
Q2: fit a deep tree

```
# 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))
```

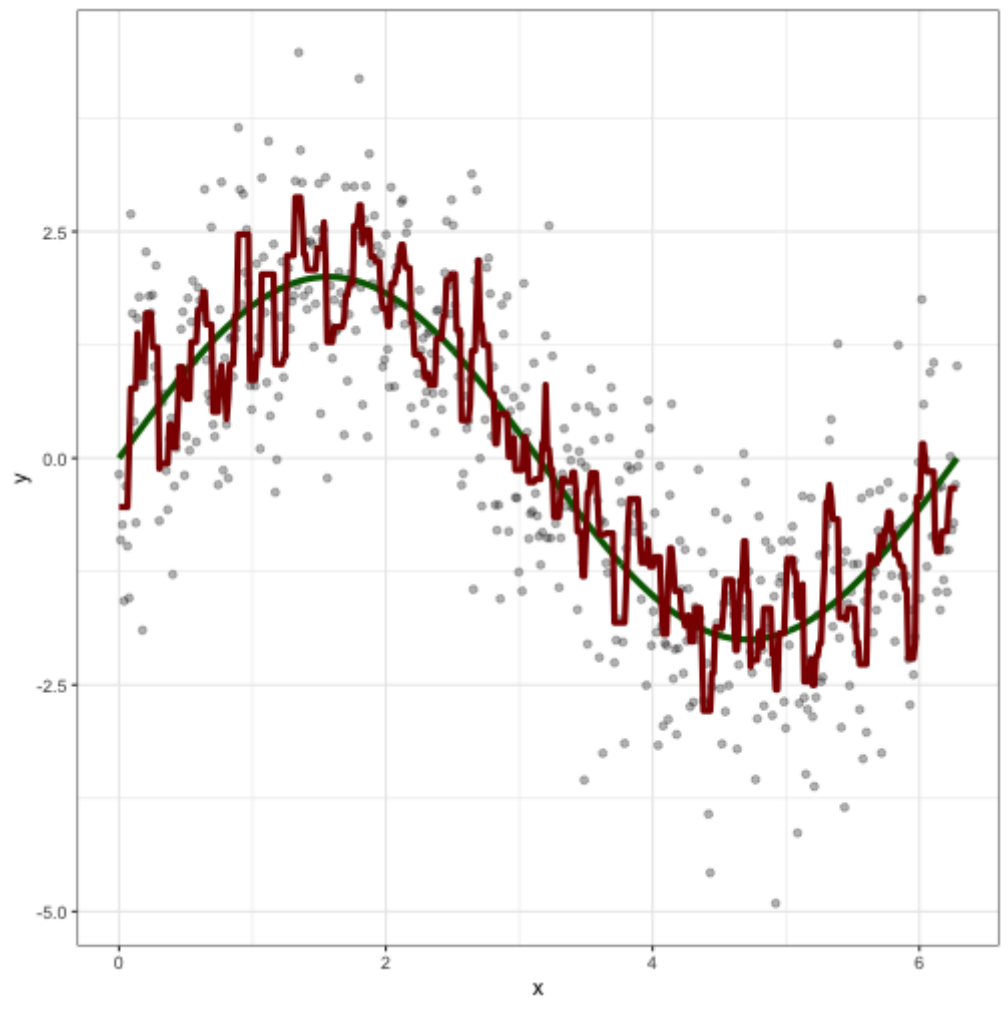
Q3: average the predictions

```
# 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 with replacement
- Some observations **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 with replacement
- Some observations **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

- Is it a **representative** sample though?

```
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

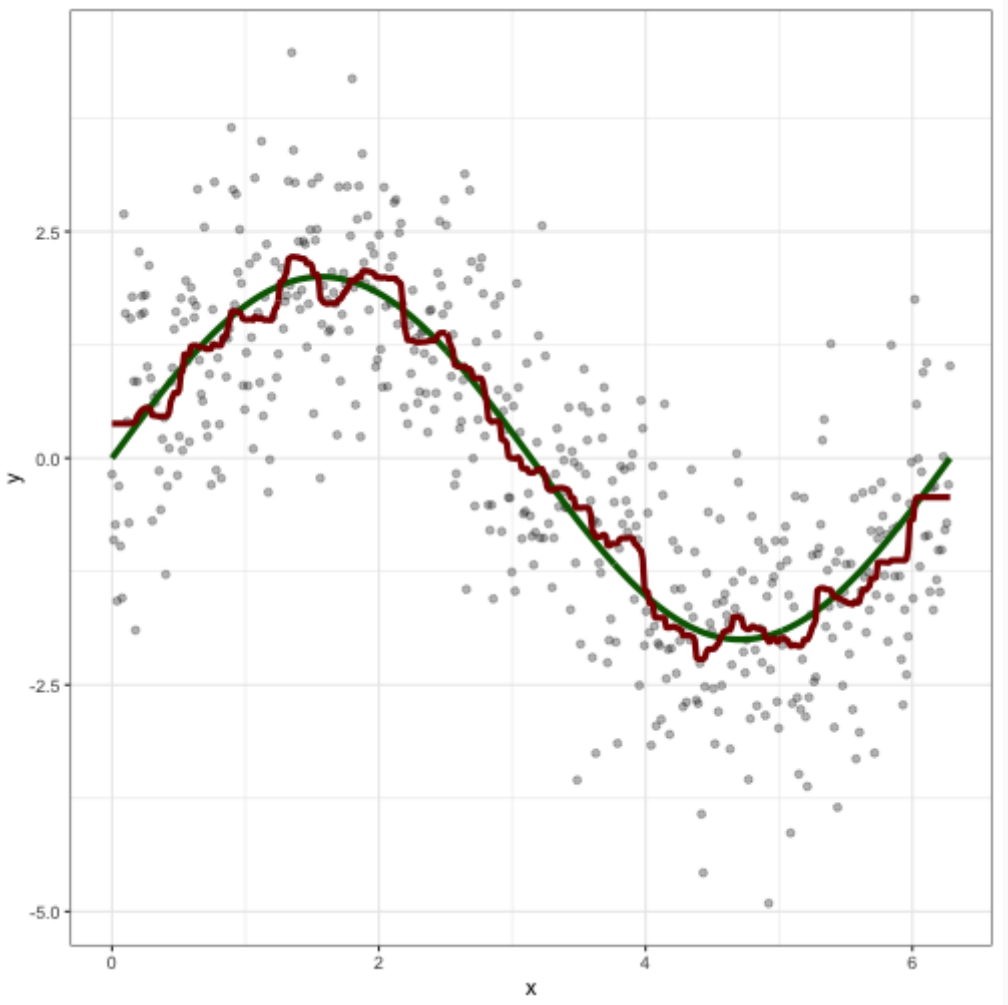
```
mean(x %in% sample(N,
                   size = 0.75*N,
                   replace = TRUE))
```

```
## [1] 0.52837
```

Bagging properly

```
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)
                      )
```

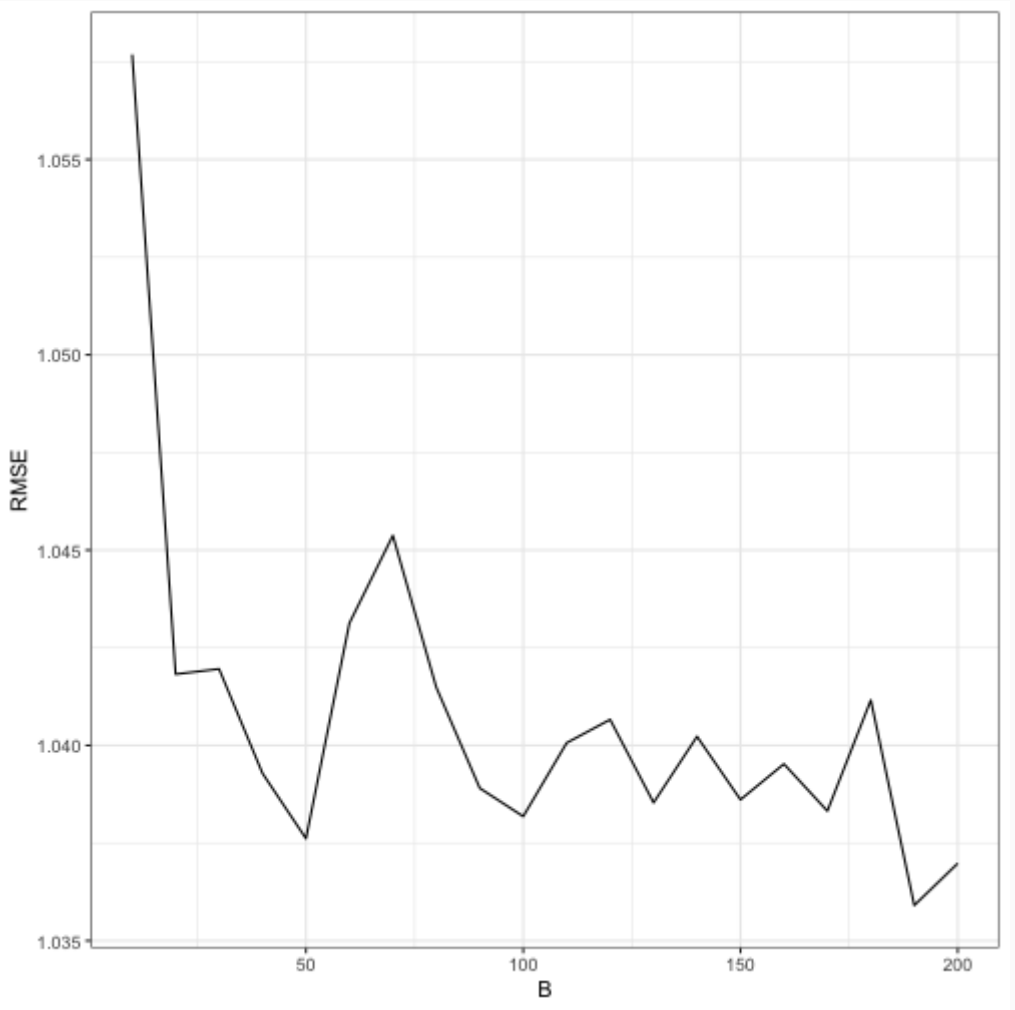
```
# Predict from this model
pred <- predict(fit, dfr)
```



😊 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
}
```



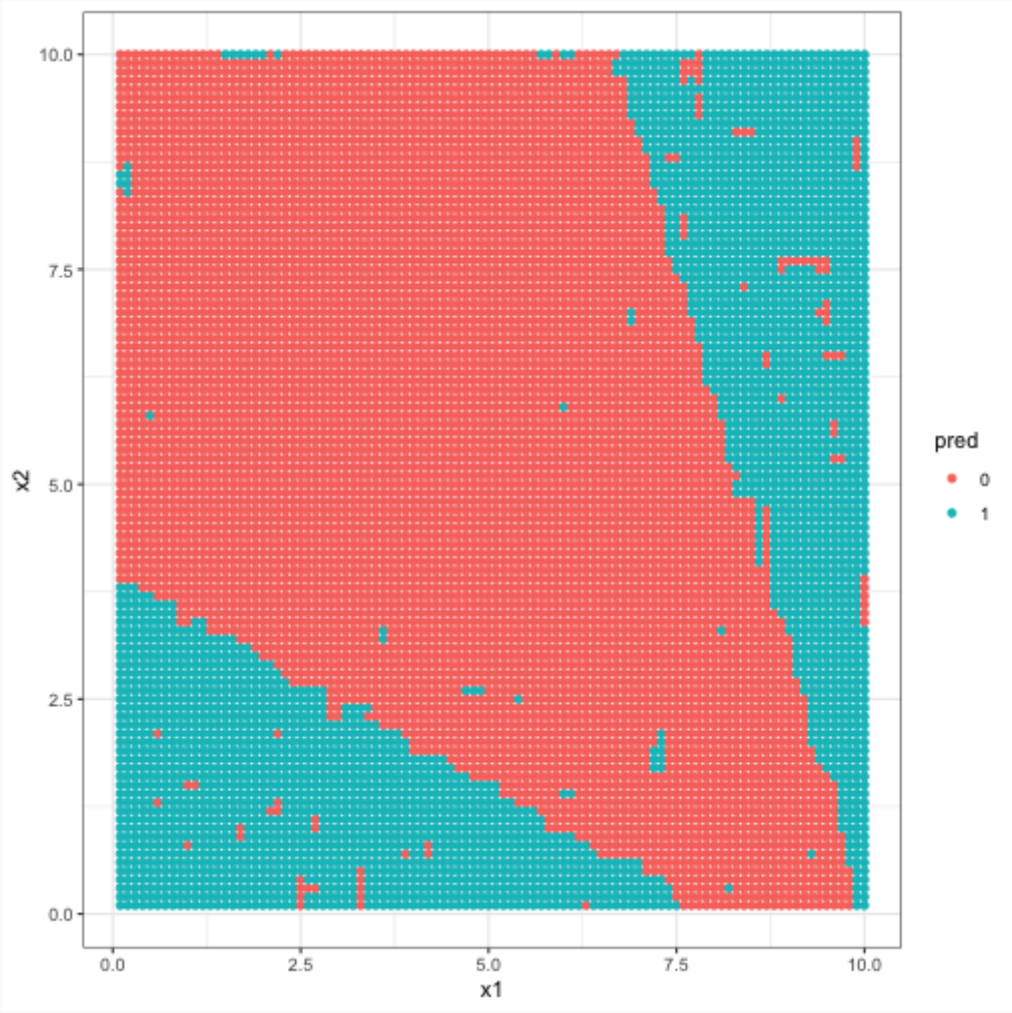
Your turn

Use {ipred} to fit a bagged tree ensemble for the toy **classification** problem with data `dfc`.
Experiment with the `nbagg` and `control` parameters to see their effect on the predictions.

Q: these 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'
                 )
```



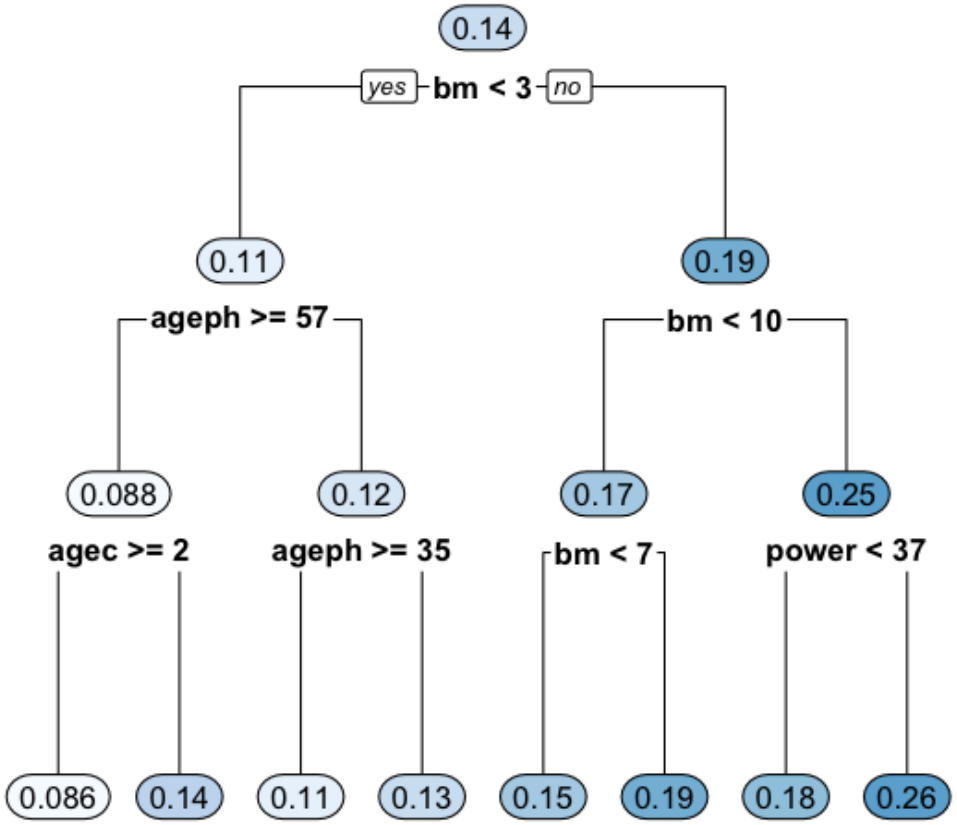
Back to the MTPL data

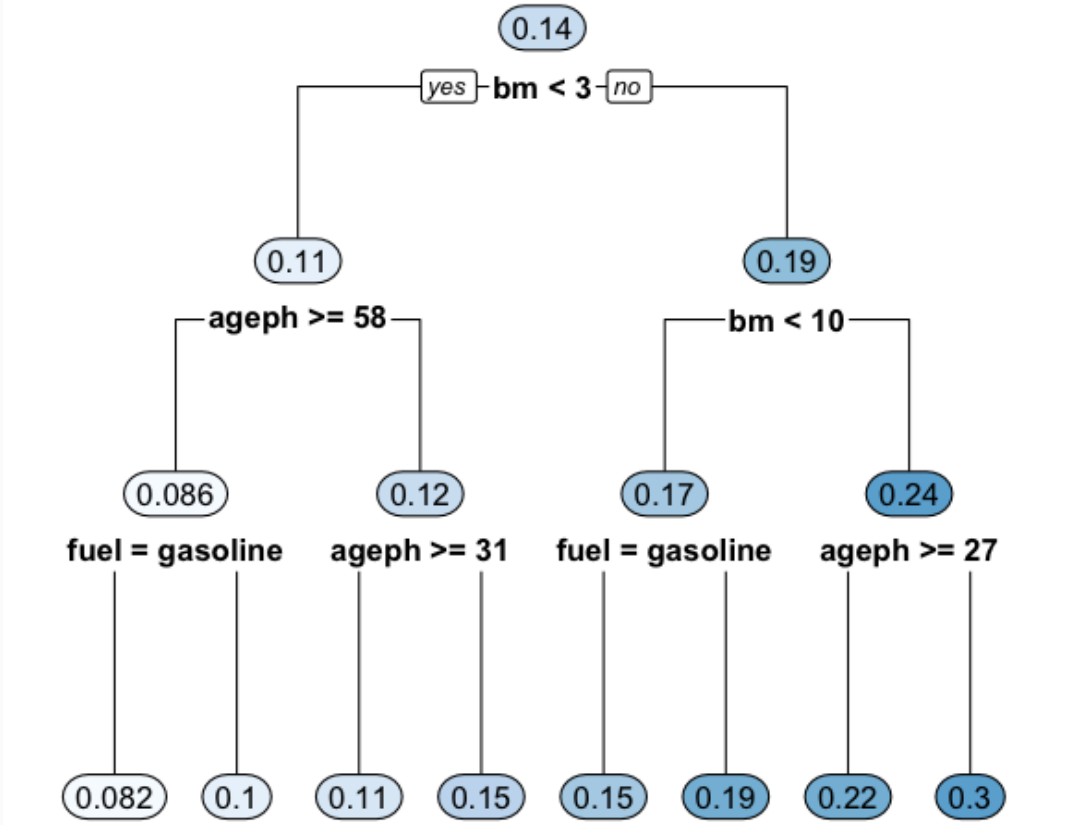
- Generate **two bootstrap samples**:

```
set.seed(486291) # reproducibility

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

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

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

- We now use {rpart} to fit a tree to **each sample**
- Let's inspect the first splits in each tree

Pretty similar, right?!





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 earlier?
  - `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 preditive 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

Suppose you have `p=10` features in your data.
Randomly pick `m=4` features as split candidates each time.
How often will a feature be an option to split on, in a tree with `n=100` splits?

1. Using basic probablity, what is the **theoretical** answer?
2. Verify this **numerically** via `sample` in R.

Q1: theoretically the answer is (m/p)*n = 40 times

```
c(p,m,n) %<-% c(10,4,100)
(m/p)*n
```

```
## [1] 40
```

Notice the very convenient **unpacking** operator `%<-%` which is available in the {zeallot} package

Q2: numerically we come close to the 40 for each feature

```
set.seed(54321)

samples <- sapply(1:n,
                  function(i) sample(p,
                                     size = m)
                  )
samples[,1:9]
```

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

```
sapply(1:p,
       function(i) sum(samples == i)
       )
```

```
##  [1] 35 43 36 41 41 35 42 46 42 39
```

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

- Many tuning parameters in a **random forest**:
  - number of trees
  - number of candidates for splitting
  - max tree depth
  - minimun leaf node size
  - sample fraction
- Set up 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       163.212    NA
## 2        200    3       163.212    NA
## 3        100    6       163.212    NA
## 4        200    6       163.212    NA
## 5        100    9       163.212    NA
## 6        200    9       163.212    NA
## 7        100    3      1632.120    NA
## 8        200    3      1632.120    NA
## 9        100    6      1632.120    NA
## 10       200    6      1632.120    NA
## 11       100    9      1632.120    NA
## 12       200    9      1632.120    NA
```

Tuning strategy

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

```
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_trn,
    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      1632.120 0.1327988
## 2        100    3      1632.120 0.1328027
## 3        200    6      1632.120 0.1328680
## 4        100    6      1632.120 0.1328943
## 5        200    9      1632.120 0.1328953
## 6        100    9      1632.120 0.1329139
## 7        200    3       163.212 0.1333084
## 8        100    3       163.212 0.1333774
## 9        200    6       163.212 0.1336937
## 10       100    6       163.212 0.1338166
## 11       200    9       163.212 0.1338307
## 12       100    9       163.212 0.1339166
```

What does the prediction error measure actually?

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

Random forest 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 ☹
- Luckily, there exists a solution to this problem
- The {distRforest} package on my 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. (2019, arXiv)

Using {distRforest}

```
rforest(formula, data, method, control = rpart.control(___),
        ntrees, ncand, subsample, redmem)
```

- `formula`: a formula as *response ~ feature1 + feature2 + ...*
- `data`: the observation data containing the response and features
- `method`: a string specifying which **loss function** to use (anova, class, poisson, gamma, lognormal)
- `control`: options to pass to `rpart.control` for the individual trees
- `ntrees`: the number of **trees** in the ensemble
- `ncand`: the number of **candidate** features for splitting
- `subsample`: fraction of observations to sample
- `redmem`: a logical indicating whether or not to reduce memory on the rpart trees

Random forest on the MTPL data

Let's fit a Poisson random forest to the **MTPL** data:

```
set.seed(54321) # reproducibility
fit_rf <- rforest(
  formula = cbind(expo,nclaims) ~
    ageph + agec + bm + power + coverage +
    fuel + sex + fleet + use,
  data = mtpl_trn,
  method = 'poisson',
  control = rpart.control(
    maxdepth = 20,
    minsplit = 2000,
    minbucket = 1000,
    cp = 0,
    xval = 0
    ),
  ntrees = 100,
  ncand = 5,
  subsample = 0.75,
  redmem = TRUE
  )
```

```
class(fit_rf)
```

```
## [1] "rf"   "list"
```

```
class(fit_rf[[1]])
```

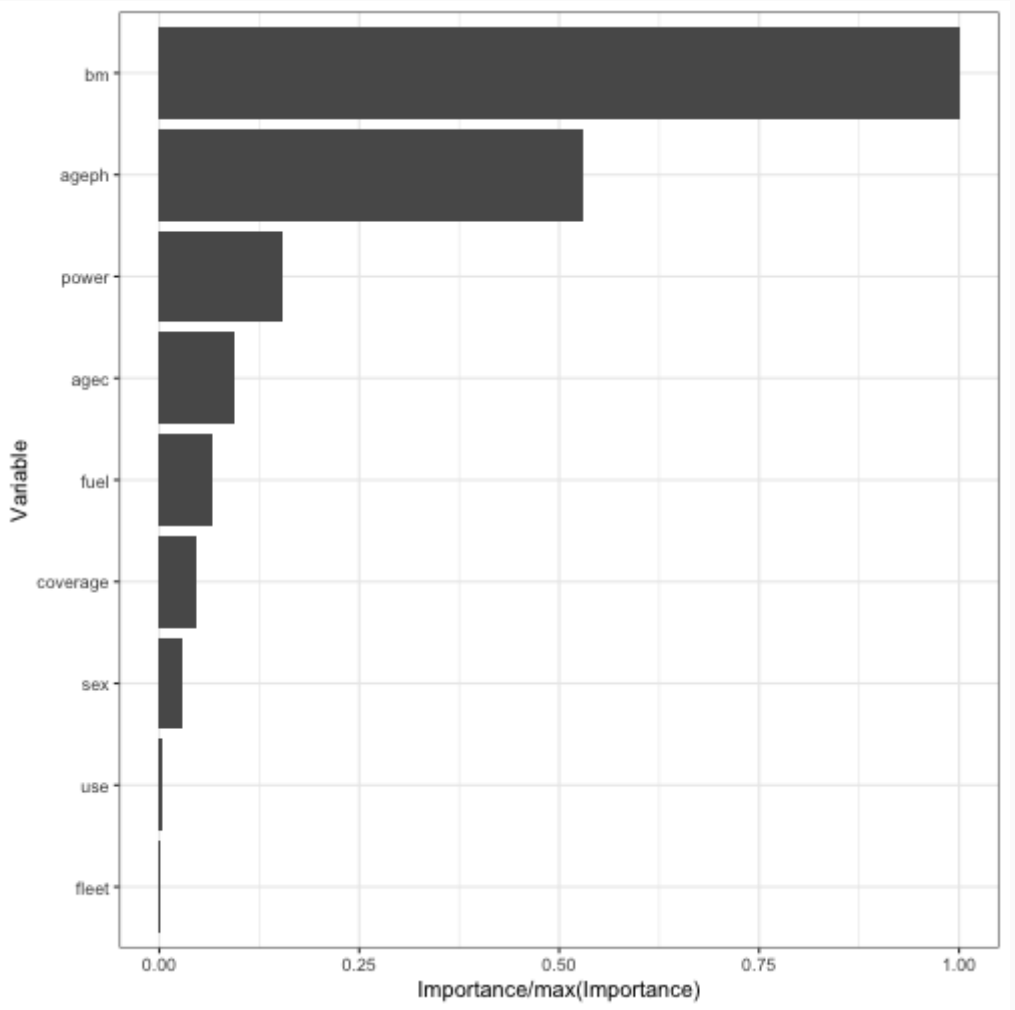
```
## [1] "rpart"
```

The `rforest` function returns a **list** of rpart trees.

Feature importance

```
# Get vi of each individual tree
var_imps <- lapply(fit_rf, vip::vi)
# Some data-wrangling to get rf vi
do.call(rbind, var_imps) %>%
  group_by(Variable) %>%
  summarise(Importance = mean(Importance)) %>%
  arrange(-Importance)
```

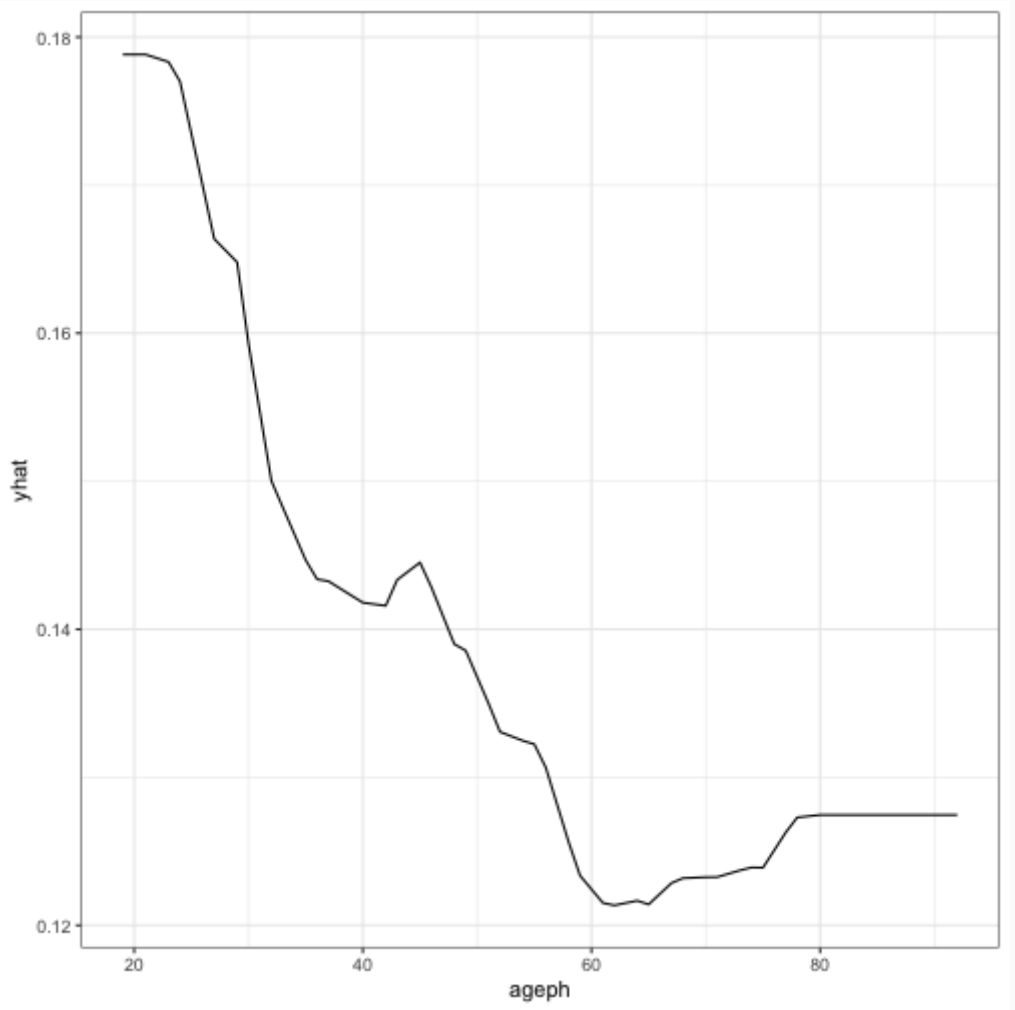
```
## # A tibble: 9 x 2
##   Variable Importance
##   <chr>         <dbl>
## 1 bm          1151.
## 2 ageph        609.
## 3 power        179.
## 4 agec         109.
## 5 fuel          75.2
## 6 coverage      53.6
## 7 sex           34.3
## 8 use            3.02
## 9 fleet          1.07
```



Partial dependence plot

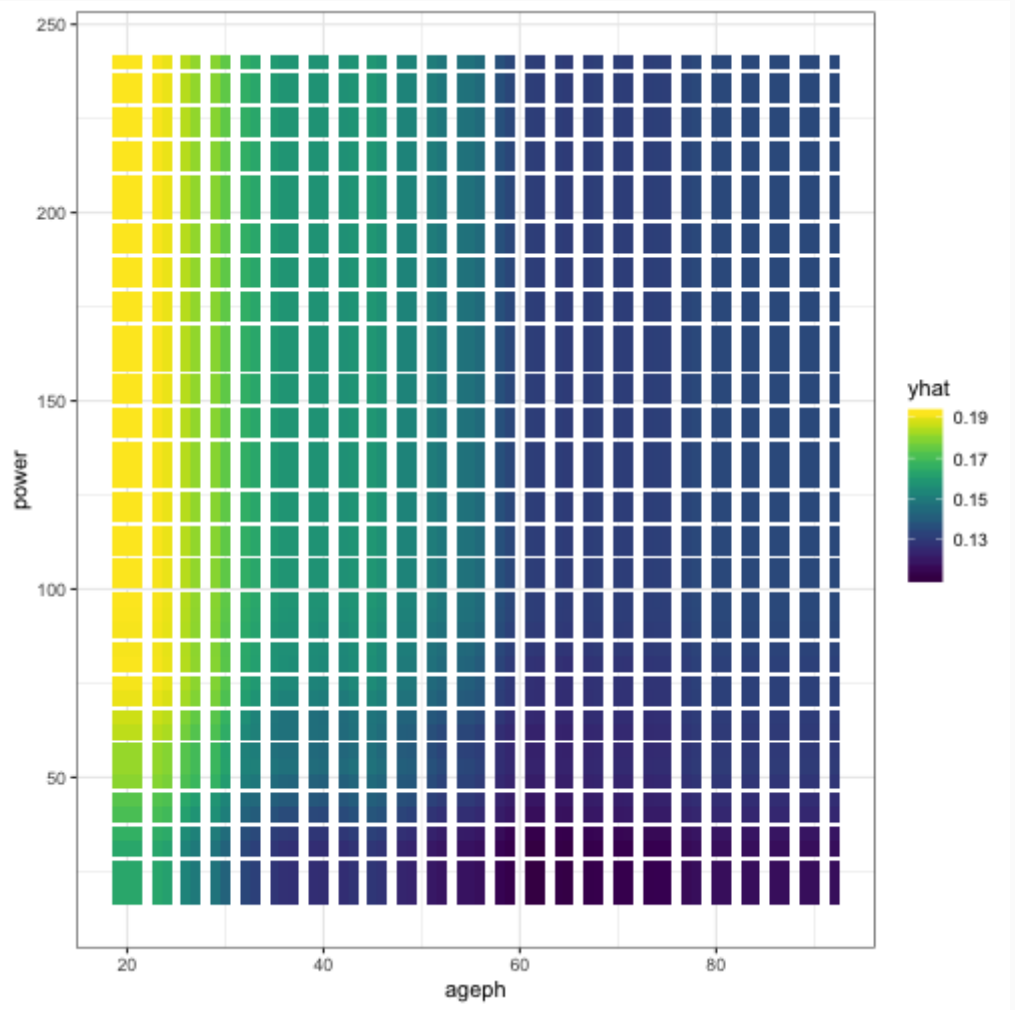
```r
# Need to define this helper function
pred.fun <- function(object,newdata){
  pred <- rep(0,nrow(newdata))
  for(i in 1:length(object)) {
    pred <- pred + predict(object[[i]], newdata)
  }
  return(mean((1/length(object))*pred))
}
```

```r
# partial: computes the marginal effect
# autoplot: creates the graph using ggplot2
fit_rf %>%
  partial(pred.var = 'ageph',
          pred.fun = pred.fun,
          train = mtpl_trn[pdp_ids,]) %>%
  autoplot()
```



Partial dependence plot in two dimensions

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



Your turn

That's a wrap on **bagging** and **random forest**! Now it's your time to experiment.

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

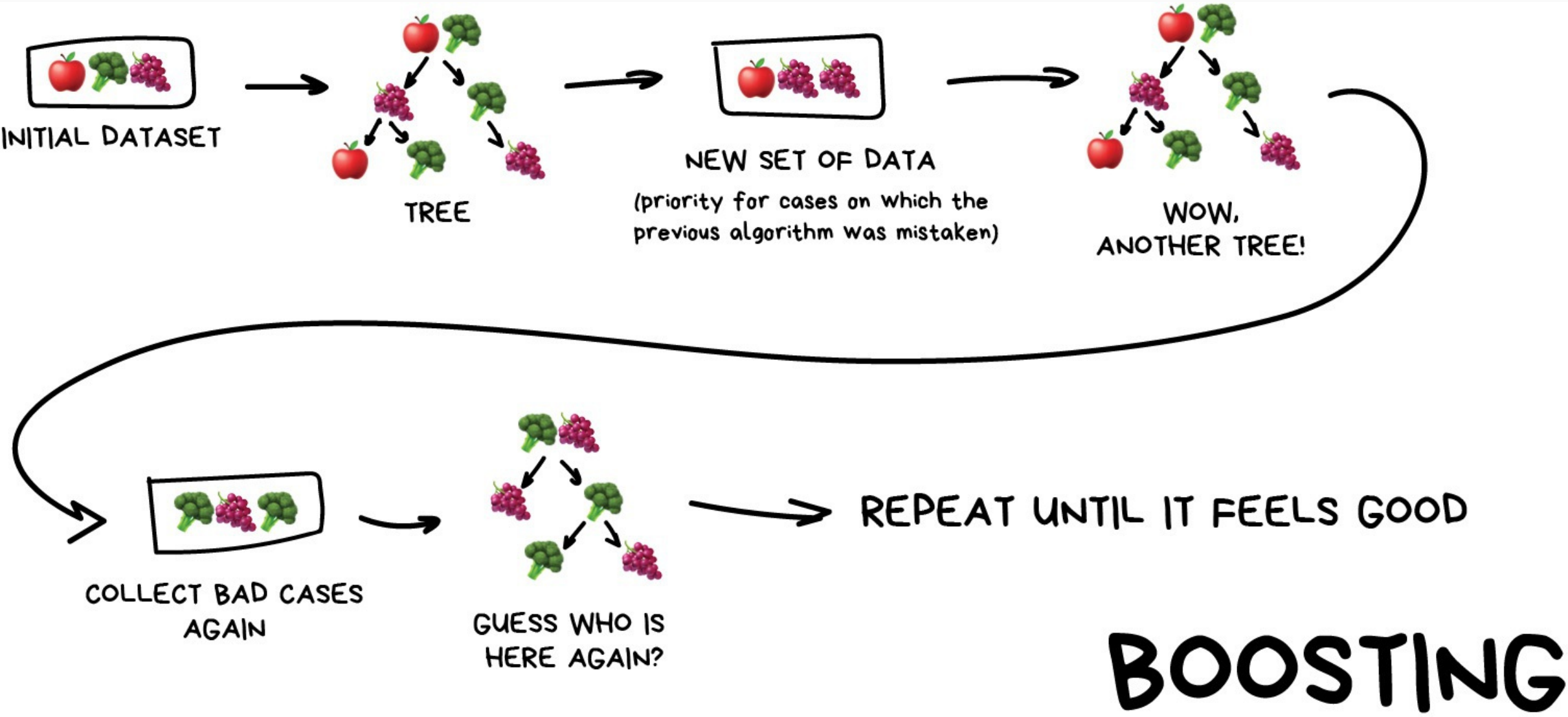
1. Use {distRforest} with a **Gamma** or **log-normal** deviance to build a severity random forest. The `mtpl` data contains the average claim amount in the feature `average`.

2. Use {ranger} to develop a random forest for the **Ames Housing** data and extract insights in the form of feature importance and partial dependence plots.

3. Develop a classification random forest to predict the **occurence** of a claim.

4. Compare insights obtained from a regression tree and a random forest. Does the extra **flexibility** result in big differences?

Gradient boosting machine


INITIAL DATASET
TREE
NEW SET OF DATA
(priority for cases on which the previous algorithm was mistaken)
WOW, ANOTHER TREE!
REPEAT UNTIL IT FEELS GOOD
COLLECT BAD CASES AGAIN
GUESS WHO IS HERE AGAIN?
BOOSTING

Boosting

- 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:

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

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

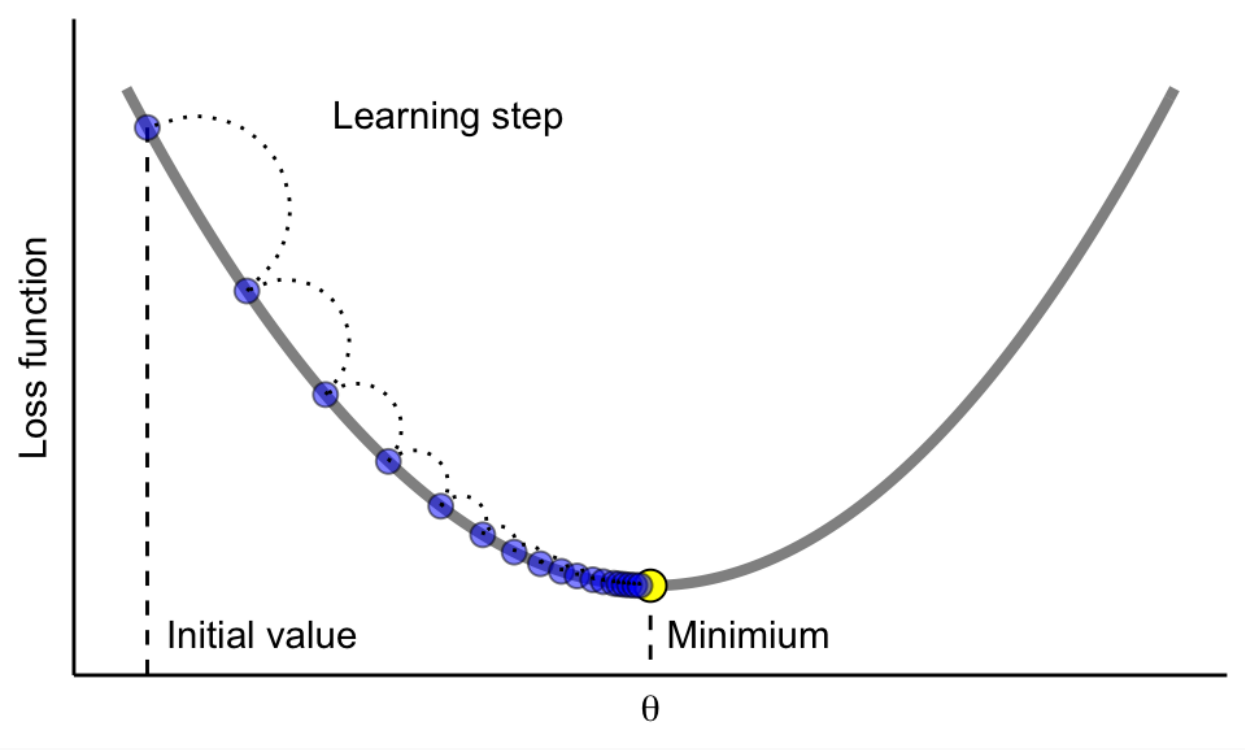


Figure 12.3 from Bradley Boehmke's HOML

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

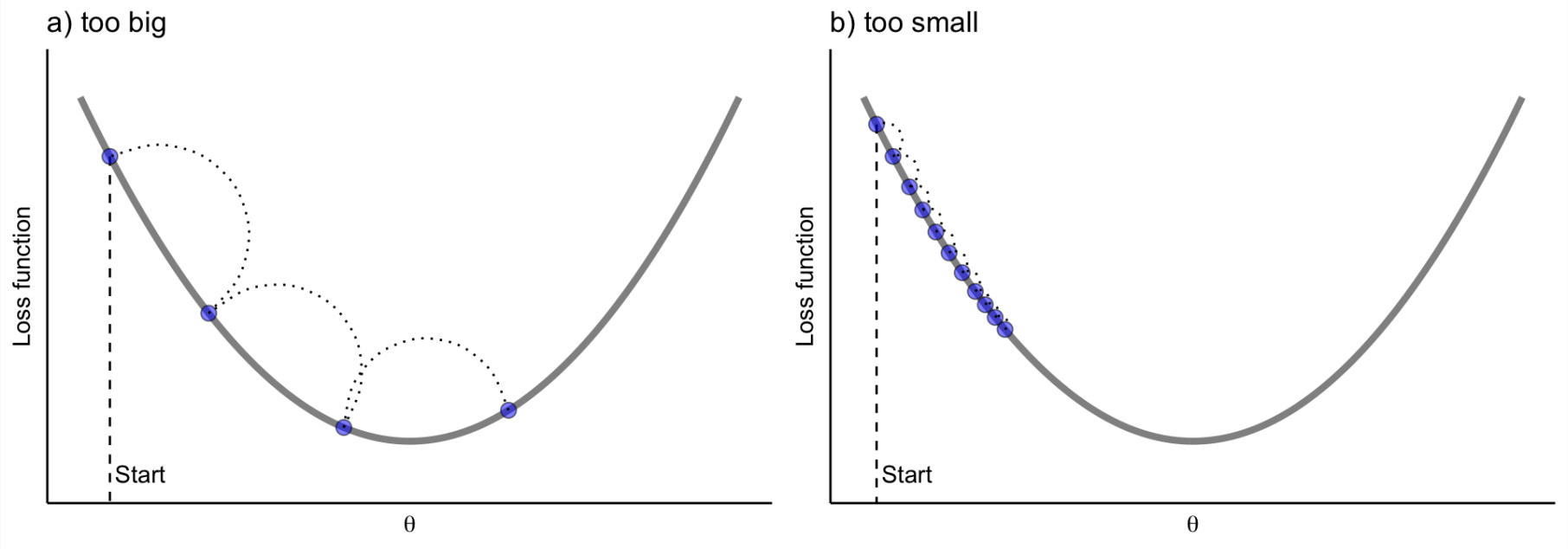


Figure 12.4 from Bradley Boehmke's HOML

- 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 **shrunken** 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

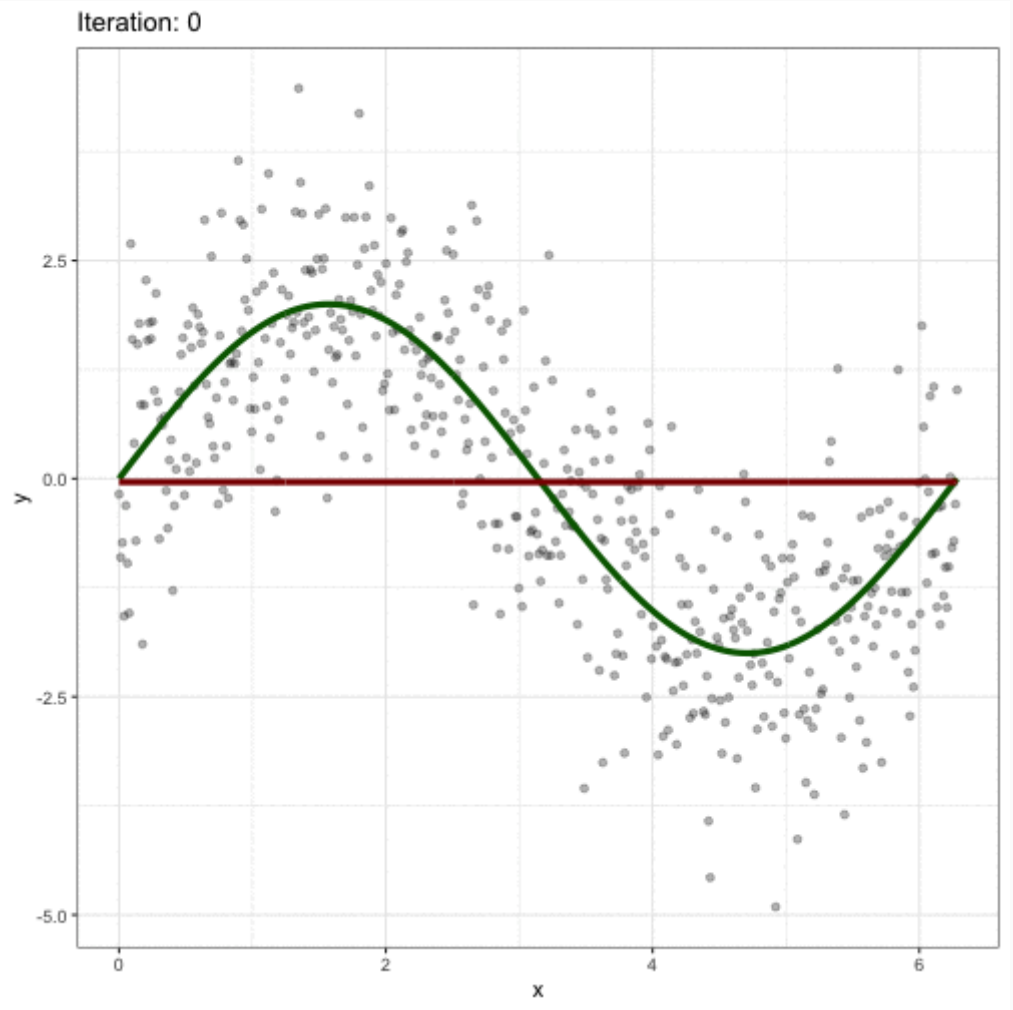
GBM parameters

- A lot of **parameters** available 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

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

```
# 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

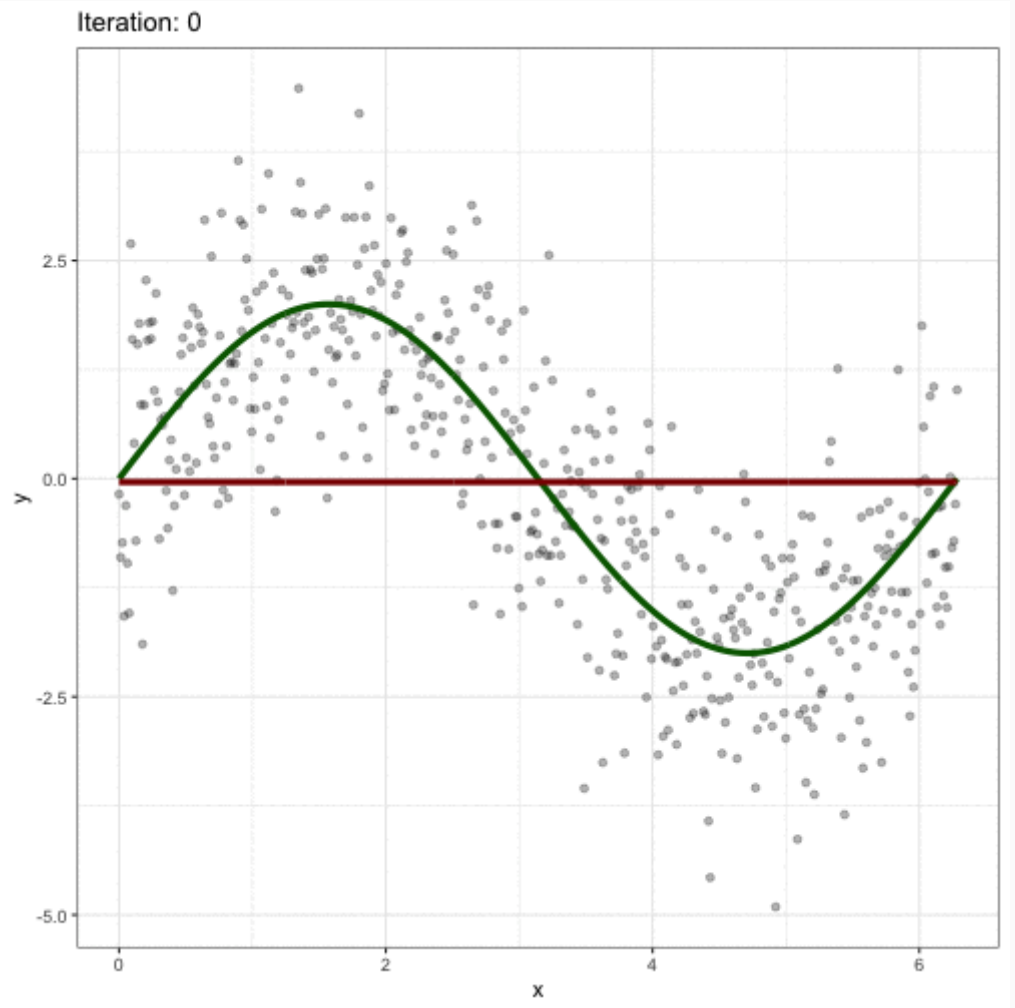
Fit a GBM of 10 stumps, **with** shrinkage:

```
# 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

☹ need more trees and **longer** training time



GBM parameters

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

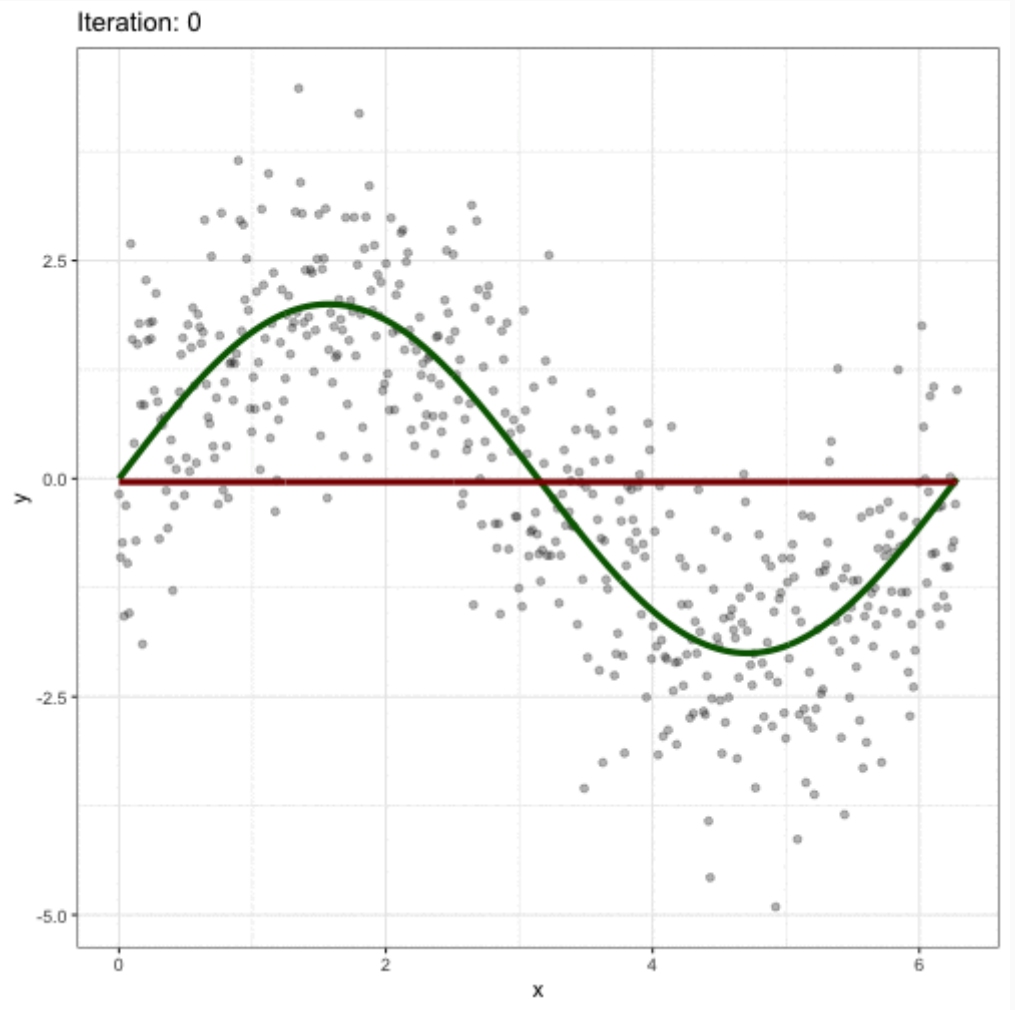
```
# Fit the GBM
fit <- gbm(formula = y ~ x,
           data = dfr,
           distribution = 'gaussian',
           n.trees = 10,
           interaction.depth = 3,
           shrinkage = 0.1
           )
```

Increasing tree depth allows more versatile splits

😊 **faster** learning

☹ risk of **overfitting** (shrinkage important!)

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



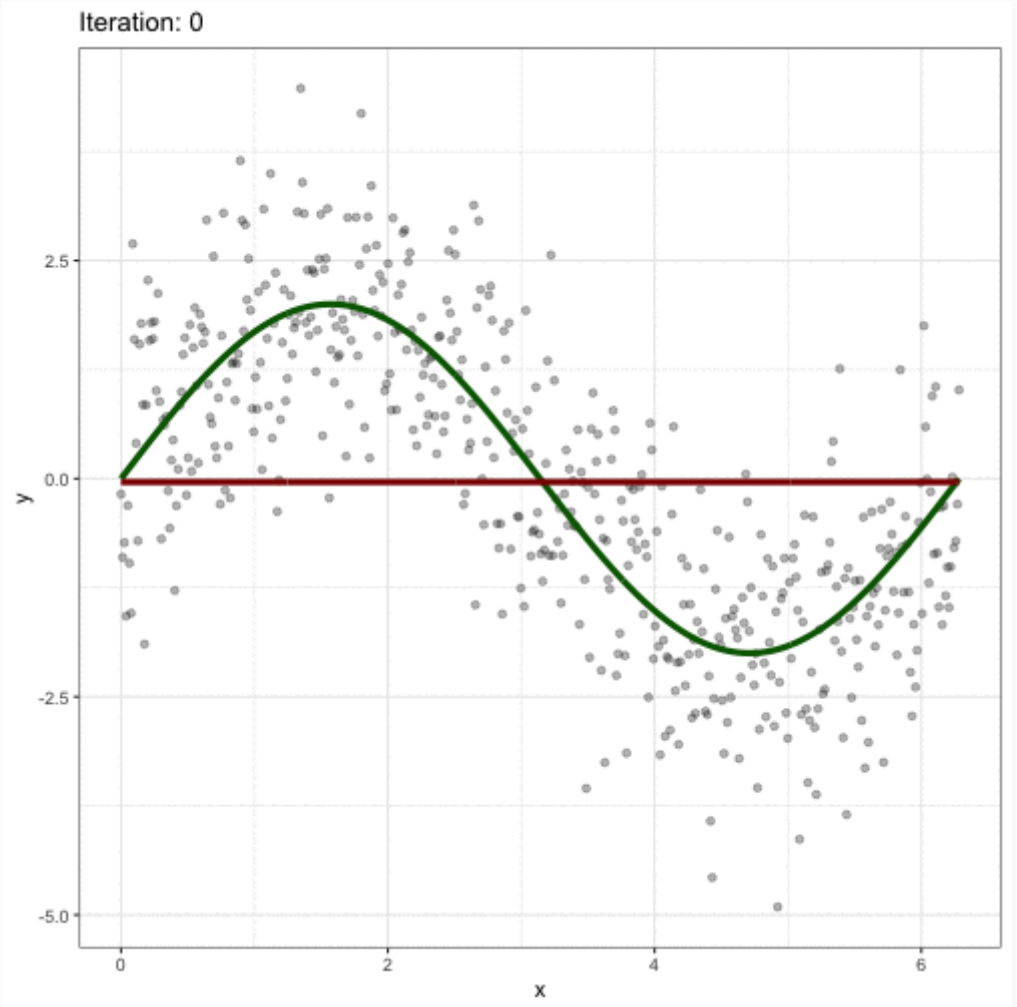
GBM parameters

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**)



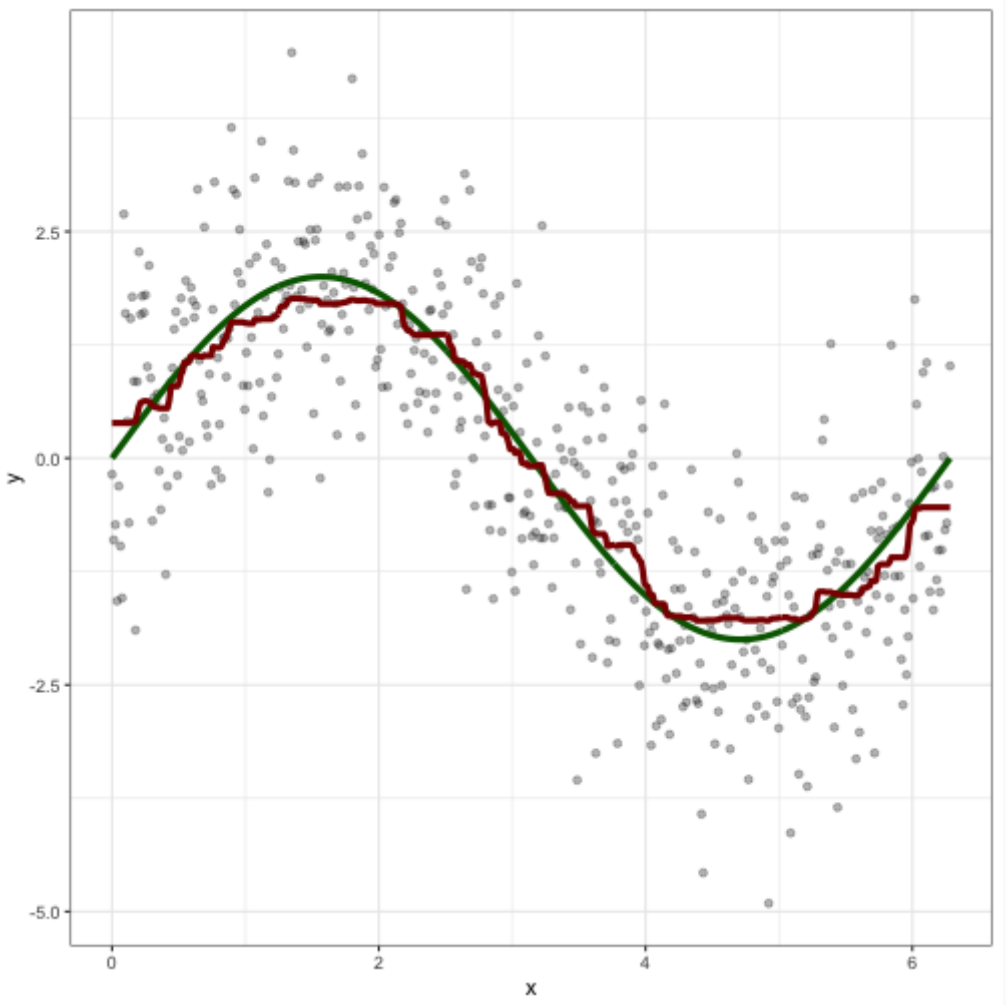
Adding trees to the ensemble

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



Your turn

Experiment with classification (`data = dfc`) to get a grip on the **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

1. 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`
2. Suggestion: set `n.trees = 100` and experiment with `interaction.depth` (e.g., 1, 3 and 5) and `shrinkage` (e.g., 0.01, 0.1 and 1). Or fix the `shrinkage` and let `n.trees` vary.

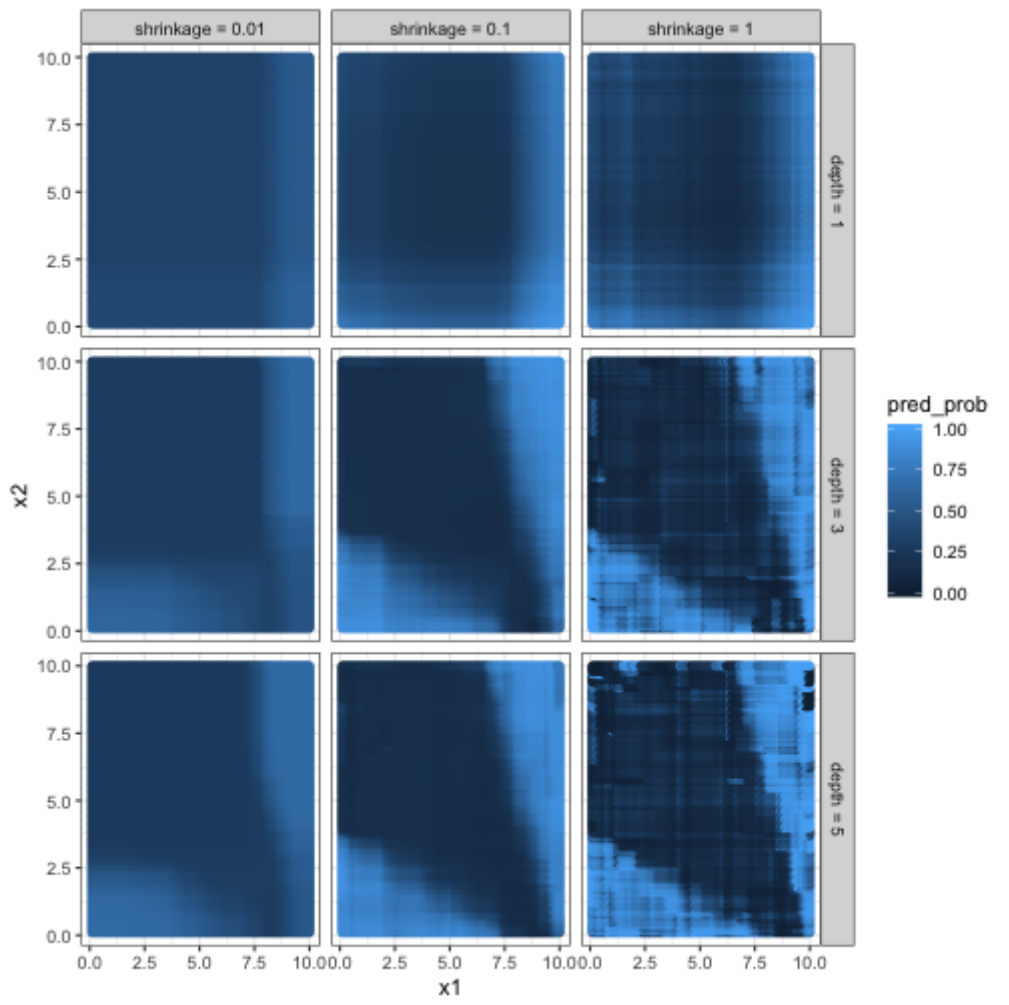
Q1: recoding the response from factor to numeric (note the convenient **pipe assignment** operator `%<>%`)

```
dfc %<>% dplyr::mutate(y_recode = as.integer(y) - 1)
```

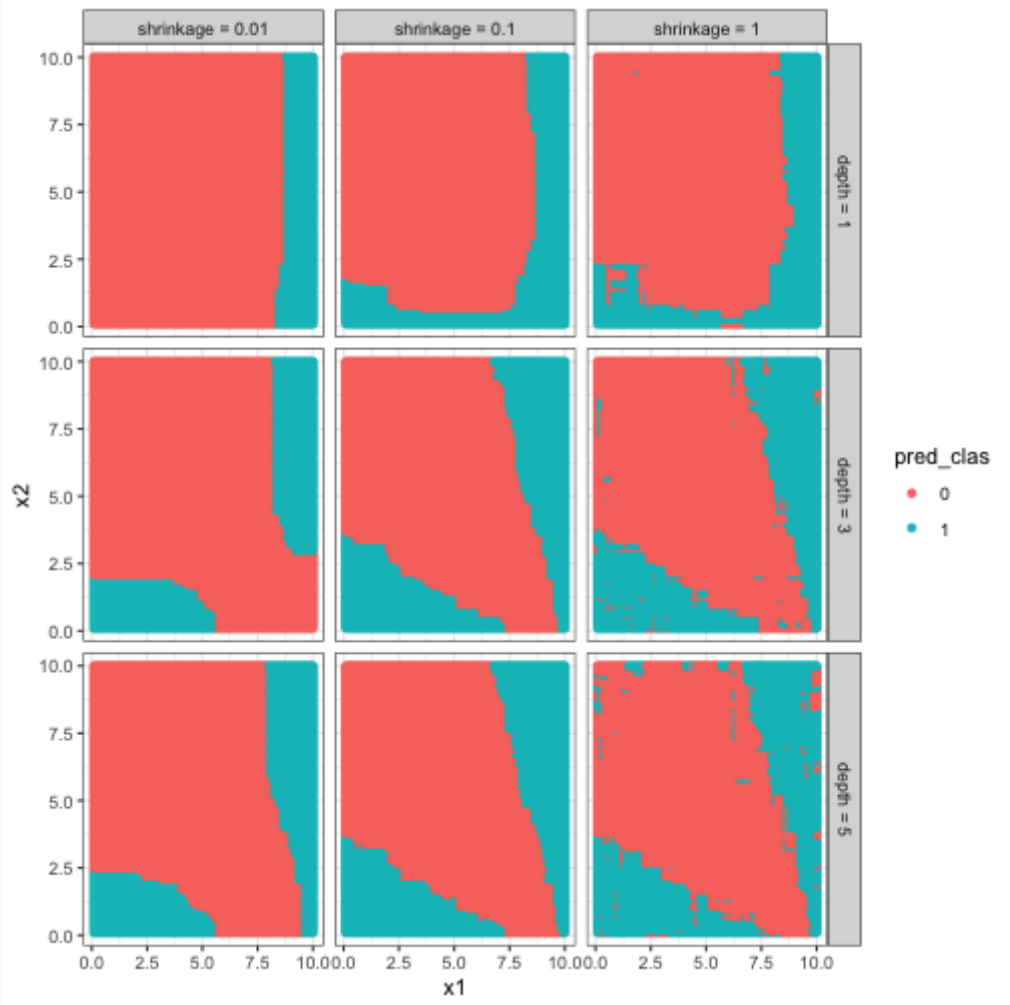
Q2: set up a grid for `interaction.depth` and `shrinkage` and fit a GBM for each combination

```
# 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 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)))
}
```

The predicted **probabilities**

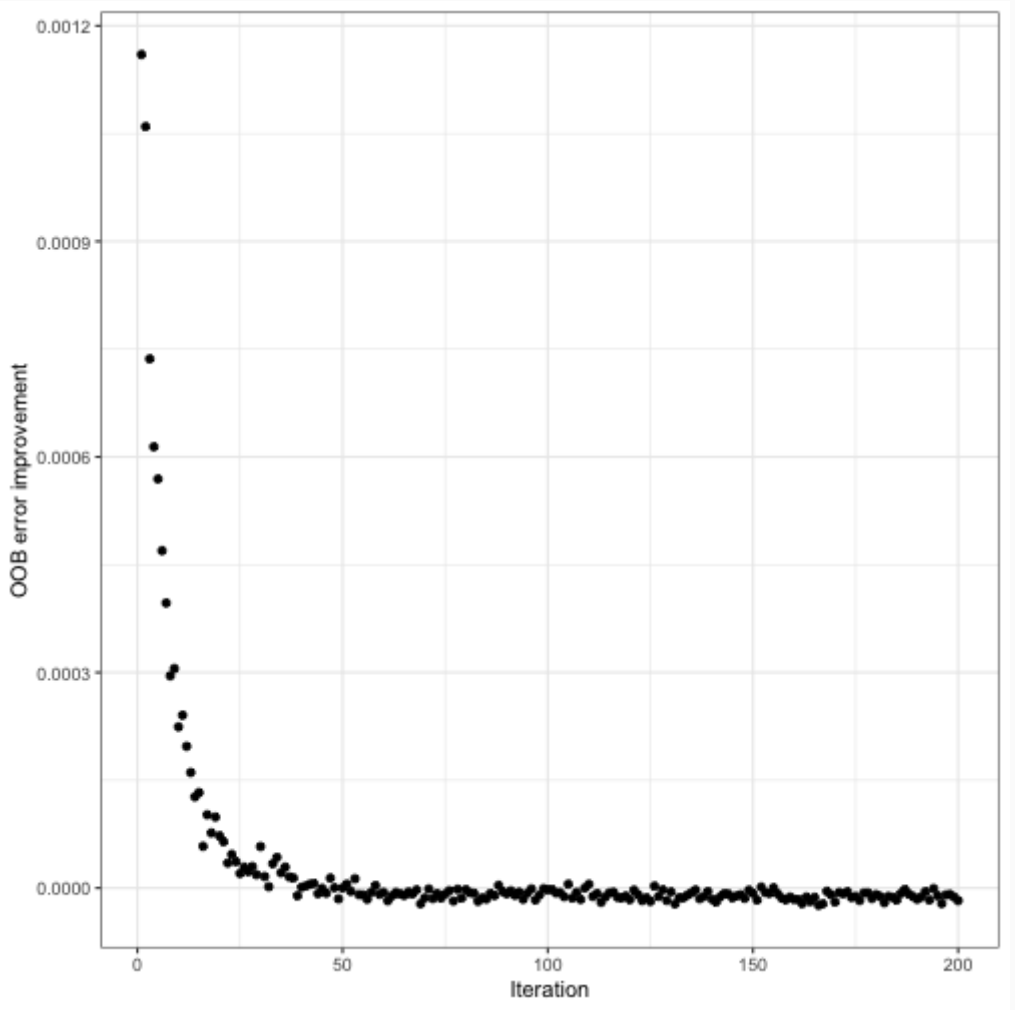


The predicted **classes**



Back to the MTPL data

```
set.seed(76539) # reproducibility
fit <- gbm(formula = nclaims ~
             ageph + agec + bm + power +
             coverage + fuel + sex + fleet + use +
             offset(log(expo)),
           data = mtpl_trn,
           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
          )
# Track the improvement in the OOB error
oob_evo <- fit$oobag.improve
```



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

Inspecting the individual trees

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

```
##   SplitVar SplitCodePred LeftNode RightNode MissingNode ErrorReduction Weight Prediction
## 0        2      6.500000        1         5           9         141.23  97928  -0.004110
## 1        2      1.500000        2         3           4          22.13  76434  -0.018552
## 2       -1     -0.028607       -1        -1          -1           0.00  53138  -0.028607
## 3       -1      0.004383       -1        -1          -1           0.00  23296   0.004383
## 4       -1     -0.018552       -1        -1          -1           0.00  76434  -0.018552
## 5        2      9.500000        6         7           8          10.65  21494   0.047247
## 6       -1      0.033002       -1        -1          -1           0.00   9762   0.033002
## 7       -1      0.059100       -1        -1          -1           0.00  11732   0.059100
## 8       -1      0.047247       -1        -1          -1           0.00  21494   0.047247
## 9       -1     -0.004110       -1        -1          -1           0.00  97928  -0.004110
```

- Does not look that **pretty** right?!
- Even if this was a nicer representation, single trees are **not** going to carry much information
- Luckily we know some **tools** to get a better understanding of the GBM fit!

Your turn

Train a GBM on the **MTPL** data and obtain some meaningful insights from this model.

1. Tune a GBM by tracking the **OOB** or **cross-validation** error. (Remember that `fit$oobag.improve` gives you the improvement in the OOB error.)
2. Use the proper tools to understand your GBM.

- Hint #1: applying the `summary` function on a object of class `gbm` shows built-in feature importance results
- Hint #2: use the following **helper function** for the partial dependence plots:

```
# Need to define this helper function for GBM
pred.fun <- function(object,newdata){
  mean(predict(object, newdata,
               n.trees = object$n.trees))
}
```

```r
# Set up a search grid
tgrid <- expand.grid('depth' = c(1,3,5),
                     'ntrees' = NA,
                     'oob_err' = NA)
```

```r
for(i in seq_len(nrow(tgrid))){
  set.seed(76539) # reproducibility
  # Fit a GBM
  fit <- gbm(formula = nclaims ~
               ageph + agec + bm + power +
               coverage + fuel + sex + fleet + use +
               offset(log(expo)),
             data = mtpl_trn, distribution = 'poisson',
             var.monotone = c(0,0,1,0,0,0,0,0,0),
             n.trees = 1000, shrinkage = 0.01,
             interaction.depth = tgrid$depth[i],
             n.minobsinnode = 1000,
             bag.fraction = 0.75, cv.folds = 0
             )
  # Retrieve the optimal number of trees
  opt <- which.max(cumsum(fit$oobag.improve))
  tgrid$ntrees[i] <- opt
  tgrid$oob_err[i] <- sum(fit$oobag.improve[1:opt])
}
```

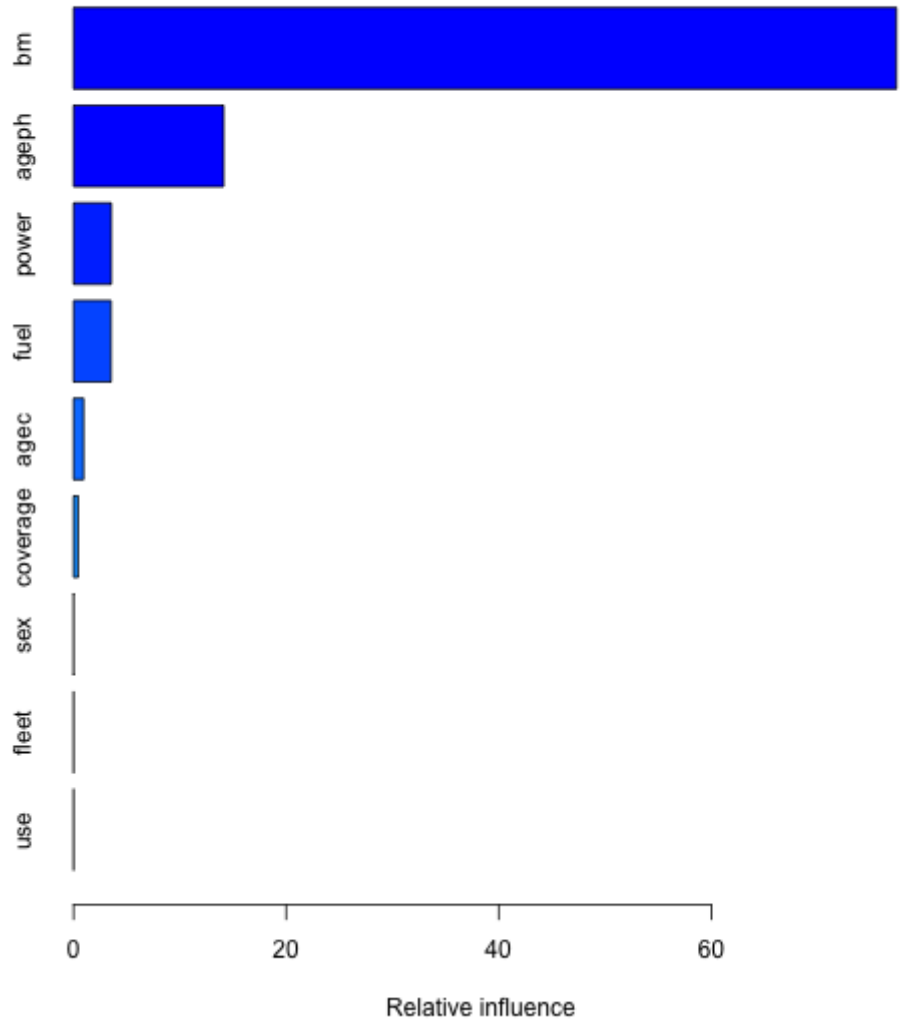
```r
tgrid %>% arrange(oob_err)
```

```
##   depth ntrees     oob_err
## 1     1    702 0.007037245
## 2     3    473 0.007581280
## 3     5    356 0.007723205
```

```r
# Order results on the OOB error
tgrid %<>% arrange(oob_err)
```

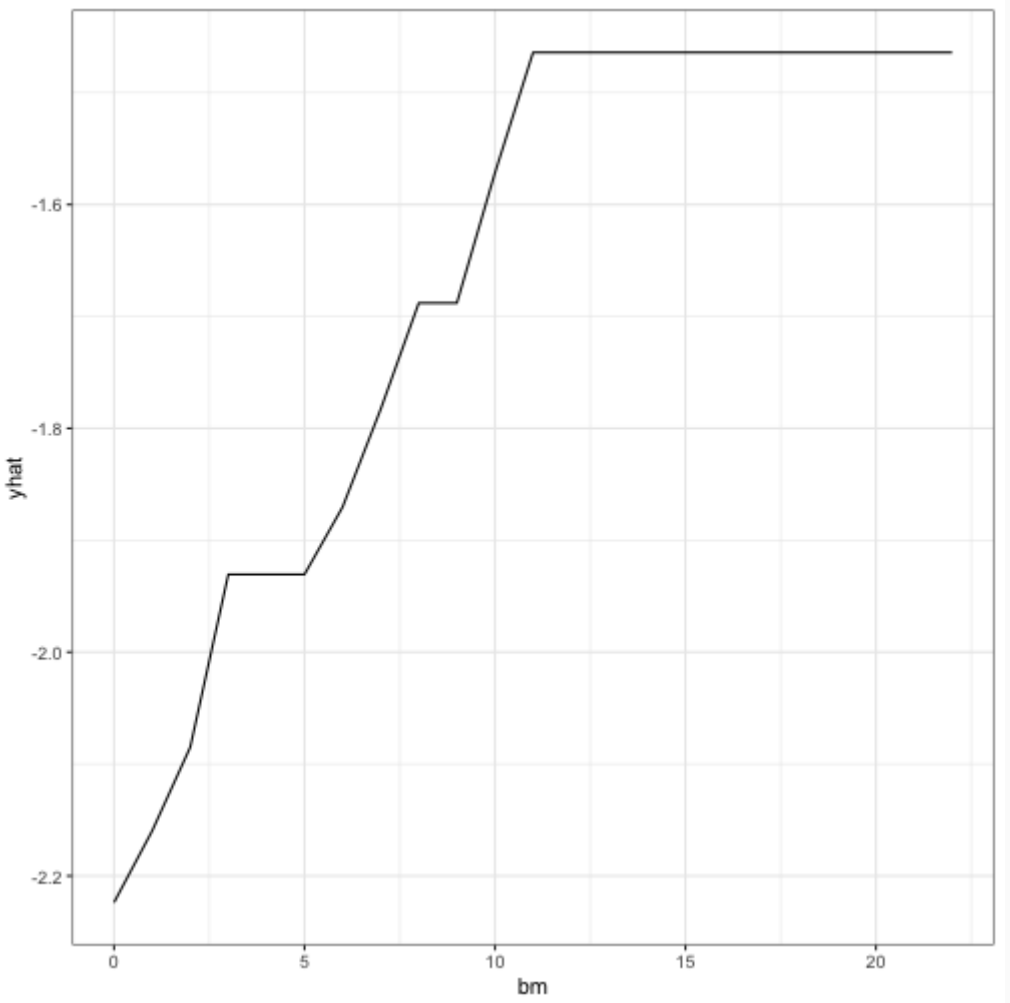
```r
# Fit the optimal GBM
set.seed(76539) # reproducibility
fit_gbm <- gbm(formula = nclaims ~
                 ageph + agec + bm + power +
                 coverage + fuel + sex + fleet + use +
                 offset(log(expo)),
               data = mtpl_trn,
               distribution = 'poisson',
               var.monotone = c(0,0,1,0,0,0,0,0,0),
               n.trees = tgrid$ntrees[1],
               shrinkage = 0.01,
               interaction.depth = tgrid$depth[1],
               n.minobsinnode = 1000,
               bag.fraction = 0.75,
               cv.folds = 0
               )
```

```r
# Get the built-in feature importance
summary(fit_gbm)
```



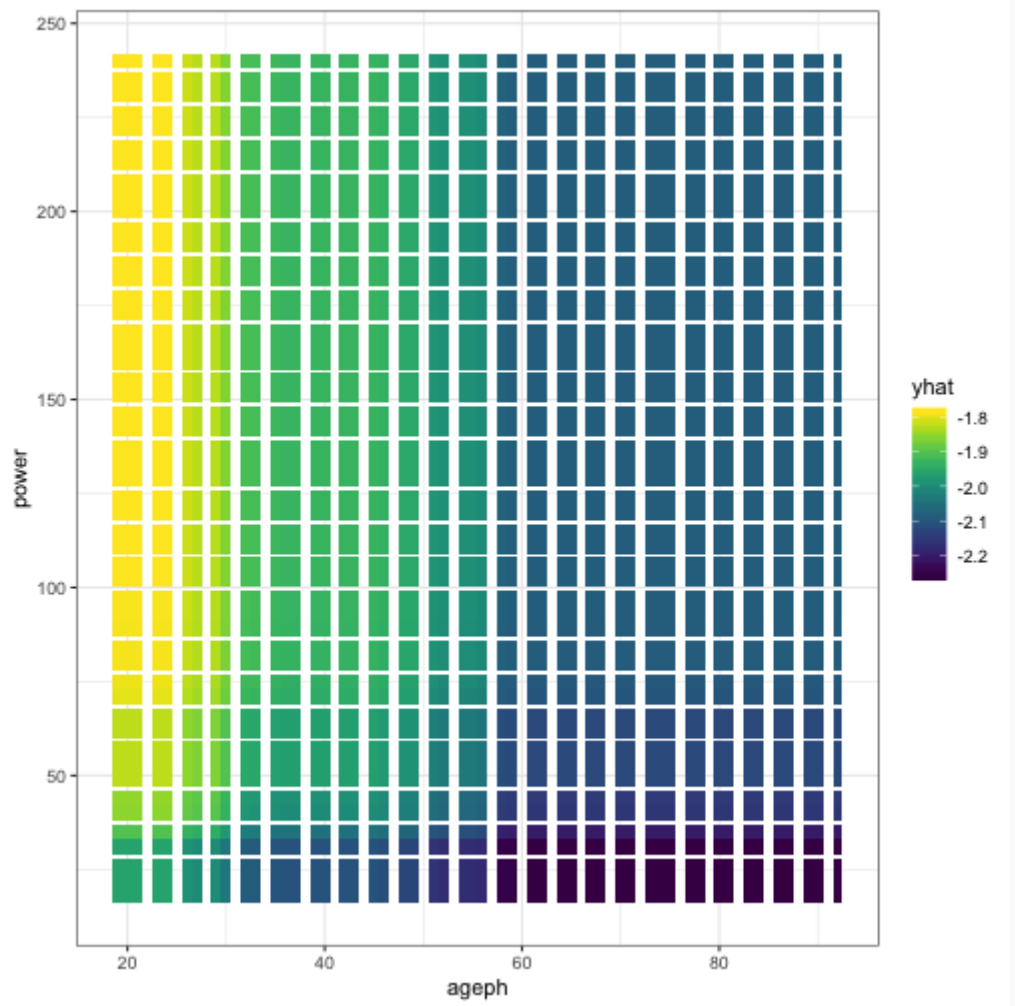
```r
# Need to define this helper function for GBM
pred.fun <- function(object,newdata){
  mean(predict(object, newdata,
               n.trees = object$n.trees))
}
```

```r
# partial: computes the marginal effect
# autoplot: creates the graph using ggplot2
fit_gbm %>%
  partial(pred.var = 'bm',
          pred.fun = pred.fun,
          train = mtpl_trn[pdp_ids,],
          recursive = FALSE) %>%
  autoplot()
```



The **monotonic constraint** holds!

```r
# partial: computes the marginal effect
# autoplot: creates the graph using ggplot2
fit_gbm %>%
  partial(pred.var = c('ageph','power'),
          pred.fun = pred.fun,
          train = mtpl_trn[pdp_ids,],
          recursive = FALSE) %>%
  autoplot()
```



XGBoost

- XGBoost stands for e**X**treme **G**radient **Boost**ing
- 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`: gbtree, gblinear 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 an 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

```
# Features go into the data argument (needs to be converted to a matrix)
# The response and offset are specified via 'label' and 'base_margin' in info respectively
mtpl_xgb <- xgb.DMatrix(data = mtpl_trn %>%
                          select(ageph,power,bm,agec,coverage,fuel,sex,fleet,use) %>%
                          data.matrix,
                        info = list(
                          'label' = mtpl_trn$nclaims,
                          'base_margin' = log(mtpl_trn$expo)))
```

```
# This results in an xgb.DMatrix object
print(mtpl_xgb)
```

```
## xgb.DMatrix  dim: 130571 x 9  info: label base_margin  colnames: yes
```

A simple XGBoost model

Train a model and **save** it for later use

```
set.seed(86493) # reproducibility
fit <- xgboost(
  data = mtpl_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
    )
  )

# Save xgboost model to a file in binary format
xgb.save(fit, fname = 'saved_models/xgb.model')
```

Load the model whenever you need it

```
# Load xgboost model from the binary model file
fit <-  xgb.load('saved_models/xgb.model')
```

Predicting from an XGBoost model

Transform your **test** data to an `xgb.DMatrix`

```
# Supply test data also as an xgb.DMatrix
test_xgb <- mtpl_tst %>%
  select(ageph,power,bm,agec,
         coverage,fuel,sex,fleet,use) %>%
  data.matrix %>%
  xgb.DMatrix

# Possible to add the 'base_margin' afterwards
test_xgb %>% setinfo('base_margin',
                     rep(log(1),
                         nrow(mtpl_tst))
                     )
```

Also add offset to **test** data

Get predictions from your model

```
# Predict from the XGBoost model
preds <- fit %>% predict(
  newdata = test_xgb,
  ntreelimit = fit$niter
  )
```

```
# Average of the predictions
preds %>% mean
```

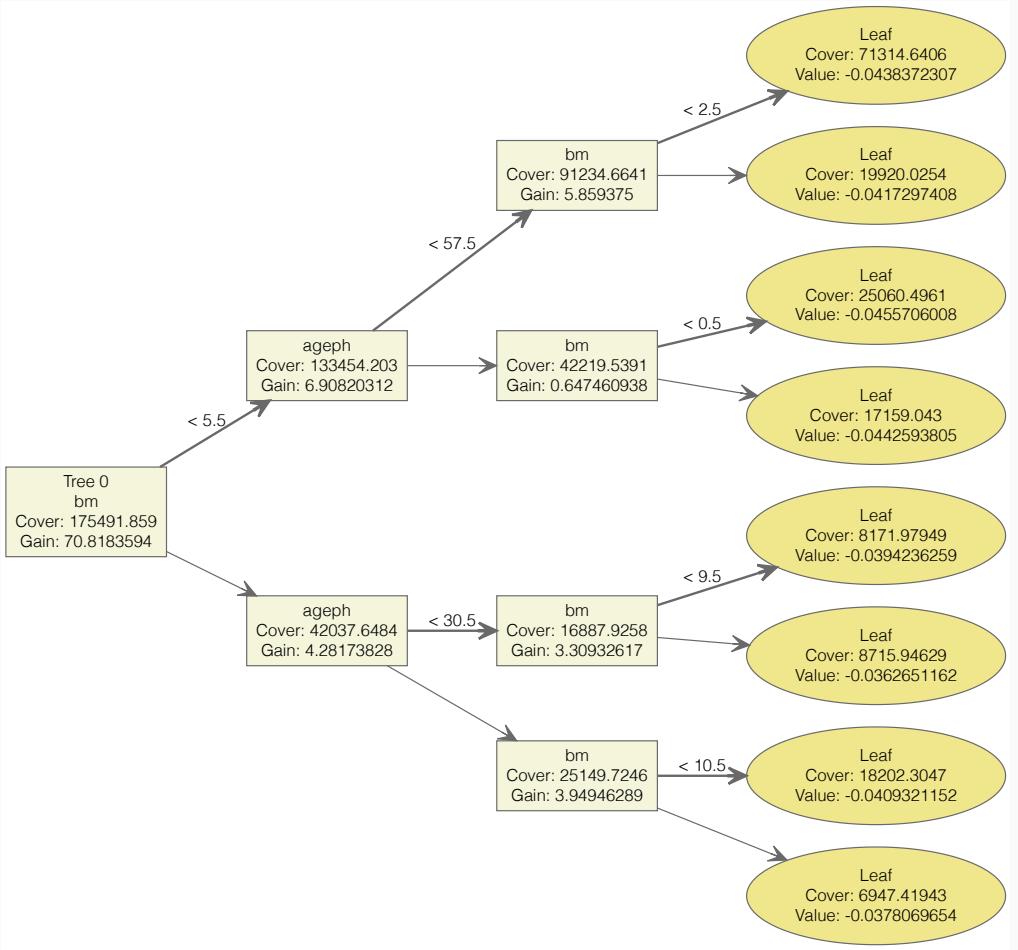
```
## [1] 0.1403883
```

Getting our annual claim **frequency** predictions

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

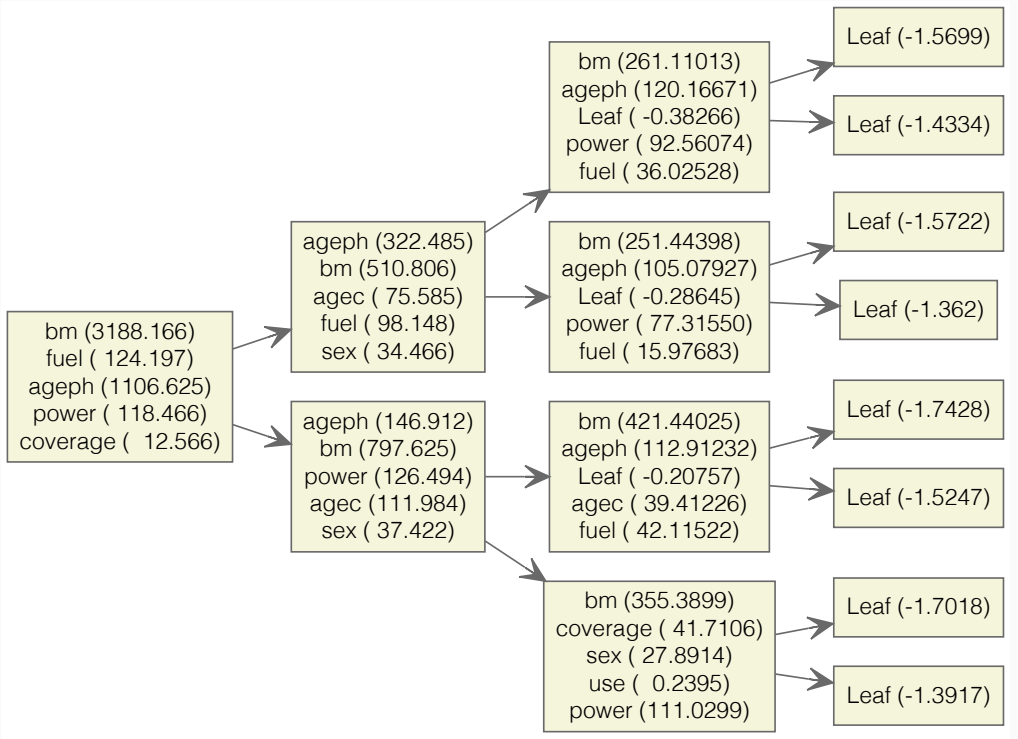
```
xgb.plot.tree(
  feature_names = colnames(mtpl_xgb),
  model = fit,
  trees = 0
  )
```



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)
  )
```

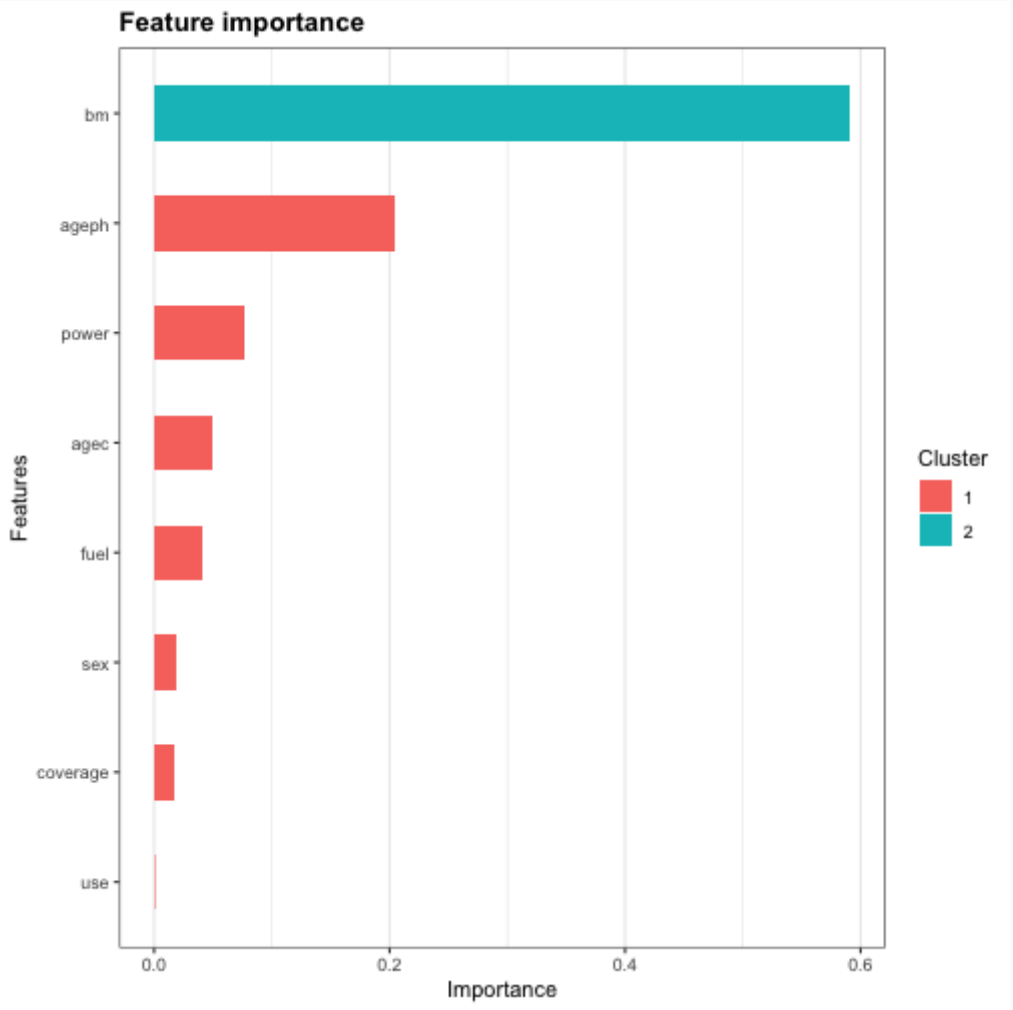


Further built-in interpretations

- 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 = mtpl_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))
```

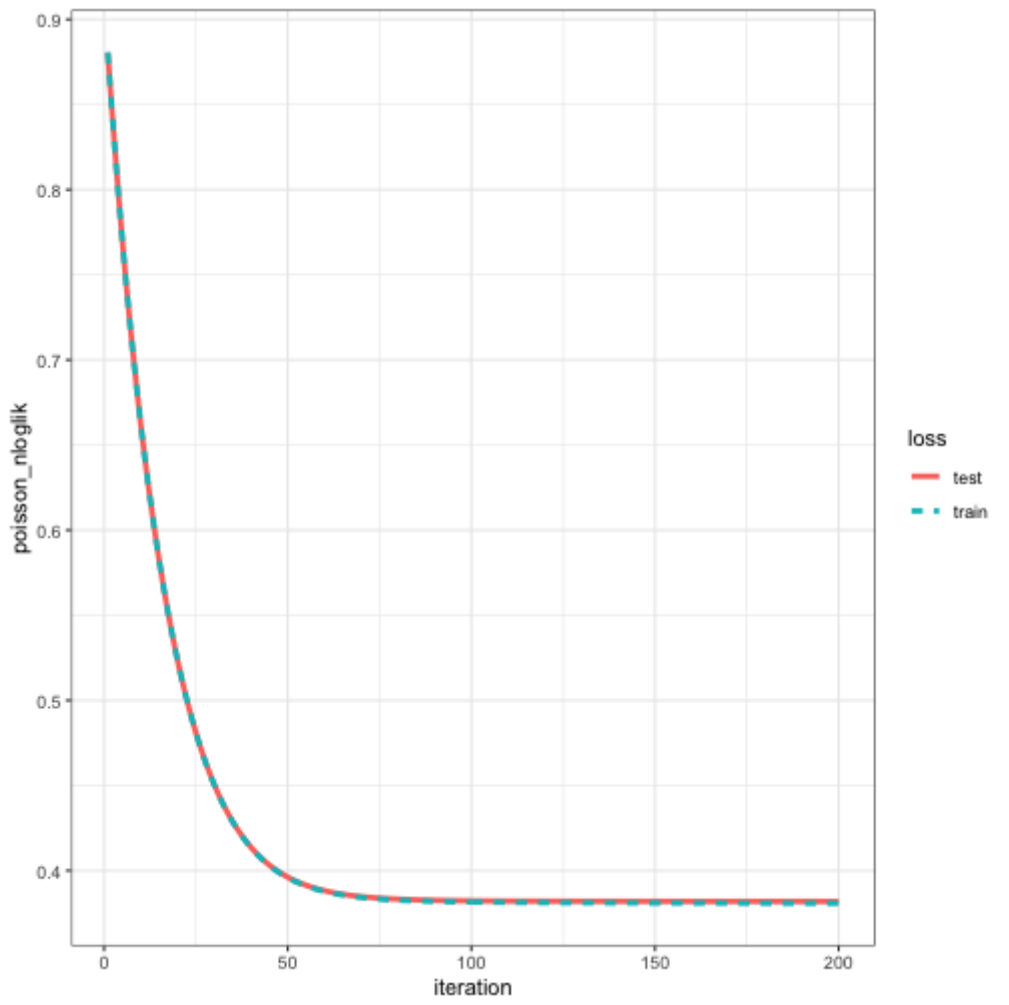
Cross-validation results

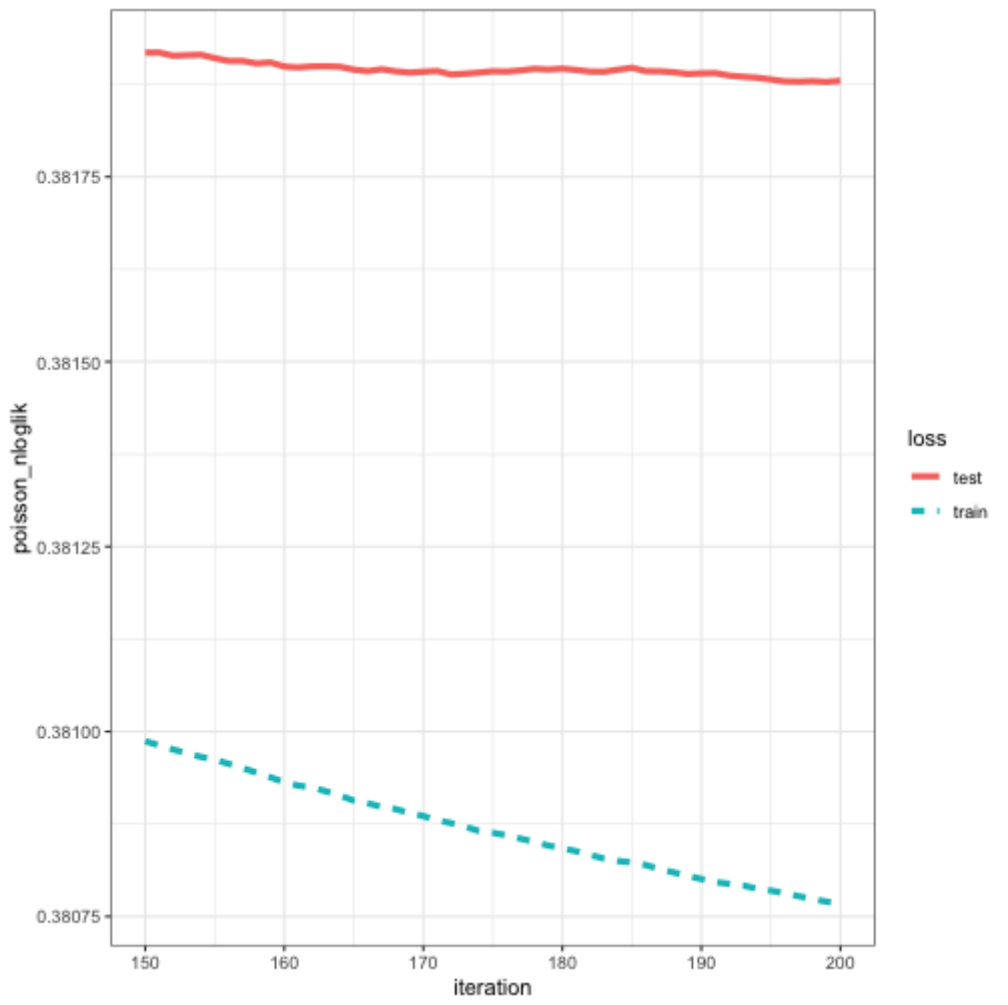
- Get the cross-validation results via `$evaluation_log`:

```
xval$evaluation_log %>% print(digits = 5)
```

```
##      iter train_poisson_nloglik_mean train_poisson_nloglik_std test_poisson_nloglik_mean test_poisson_nloglik_std
##   1:    1                    0.88056                 0.00024334                   0.88052                0.0012427
##   2:    2                    0.84993                 0.00039388                   0.85025                0.0012830
##   3:    3                    0.82151                 0.00038446                   0.82158                0.0012691
##   4:    4                    0.79452                 0.00025874                   0.79462                0.0012479
##   5:    5                    0.76906                 0.00039186                   0.76908                0.0012911
##  ---
## 196:  196                    0.38078                 0.00103761                   0.38188                0.0041933
## 197:  197                    0.38078                 0.00103699                   0.38188                0.0041922
## 198:  198                    0.38077                 0.00103592                   0.38188                0.0041895
## 199:  199                    0.38077                 0.00103708                   0.38188                0.0041893
## 200:  200                    0.38077                 0.00103712                   0.38188                0.0041919
```

Cross-validation results





Your turn

That's a wrap on boosting with **GBM** and **XGBoost**! Now it's your time to experiment.

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

1. Define a **tuning grid** for some parameters that you want to investigate, and tune your boosting model with built-in cross-validation functions. Beware that tuning can take up a lot of time, so do not overdo this.

2. Apply GBM or XGBoost on a classification problem, for example to predict the **occurence** of a claim.

3. Use a **Gamma** deviance to build a severity XGBoost model. The `mtpl` data contains the average claim amount in the feature `average`. Small note: 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 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**?

H2O

H2O

- H2O is a popular open source platform for **machine learning**
- Able to handle big data thanks to **distributed in-memory compression**
- Designed to run in standalone mode, on Hadoop, or within a Spark Cluster
  - easy to test code locally and then **scale up** to a big cluster
- Interfaces for many programming languages
  - **R**, Python, Scala, Java, JSON, and CoffeeScript/JavaScript
  - we will use the {h2o} package in R 
- Includes many common machine learning algorithms:
  - generalized linear models
  - distributed random forest
  - GBM and XGBoost
  - deep learning

Initializing an H2O instance

- Let's **initialize** an H2O cluster with the maximal number of threads available and 4GB of memory:

```
h2o.init(nthreads = -1, max_mem_size = '4g')
```

- You can check the cluster **info** and **status** at any time via `h2o.clusterInfo` and `h2o.clusterStatus`

```
## R is connected to the H2O cluster:
##     H2O cluster uptime:         3 seconds 300 milliseconds
##     H2O cluster timezone:       Europe/Brussels
##     H2O data parsing timezone:  UTC
##     H2O cluster version:        3.28.0.2
##     H2O cluster version age:    20 days
##     H2O cluster name:           H2O_started_from_R_Roel_vlt494
##     H2O cluster total nodes:    1
##     H2O cluster total memory:   4.00 GB
##     H2O cluster total cores:    4
##     H2O cluster allowed cores:  4
##     H2O cluster healthy:        TRUE
##     H2O Connection ip:          localhost
##     H2O Connection port:        54321
...
```

Getting your data in H2O

- Local data can be stored in the **H2O cluster** via `h2o.uploadFile`:

```
# Define the local path
mtpl_path <- paste0(data_path,'MTPL.csv')
# Upload the data to the cluster
mtpl_h2o <- h2o.uploadFile(path = mtpl_path,
                           destination_frame = 'mtpl.h2o')
```

- Your data is now an H2O Frame:

```
mtpl_h2o %>% class
```

```
## [1] "H2OFrame"
```

- Adding a **new column** to an H2O Frame:

```
mtpl_h2o[,'log_expo'] <- log(mtpl_h2o[,'expo'])
```

Transferring data between R and H2O

H2O → **R** is possible via `as.data.frame`:

```
mtpl_r <- mtpl_h2o %>% as.data.frame
```

```
mtpl_r %>% class
```

```
## [1] "data.frame"
```

R → **H2O** is possible via `as.h2o`:

```
mtpl2_h2o <- mtpl_r %>% as.h2o(
  destination_frame = 'mtpl2.h2o')
```

```
mtpl_h2o %>% class
```

```
## [1] "H2OFrame"
```

Let's **remove** this copy:

```
# List the objects in the cluster
h2o.ls()
```

```
##                key
## 1 RTMP_sid_9f15_2
## 2         mtpl.h2o
## 3        mtpl2.h2o
```

```
# Remove the object from the H2O cluster
'mtpl2.h2o' %>% h2o.rm
# Remove the R object
mtpl2_h2o %>% remove
# List the objects in the cluster
h2o.ls()
```

```
##                key
## 1 RTMP_sid_9f15_2
## 2         mtpl.h2o
```

Base R functions in H2O

- Some can be directly applied to H2OFrames, possibly with different **default** setting:

```
mtpl_h2o[,'ageph'] %>% quantile # on H2O object
```

```
##  0.1%    1%   10%   25% 33.3%   50% 66.7%   75%   90%   99% 99.9%
##    20    22    28    35    39    46    53    58    68    80    87
```

```
mtpl_r[,'ageph'] %>% quantile # on R object
```

```
##   0%  25%  50%  75% 100%
##   18   35   46   58   95
```

- Others do not work and have an **H2O variant**:

```
mtpl_h2o[,'fuel'] %>% h2o.table
```

```
##       fuel  Count
## 1   diesel  50394
## 2 gasoline 112818
...
```

```
mtpl_h2o[,'fuel'] %>% table
#Error:
#unique.default(x, nmax = nmax):
#invalid type/length (environment/0)
```

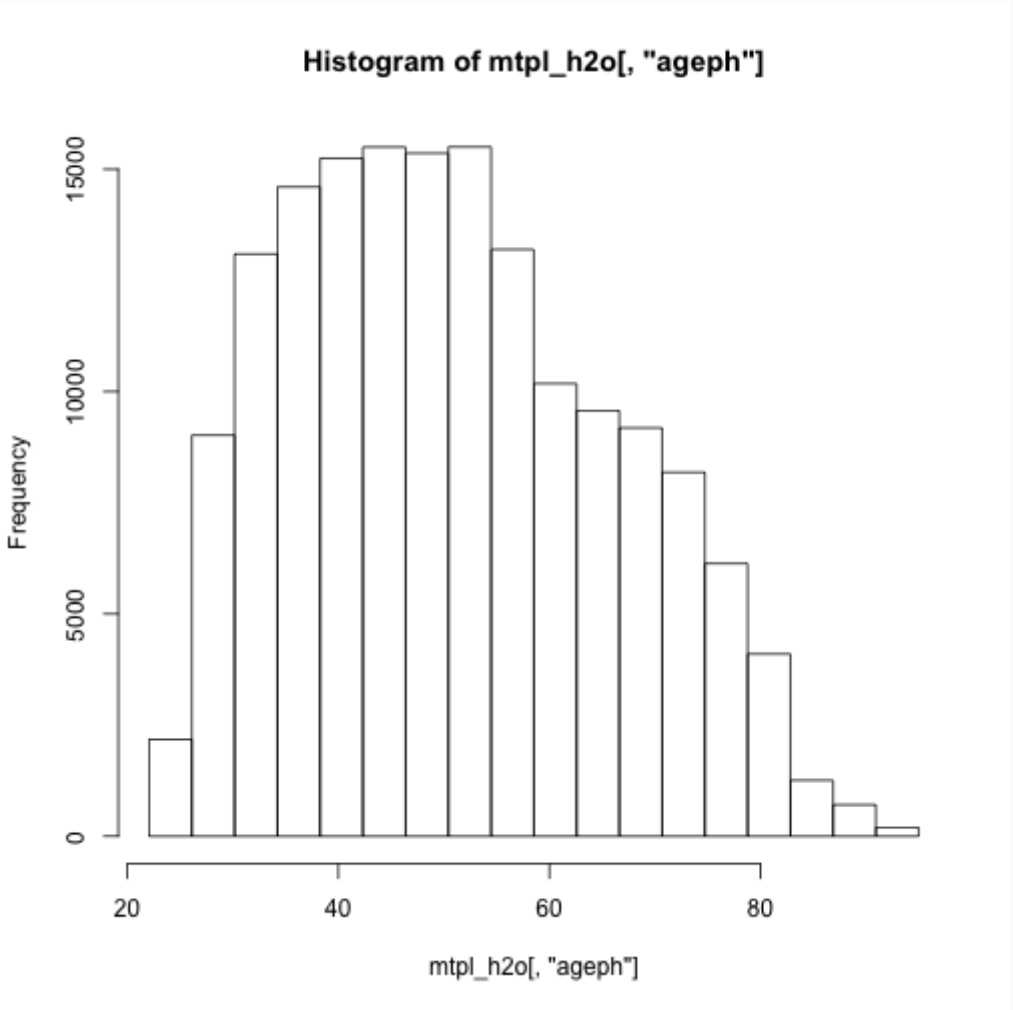
Data munging in H2O

- Specific functions for data **munging** and **exploration**:

```
mtpl_h2o %>% h2o.group_by(by = 'ageph',
                          sum('expo'))
```

```
##   ageph    sum_expo
## 1    18    4.621918
## 2    19   93.021918
## 3    20  342.284932
## 4    21  596.389041
## 5    22  778.827397
## 6    23 1165.358904
##
## [78 rows x 2 columns]
```

```
mtpl_h2o[,'ageph'] %>% h2o.hist
```



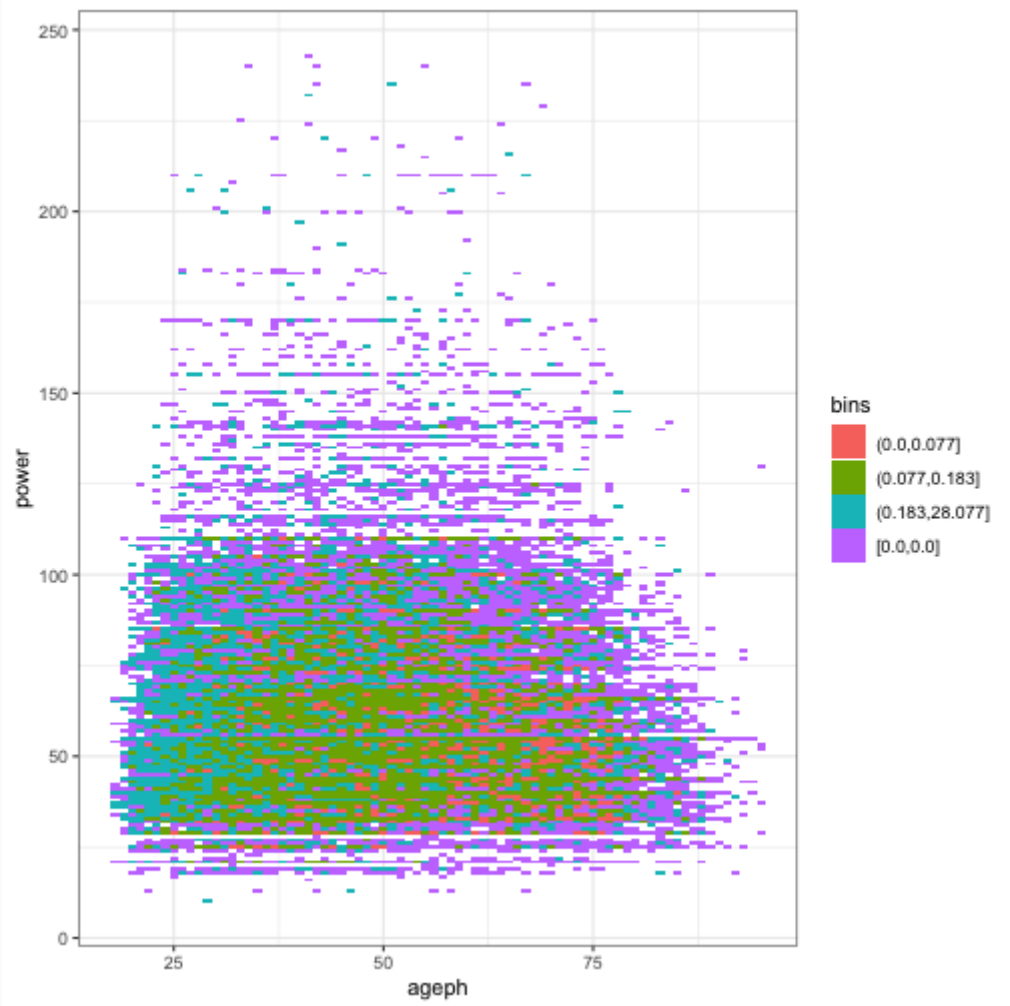
Your turn

We already saw a couple of times that **young policyholders** driving a **high powered car** are predicted as a high risk in the MTPL data.
Verify this on the raw data by some data wrangling in H2O.

1. Use `h2o.group_by` on `mtpl_h2o` to calculate the sum of `nclaims` by `ageph` and `power`
2. Do the same for `expo`
3. Now merge these two H2OFrames with `h2o.merge`
4. Add the empirical frequency to your H2OFrame by dividing the sum of claims and exposures
5. Group the empirical frequency in 4 equally sized groups via `h2o.cut` and `quantiles`
6. Transfer your H2OFrame to an R data.frame via `as.data.frame`
7. Plot this data.frame via `ggplot` and `geom_tile` (hint: aesthetic `fill` might be useful)

```
by_vars <- c('ageph','power')
nclaim_by <- mtpl_h2o %>% h2o.group_by(by = by_vars,
                                       sum('nclaims'))
expo_by <- mtpl_h2o %>% h2o.group_by(by = by_vars,
                                     sum('expo'))

by_h2o <- h2o.merge(nclaim_by,expo_by)

by_h2o[,'freq'] <- {by_h2o[,'sum_nclaims']/
    by_h2o[,'sum_expo']}

by_h2o[,'bins'] <- by_h2o[,'freq'] %>%
  h2o.cut(breaks = quantile(by_h2o[,'freq'],
                            probs = seq(0,1,0.25)),
          include.lowest = TRUE)

by_r <- by_h2o %>% as.data.frame
```

```
by_r %>% ggplot(aes(x = ageph, y = power)) +
  geom_tile(aes(fill = bins))
```



Data splitting in H2O

Splitting data in **train** and **test** sets can be performed via `h2o.splitFrame`:

```
mtpl_split <- mtpl_h2o %>%  h2o.splitFrame(ratios = 0.7,
                                           destination_frames = c('train.h2o',
                                                                  'test.h2o'),
                                           seed = 54321)
```

Retrieve the training data from the **1st** list element and the test data from the **2nd** one:

```
train_h2o <- mtpl_split[[1]]
train_h2o %>% dim
```

```
## [1] 114327     19
```

```
test_h2o <- mtpl_split[[2]]
test_h2o %>%  dim
```

```
## [1] 48885    19
```

Model building in H2O

- H2O supports a **diverse** set of algorithms, also those discussed so far:
  - `h2o.glm`, `h2o.randomForest`, `h2o.gbm`, `h2o.xgboost`
  - sidenote: Poisson loss function is **not** available for the random forest
- The implementations are very flexible, resulting in **many** tuning parameters
  - full **Cartesian** grid searches are not always computationally feasible
  - H2O therefore also implements **random** grid searches with early stopping measures

- Cartesian grid search:
  - evaluate **each** possible parameter combination
  - 😊 find optimal combination
  - 😢 can be very time consuming

- Random grid search:
  - **random** search over the possible combinations
  - 😊 much faster
  - 😢 maybe not finding the optimal combination

Random grid search in H2O

Define a search grid containing your tuning parameters as a **list**:

```
gbm_grid <- list(
  max_depth = c(1,3,5), # depth of a tree
  sample_rate = c(0.5,0.75,1), # row sample rate
  col_sample_rate = c(0.5,0.75,1) # column sample rate
  )
```

Define some settings for the grid search as a **list**:

```
gbm_settings <- list(
  strategy = 'RandomDiscrete', # set to 'Cartesian' for full Cartesian grid search
  stopping_metric = 'deviance',
  stopping_tolerance = 0.001,
  stopping_rounds = 10,
  max_runtime_secs = 10*60
  )
```

- Note that we implement two early stopping measures:
  - maximum runtime of 5 minutes **+** stop if the **deviance improvement** did not exceed 0.001 in 10 consecutive rounds
  - can also specify a maximum number of combinations to try out via `max_models`

Performing the random grid search

Perform the grid search

```
gbm_search <- h2o.grid(
  algorithm = 'gbm',
  distribution = 'poisson',
  x = names(train_h2o)[7:15],
  y = 'nclaims',
  offset_column = 'log_expo',
  training_frame = train_h2o,
  hyper_params = gbm_grid, # our defined grid
  grid_id = 'gbm_grid', # used to get results
  ntrees = 500,
  learn_rate = 0.05,
  learn_rate_annealing = 0.99, # decr. step size
  search_criteria = gbm_settings, # settings
  nfolds = 5,
  seed = 54321
)
```

Retrieve the results

```
gbm_perf <- h2o.getGrid(
  grid_id = 'gbm_grid',
  sort_by = 'residual_deviance',
  decreasing = FALSE
  )
print(gbm_perf)
```

(shown on next slide)

Grid search results

```
## H2O Grid Details
## ================
##
## Grid ID: gbm_grid
## Used hyper parameters:
##   -  col_sample_rate
##   -  max_depth
##   -  sample_rate
## Number of models: 7
## Number of failed models: 0
##
## Hyper-Parameter Search Summary: ordered by increasing residual_deviance
##   col_sample_rate max_depth sample_rate        model_ids  residual_deviance
## 1            0.75         3        0.75 gbm_grid_model_1 0.7482509206438585
## 2             1.0         5         1.0 gbm_grid_model_5 0.7497538567606931
## 3             0.5         3         1.0 gbm_grid_model_6 0.7501828776706148
## 4            0.75         1         1.0 gbm_grid_model_2 0.7503135066816209
## 5            0.75         5        0.75 gbm_grid_model_4 0.7521096138665183
## 6             0.5         5         0.5 gbm_grid_model_3 0.7531107077788328
## 7            0.75         5         1.0 gbm_grid_model_7 0.7610825837056191
```

Get the optimal model and check performance

```
# Retrieve the best performing model
gbm_h2o <- h2o.getModel(gbm_perf@model_ids[[1]])
print(gbm_h2o@model[['model_summary']])
```

```
## Model Summary:
##   number_of_trees number_of_internal_trees model_size_in_bytes min_depth
## 1             500                      500               79102         3
##   max_depth mean_depth min_leaves max_leaves mean_leaves
## 1         3    3.00000          7          8     7.87600
```

```
# Check the performance on the test set
gbm_h2o %>% h2o.performance(newdata = test_h2o)
```

```
## H2ORegressionMetrics: gbm
##
## MSE:  0.1305007
## RMSE:  0.3612488
## MAE:  0.2149038
## RMSLE:  0.235823
## Mean Residual Deviance :  0.7448572
## R^2 :  0.01958949
```

Do the predictions make sense?

```
# Get predictions on the test set
test_pred <- gbm_h2o %>%
  h2o.predict(newdata = test_h2o)
```

```
# Group observations by number of claims
# and calculate the mean prediction
h2o.cbind(test_h2o,test_pred) %>%
  h2o.group_by(by = 'nclaims',
               mean('predict'))
```

```
##   nclaims mean_predict
## 1       0    0.1212257
## 2       1    0.1437905
## 3       2    0.1603233
## 4       3    0.1764597
## 5       4    0.1811591
## 6       5    0.2058805
##
## [6 rows x 2 columns]
```

```
gbm_h2o %>% h2o.varimp_plot
```

Your turn

H2O also offers an implementation of GLMs with regularization.
Use your GLM skills to build a **claim frequency model** on the MTPL data.

- Use `?h2o.glm` to investigate the parameters that you can specify
- Many will be similar to the GBM case:
  - `training_frame`, `x`, `y` and `offset_column`
- The following ones are interesting for the **model structure**:
  - `family`, `link`, `intercept`, `interactions` and `interaction_pairs`
- The following ones are interesting for **regularization**:
  - `alpha`, `lambda`, `lambda_search`, `nlambdas` and `early_stopping`

1. Fit a GLM with or without **regularization**
2. Inspect the **coefficients** via `@model[['coefficients_table']]`
3. How do the GLM and GBM compare against each other?

Q1: fitting a regularized GLM

```
glm_h2o <- h2o.glm(
  training_frame = train_h2o,
  x = names(mtpl_h2o)[7:15],
  y = 'nclaims',
  offset_column = 'log_expo',
  family = 'poisson',
  link = 'Log',
  intercept = TRUE,
  interaction_pairs = list(
    c('ageph','power')
  ),
  alpha = 0.5,
  lambda_search = TRUE,
  nlambdas = 100,
  early_stopping = TRUE
  )
```

Q2: inspecting the fitted coefficients

```
# Print the coeficients
print(glm_h2o@model[['coefficients_table']])
```

```
## Coefficients: glm coefficients
##             names coefficients standardized_coefficients
## 1       Intercept    -2.026695                 -2.001021
## 2    coverage.TPL     0.027196                  0.027196
## 3   coverage.TPL+    -0.004854                 -0.004854
## 4  coverage.TPL++     0.000000                  0.000000
## 5      sex.female     0.000000                  0.000000
## 6        sex.male     0.000000                  0.000000
## 7     fuel.diesel     0.055608                  0.055608
## 8   fuel.gasoline    -0.059904                 -0.059904
## 9     use.private     0.000000                  0.000000
## 10       use.work     0.000000                  0.000000
## 11    ageph_power     0.000000                  0.000000
## 12          ageph    -0.007430                 -0.110259
## 13             bm     0.059413                  0.237212
## 14          power     0.003271                  0.062154
## 15           agec     0.000000                  0.000000
## 16          fleet    -0.063509                 -0.011150
```

Q3: comparing the GLM and GBM

```
glm_h2o %>% h2o.performance(newdata = test_h2o)
```

```
## H2ORegressionMetrics: glm
##
## MSE:  0.1305983
## RMSE:  0.3613839
## MAE:  0.2153676
## RMSLE:  0.2359556
## Mean Residual Deviance :  0.5304612
## R^2 :  0.01885604
## Null Deviance :26798.92
## Null D.o.F. :48884
## Residual Deviance :25931.59
## Residual D.o.F. :48876
## AIC :37277.56
```

```
gbm_h2o %>% h2o.performance(newdata = test_h2o)
```

```
## H2ORegressionMetrics: gbm
##
## MSE:  0.1305007
## RMSE:  0.3612488
## MAE:  0.2149038
## RMSLE:  0.235823
## Mean Residual Deviance :  0.7448572
## R^2 :  0.01958949
```

Q3: comparing the GLM and GBM

```
test_pred <- glm_h2o %>%
  h2o.predict(newdata = test_h2o)
```

```
h2o.cbind(test_h2o,test_pred) %>%
  h2o.group_by(by = 'nclaims', mean('predict'))
```

```
##   nclaims mean_predict
## 1       0    0.1216766
## 2       1    0.1432537
## 3       2    0.1592038
## 4       3    0.1787354
## 5       4    0.1843094
## 6       5    0.1770974
##
## [6 rows x 2 columns]
```

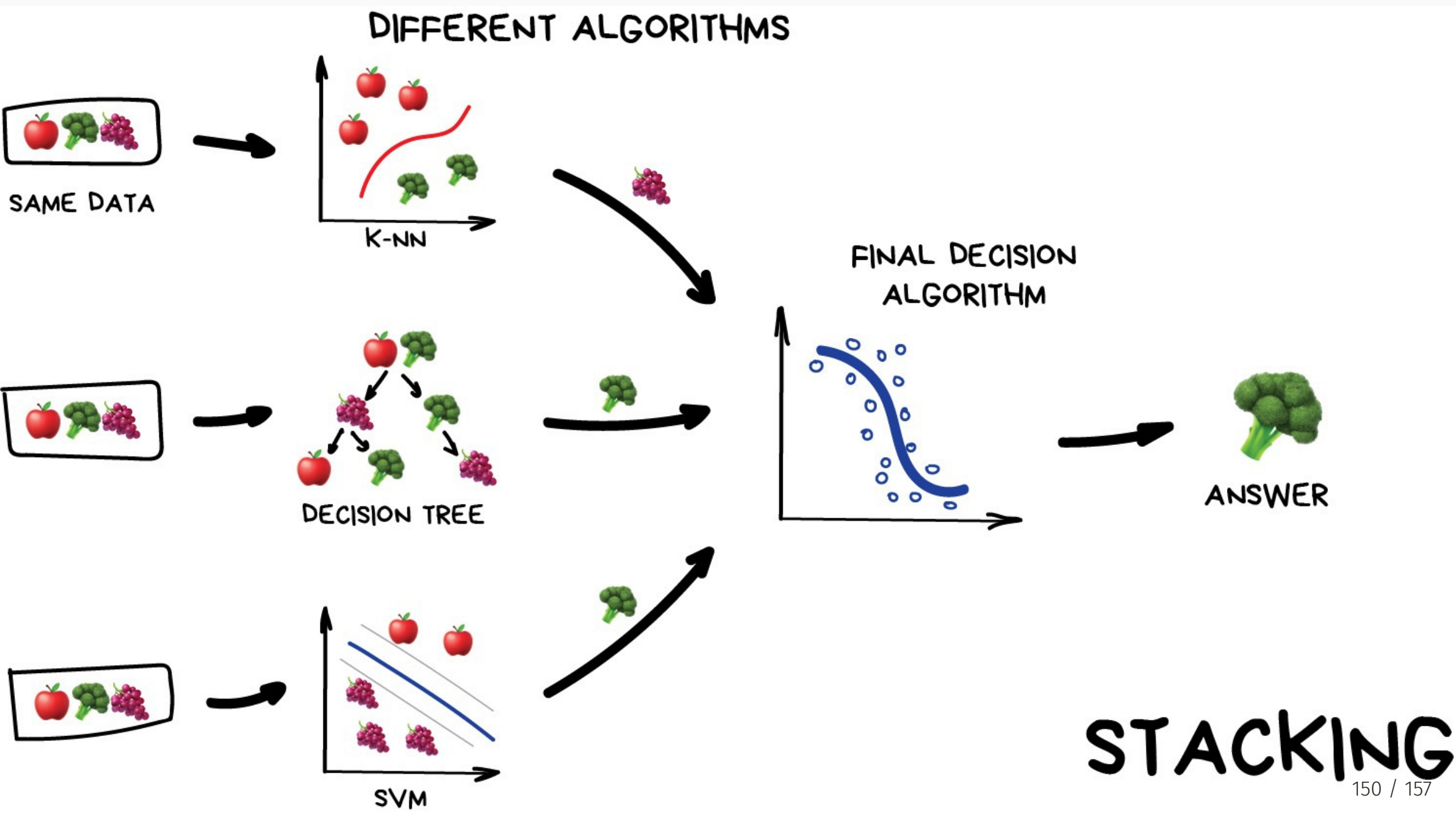
```
test_pred <- gbm_h2o %>%
  h2o.predict(newdata = test_h2o)
```

```
h2o.cbind(test_h2o,test_pred) %>%
  h2o.group_by(by = 'nclaims', mean('predict'))
```

```
##   nclaims mean_predict
## 1       0    0.1212257
## 2       1    0.1437905
## 3       2    0.1603233
## 4       3    0.1764597
## 5       4    0.1811591
## 6       5    0.2058805
##
## [6 rows x 2 columns]
```

We now have two models, a GLM and a GBM, so which one do we **choose**?

Do we have to choose?

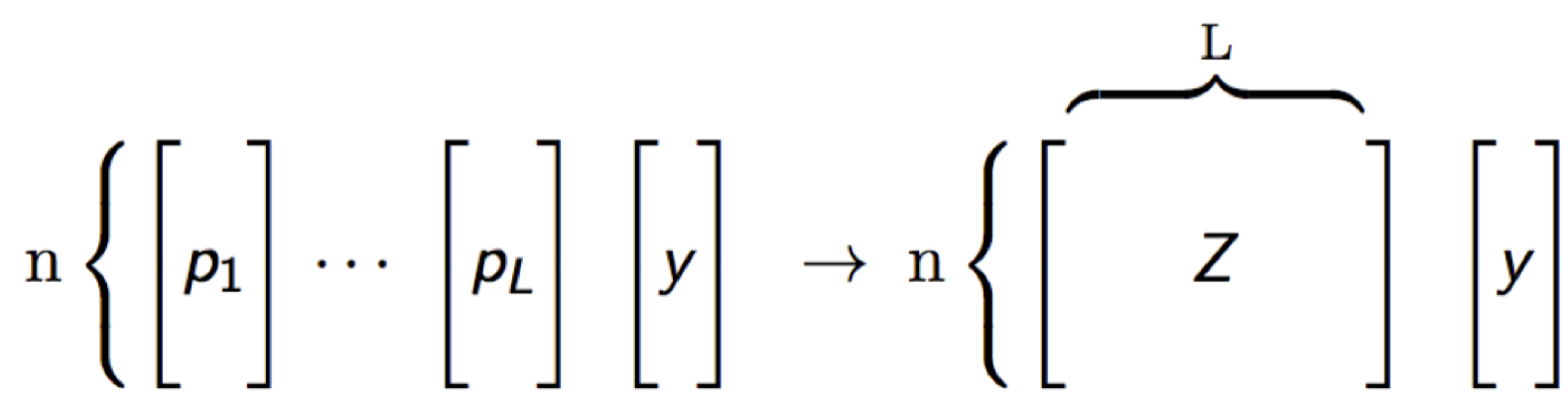

DIFFERENT ALGORITHMS
SAME DATA
K-NN
DECISION TREE
SVM
FINAL DECISION ALGORITHM
ANSWER
STACKING

Stacking

- Stacking (or stacked generalization) takes **ensembling** to a whole new level
- Training a new learning algorithm to combine the predictions of several **base learners**
- Sounds familiar?!
  - basically what a random forest or GBM do with trees as base learners
- Stacking uses random forests, GBMs and other models as base learners
- A meta algorithm, called the super learner, combines this diverse group of strong learners
- A stacked ensemble performs **at least as good** as the best base learner
  - this holds on the training data
  - not necessarily also true on the test data
  - 📢 beware of **overfitting**

How to stack?

- Stacking uses internal **k-fold cross-validation** to generate the so-called level-one data



- This puts some requirements on the training of the **individual** base learners:
  - trained on the same training set
  - trained with the same number of CV folds
  - same fold assignment to ensure the same observations are used
  - cross-validated predictions from all of the models must be preserved

Buiding a GLM and GBM

Train and **cross-validate** a GLM

```
stack_glm <- h2o.glm(
  training_frame = train_h2o,
  x = names(mtpl_h2o)[7:15],
  y = 'nclaims',
  offset_column = 'log_expo',
  family = 'poisson',
  link = 'Log',
  intercept = TRUE,
  alpha = 0.5,
  lambda_search = TRUE,
  nlambdas = 100,
  early_stopping = TRUE,
  nfolds = 5,
  fold_assignment = 'Modulo',
  keep_cross_validation_predictions = TRUE,
  seed = 987654321
)
```

Train and **cross-validate** a GBM

```
stack_gbm <- h2o.gbm(
  training_frame = train_h2o,
  x = names(mtpl_h2o)[7:15],
  y = 'nclaims',
  offset_column = 'log_expo',
  distribution = 'poisson',
  ntrees = 500,
  learn_rate = 0.05,
  learn_rate_annealing = 0.99,
  max_depth = 3,
  sample_rate = 0.75,
  col_sample_rate = 0.5,
  nfolds = 5,
  fold_assignment = 'Modulo',
  keep_cross_validation_predictions = TRUE,
  seed = 987654321
)
```

Stack the GLM and GBM

- **Combine** the GLM and GBM predictions with a super learner
- The options for the super learner **algorithm** are:
    - `glm`, `gbm`, `drf` and `deeplearning`

```
stack_ens <- h2o.stackedEnsemble(
  training_frame = train_h2o,
  x = names(mtpl_h2o)[7:15],
  y = 'nclaims',
  model_id = 'my_stack',
  base_models = list(
    stack_glm,
    stack_gbm
    ),
  metalearner_algorithm = 'deeplearning',
  metalearner_params = list(
    distribution = 'poisson'
  ),
  seed = 123456789
)
```

Let's compare

Predictions on the **training** data

```
preds_train <- h2o.cbind(
  h2o.predict(stack_glm, newdata = train_h2o),
  h2o.predict(stack_gbm, newdata = train_h2o),
  h2o.predict(stack_ens, newdata = train_h2o)
  )
names(preds_train) <- c('glm','gbm','stack')
```

Predictions on the **test** data

```
preds_test <- h2o.cbind(
  h2o.predict(stack_glm, newdata = test_h2o),
  h2o.predict(stack_gbm, newdata = test_h2o),
  h2o.predict(stack_ens, newdata = test_h2o)
  )
names(preds_test) <- c('glm','gbm','stack')
```

Poisson deviance on the **training** data

```
dev_poiss(y = as.vector(train_h2o[,'nclaims']),
          yhat = as.matrix(preds_train))
```

```
##       glm       gbm     stack
## 0.5370740 0.5331996 0.5334241
```

Poisson deviance on the **test** data

```
dev_poiss(y = as.vector(test_h2o[,'nclaims']),
          yhat = as.matrix(preds_test))
```

```
##       glm       gbm     stack
## 0.5304067 0.5296093 0.5300134
```

Note that tuning of the **super learner** is adviced!

Other forms of stacking

- Stacking existing models
  - train some **different types** of base learners and combine them with a super learner
  - (what we did so far)
- Stacking a grid search
  - combine multiple base learners of the **same type**
  - instead of choosing the one optimal combination of tuning parameters, stack the **best n ones**
  - most effective when the leading combinations show high variability
- Automated machine learning
  - automated search across a **variety** of tuning parameter settings for many **different** base learners
  - H2O implements **AutoML** via `h2o.automl`
  - current base learners: (extremely-randomized) random forest, GBM and DNN

Your turn

That's all folks for **today**. Ready to shutdown.

```
h2o.shutdown()
```

Or not completely yet!

Time to play and experiment **one last time**.

Who builds the best model?

1. If you prefer the **R** environment: build your model on `mtpl_trn` and calculate performance on `mtpl_tst`
2. If you prefer the **H2O** environment: build your model on `train_h2o` and calculate performance on `test_h2o`