Trees, Forests, Boosting

Trees

Motivation

The following fits a **regression tree** for Amount on Duration and Job:

```
R> load("german.rda")
R> require("rpart")
R> atree <- rpart(Amount ~ Duration + Job, data = german)
```

We can inspect the tree by printing it

```
R> atree
```

or by plotting it. For the latter we use

```
R> op <- par(xpd = TRUE)
R> plot(atree)
R> text(atree, use.n = TRUE)
R> par(op)
```

Even more information can be obtained using `summary()`.

Motivation

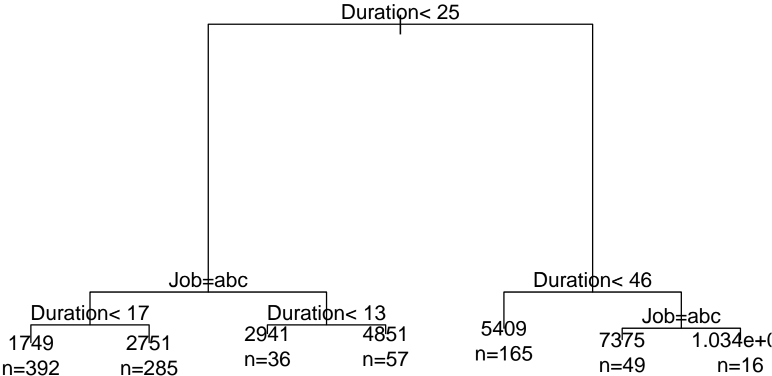
```
R> atree

n= 1000

node), split, n, deviance, yval
      * denotes terminal node

 1) root 1000 7959876000  3271.258
   2) Duration< 25 770 2765776000  2405.131
     4) Job=A171,A172,A173 677 1519491000  2170.721
       8) Duration< 17 392  642068600  1748.753 *
       9) Duration>=17 285  711620700  2751.112 *
     5) Job=A174 93  938285500  4111.538
      10) Duration< 13 36  265429300  2940.556 *
      11) Duration>=13 57  592316400  4851.105 *
   3) Duration>=25 230 2682642000  6170.900
     6) Duration< 46 165 1474668000  5408.976 *
     7) Duration>=46 65  869034000  8105.015
      14) Job=A171,A172,A173 49  542591500  7375.388 *
      15) Job=A174 16  220470300 10339.500 *
```

Motivation



Motivation

From either printing or plotting we see that the prediction rule for this tree is obtained as follows:

- One first navigates through the (here, binary) splitting rules. E.g., first rule splits according to `Duration < 25` (left branch) or not (right branch).
- Eventually, when there are no more splits, one has reached a **terminal node** of the tree.
- Each terminal node predicts some value.

E.g., the left-most terminal node (number 8 the printed representation) corresponds to

$$\mathrm{Duration} < 25, \mathrm{Job} \in \{A171, A172, A173\}, \mathrm{Duration} < 17$$

and for such covariates we predict an amount of 1748.753.

Motivation

The splitting rules for this node can be simplified into

$$\text{Duration} < 17, \text{Job} \neq A174$$

and we can easily verify that the prediction is the mean amount for all (392) observations in this node:

```
R> with(subset(german,
+              Duration < 17 & Job != "A174"),
+      mean(Amount))
[1] 1748.753
```

Motivation

Overall, there are 7 terminal nodes, which partition the space of covariates x (here, $(\mathrm{Amount}, \mathrm{Job})$ pairs) into 7 disjoint regions $R_1, \ldots, R_7$.

If $x \in R_j$, we predict y (here, Amount) as

$$r(x) = \bar{y}_{R_j} = \frac{1}{n_j} \sum_{i:x_i \in R_j} y_i, \quad n_j = \#\{i : x_i \in R_j\}.$$

Regression trees

Regression trees (for squared error loss) do the following:

- The space of covariates x is **recursively partitioned** into disjoint regions $R_1, \dots, R_J$.
- For each region R_j, y is predicted as the average of the y_i for which $x_i \in R_j$.

This rule is best for squared error among all rules of the form

$$r(x) = \sum_j c_j I_{R_j}(x)$$

(with I again the indicator function).

But how is the partition into disjoint regions obtained? In general, "it depends" (CART, C4.5, party, ...).

Regression trees

Simple idea (CART, implemented in `rpart::rpart`): split in a way that mean squared error is maximally reduced.

With R some region in the space of predictors x with covariates (awkward notation) $x_1, \dots, x_p$, binary splits for an ordinal x_j split R into the "children"

$$R_{\mathrm{left}}(j, s) = \{x \in R : x_j < s\}, \quad R_{\mathrm{right}}(j, s) = \{x \in R : x_j \geq s\}$$

with error contribution

$$\sum_{i : x_i \in R_{\mathrm{left}}(j,s)} (y_i - \bar{y}_{R_{\mathrm{left}}(j,s)})^2 + \sum_{i : x_i \in R_{\mathrm{right}}(j,s)} (y_i - \bar{y}_{R_{\mathrm{right}}(j,s)})^2.$$

The "best" split is the one which minimizes the error for all possible j and s.

For nominal x_j, we consider all possible splits from combining levels into two groups.

Stop splitting when there are too few observations in a node, or when the reduction in lack of fit is too small. (See `? rpart.control`.)

Regression trees

To grow the tree, simply apply the above split-finding rule and stop "as appropriate".

When to stop? Simple ideas:

- When nodes would get too small (e.g., insist on a minimal number of data points in each of the nodes/regions).
- When splitting no longer improves the squared error (or achieves a desired percentage of reduction thereof).

These simple heuristics typically grow trees which are "too complex".

Regression trees

In our example, splitting as long as possible can be done via

```
R> btree <- rpart(Amount ~ Duration + Job, data = german,
+                  control = rpart.control(minsplit = 2, cp = 0))
```

This has smaller training error than before:

```
R> c(atree = sum( (german$Amount - predict(atree))^2 ),
+    btree = sum( (german$Amount - predict(btree))^2 ))

     atree      btree
4449165047 3906338017
```

but uses many more terminal nodes:

```
R> c(atree = length(unique(atree$where)), btree = length(unique(btree$where)))

atree btree
    7    86
```

So "most likely", it has worse generalization ability. (How could we substantiate that?)

Regression trees

One strategy to control complexity is to perform **cost-complexity pruning**.

For a given tree T_0, its sub-trees are obtained by collapsing internal (non-terminal) nodes. For such sub-trees T with terminal nodes (regions) $R_1, \dots, R_J$, one considers the cost-complexity criterion

$$C_\alpha(T) = \sum_{j=1}^{J} \sum_{i: x_i \in R_j} (y_i - \bar{y}_{R_j})^2 + \alpha J$$

where $\alpha \geq 0$ controls the amount of penalization for size (number of terminal nodes).

One can show: for each α there is a unique sub-tree T_α which minimizes $C_\alpha(T)$ (can be found by weakest link pruning).

Idea now: for fixed α, grow a large tree and apply cost-complexity pruning. Determine the best α via cross-validation.

Regression trees

For rpart, this gives a table of optimal prunings according to the complexity parameter cp employed.

We can get nice summaries using `printcp()` or `plotcp()`, and also inspect the table directly:

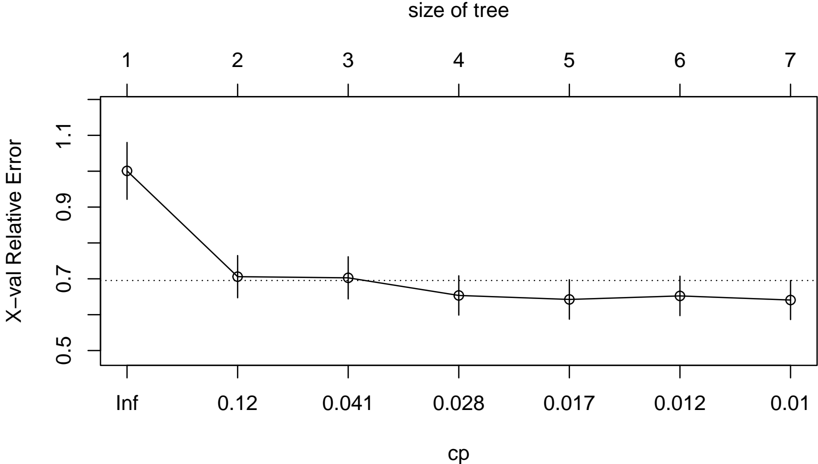
```
R> atree$cptable
```

```
          CP nsplit rel error    xerror       xstd
1 0.31551470      0 1.0000000 1.0009006 0.07924817
2 0.04258099      1 0.6844853 0.7059788 0.05883795
3 0.03869401      2 0.6419043 0.7027514 0.05875552
4 0.02082971      3 0.6032103 0.6536243 0.05481945
5 0.01331330      4 0.5823806 0.6424706 0.05503733
6 0.01011824      5 0.5690673 0.6522976 0.05492994
7 0.01000000      6 0.5589491 0.6408255 0.05444085
```

Clearly, `xerror` is the cross-validated error estimate, and `xstd` the estimated standard deviation of this.

Regression trees

```
R> plotcp(atree)
```



Regression trees

How should we now prune the tree?

Obvious idea: use a `cp` which gives the smallest `xerror`.

However, standard recipe: take the smallest tree where `xerror` does not exceed the minimal `xerror` plus its standard error.

In the plot: when the `xerror` first gets below the dotted line.

So in our case, the optimal tree has size 4, i.e., 3 splits:

```
R> ptree <- prune(atree, cp = 0.021)
```

Regression trees

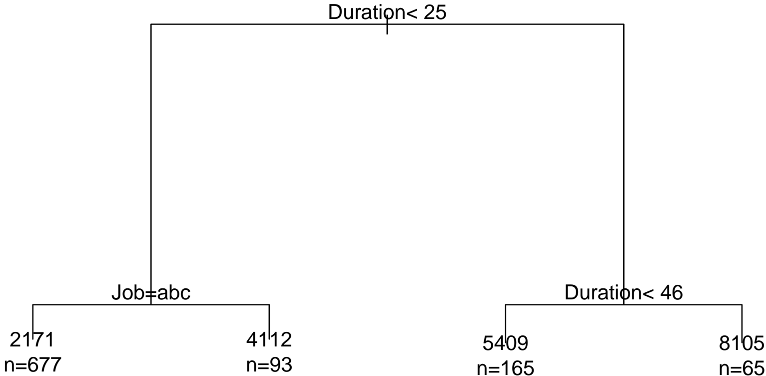
```
R> ptree

n= 1000

node), split, n, deviance, yval
      * denotes terminal node

1) root 1000 7959876000 3271.258
  2) Duration< 25 770 2765776000 2405.131
    4) Job=A171,A172,A173 677 1519491000 2170.721 *
    5) Job=A174 93  938285500 4111.538 *
  3) Duration>=25 230 2682642000 6170.900
    6) Duration< 46 165 1474668000 5408.976 *
    7) Duration>=46 65  869034000 8105.015 *
```

Regression trees



Regression trees

Interestingly, there is no built-in functionality for optimal pruning! (Could easily write an R function doing this for us, of course.)

Component `variable.importance` of the fitted tree gives the aggregate reduction in lack of fit obtained by splitting on the variables used for splitting:

```
R> (varimp <- ptree$variable.importance)

  Duration        Job
2850397099  307999478

R> prop.table(varimp)

  Duration        Job
0.90248233 0.09751767
```

Classification trees

Suppose now the response variable y is categorical with K classes.

Prediction is straightforward: if we use a single value/class for each terminal node/region, then to minimize classification error we need to do the following: if $x \in R_j$, then

$$r(x) = \text{the class } k \text{ with the most } x_i \in R_j.$$

Formally, with

$$\hat{p}_{jk} = \frac{1}{n_j} \#\{i : x_i \in R_j, y_i = k\}$$

the relative frequency of training data with $x_i \in R_j$ and y_i from class k,

$$r(x) = \mathrm{argmax}_k \hat{p}_{jk}, \quad x \in R_j.$$

Classification trees

But how to split?

Obvious idea: choose splits to maximally reduce classification error.

However: one typically gets "better" results by instead splitting to maximally increase **purity** (reduce impurity), measured by either the **Gini index**

$$\sum_{k=1}^{K} \hat{p}_{jk}(1 - \hat{p}_{jk})$$

or **entropy**

$$-\sum_{k=1}^{K} \hat{p}_{jk} \log(\hat{p}_{jk}).$$

(Both are smallest when one p is one and all others zero.)

For rpart, default is to use Gini for classification trees. (See argument `parms` to `rpart()` for details.)

Classification trees

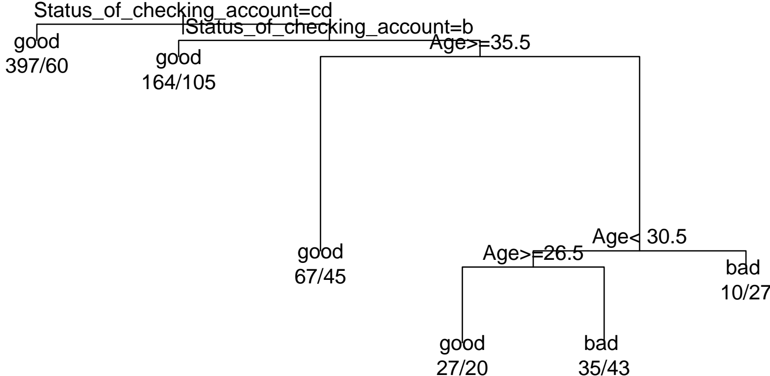
```
R> ctree <- rpart(Class ~ Age + Status_of_checking_account, data = german)
R> ctree

n= 1000

node), split, n, loss, yval, (yprob)
      * denotes terminal node

 1) root 1000 300 good (0.7000000 0.3000000)
   2) Status_of_checking_account=p_hi,none 457  60 good (0.8687090 0.1312910) *
   3) Status_of_checking_account=neg,p_lo 543 240 good (0.5580110 0.4419890)
     6) Status_of_checking_account=p_lo 269 105 good (0.6096654 0.3903346) *
     7) Status_of_checking_account=neg 274 135 good (0.5072993 0.4927007)
      14) Age>=35.5 112  45 good (0.5982143 0.4017857) *
      15) Age< 35.5 162  72 bad (0.4444444 0.5555556)
        30) Age< 30.5 125  62 bad (0.4960000 0.5040000)
          60) Age>=26.5 47  20 good (0.5744681 0.4255319) *
          61) Age< 26.5 78  35 bad (0.4487179 0.5512821) *
        31) Age>=30.5 37  10 bad (0.2702703 0.7297297) *
```

Classification trees



Classification trees

What about pruning?

```
R> ctree$cptable
```

```
          CP nsplit rel error xerror       xstd
1 0.02000000      0 1.0000000   1.00 0.04830459
2 0.01166667      3 0.9400000   1.02 0.04857571
3 0.01000000      5 0.9166667   1.05 0.04896427
```

So the best classification tree would not split at all!

This would give an in-sample classification error of 30%.

For the fitted tree, we would get a misclassification rate of

```
R> mean(german$Class != predict(ctree, type = "class"))
```

```
[1] 0.275
```

Classification trees

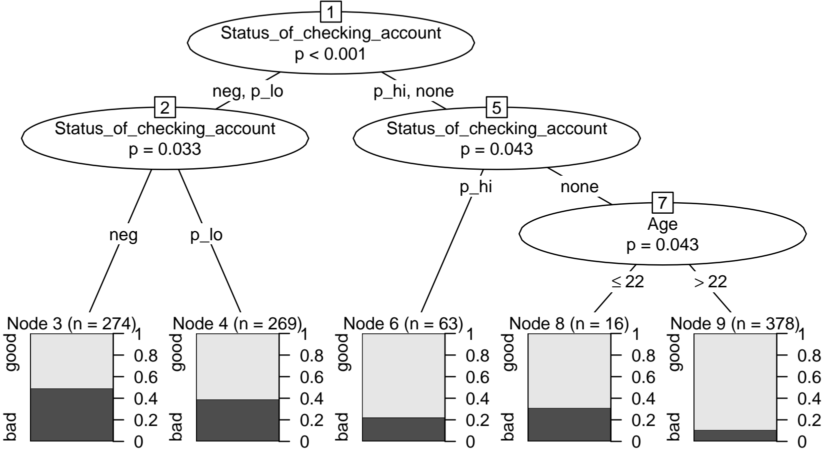
As already mentioned, there are also other "heuristics" for obtaining classification or regression trees.

E.g., packages `partykit` (and `party`) provide "conditional inference trees" which find splits using conditional independence tests.

```
R> ctree <- partykit::ctree(Class ~ Age + Status_of_checking_account,
+                           data = german)
```

```
R> plot(ctree)
```

Classification trees



Classification trees

Plot definitely looks nicer, but still:

```
R> mean(german$Class != predict(ctree, type = "response"))
[1] 0.3
```

Trees: pros and cons

+ Trees are very easy to explain.

Trees: pros and cons

+ Trees are very easy to explain.
+ Some people believe that trees work similarly to human decision making.

Trees: pros and cons

+ Trees are very easy to explain.
+ Some people believe that trees work similarly to human decision making.
+ Trees can be plotted nicely.

Trees: pros and cons

+ Trees are very easy to explain.
+ Some people believe that trees work similarly to human decision making.
+ Trees can be plotted nicely.
+ Trees can easily handle factors.

Trees: pros and cons

+ Trees are very easy to explain.
+ Some people believe that trees work similarly to human decision making.
+ Trees can be plotted nicely.
+ Trees can easily handle factors.
– Trees generally do not give the best predictions.

Trees: pros and cons

+ Trees are very easy to explain.
+ Some people believe that trees work similarly to human decision making.
+ Trees can be plotted nicely.
+ Trees can easily handle factors.
− Trees generally do not give the best predictions.
− Trees can be rather non-robust ("unstable"): small changes in the data can cause large changes in the fitted tree model.

Forests

Bagging

Trees are “nice” (see the pros above). When grown big, they can achieve low bias.
However, they have **high variance**:

- small changes in the data can result in rather different trees,
- fitting trees to two random halves of the data will usually give quite different results.

Is there a way to reduce variance?

Bagging

Trees are "nice" (see the pros above). When grown big, they can achieve low bias. However, they have **high variance**:

- small changes in the data can result in rather different trees,
- fitting trees to two random halves of the data will usually give quite different results.

Is there a way to reduce variance?

Idea: means of i.i.d. random variables with variance σ^2 have variance σ^2/n. Perhaps averaging trees obtained from independent data sets will reduce variance (without increasing bias)?

Bagging

How could we obtain independent data sets?

Bagging

How could we obtain independent data sets? Use bootstrap samples!

Bagging

How could we obtain independent data sets? Use bootstrap samples!

Motivates the following idea for regression trees: given training data D,

- generate bootstrap data sets $D^*_{(b)}$, $b = 1, \dots, B$
- fit trees T_b to the $D^*_{(b)}$.
- use aggregate prediction rule

$$r(x) = \frac{1}{B} \sum_{b=1}^{B} r_b(x), \qquad r_b \text{ rule corresponding to } T_b.$$

I.e., predict using the average of an **ensemble of trees**, also known as a **forest**.

For classification trees, use **majority voting** to aggregate.

This approach is called **bagging** (bootstrap aggregating). Clearly, not restricted to trees.

Bagging

Bagging also allows to estimate errors rather straightforwardly: for each b, estimate performance using the observations in D but not in $D^*_{(b)}$ as ("independent") test set.

These observations are called the **out-of-bag** (OOB) observations.

Remember: with $N_1, \dots, N_n$ the counts of the (mutually distinct) $x_1, \dots, x_n$ in the boostrap samples, $(N_1, \dots, N_n)$ is Multinomial with parameters n and $1/n, \dots, 1/n$.

Each N_i is thus Binomial with parameters n and $p = 1/n$. Hence:

$$\mathbb{P}(N_i = 0) = (1 - 1/n)^n \approx e^{-1} = 0.3678794.$$

On average, roughly $1/3$ of the observations are not in a bootstrap sample (and hence, are OOB observations).

Bagging

Of course, a forest model can no longer nicely be visualized: no longer conveniently interpretable.

Can assess the importance of each variable as the average of the importances in the individual trees.

However: simple bagging does not reduce variance as much as hoped for.

Reason: predictions of trees from independent samples are not really independent.

Idea: increase independence by using "random splits". More precisely:

Before each split, select $m \le p$ covariates at random as candidates for splitting.

This gives **random forests**.

Random forests

Random forest algorithm

For $b = 1, \dots, B$:

1. Draw a bootstrap sample $D^*_{(b)}$
2. Grow a random-forest tree to $D^*_{(b)}$, by repeating the following for each terminal node of the tree until reaching a minimum node size $n_{\min}$:
   1. Randomly select m from the p covariates.
   2. Perform the best split obtainable with the selected covariates.

To predict, use the average (regression) or majority (classification) of the individual predictions.

Reference implementation: `randomForest::randomForest`, which uses defaults: $m = \lfloor \sqrt{p} \rfloor$, $n_{\min} = 1$ for classification; $m = \lfloor p/3 \rfloor$, $n_{\min} = 5$ for regression.

Using $m = p$ gives general bagging as introduced first/above.

Random forests

```
R> require("randomForest")
R> rf_s <- randomForest(Class ~ Age + Status_of_checking_account,
+                       data = german, importance = TRUE)
R> rf_s

Call:
 randomForest(formula = Class ~ Age + Status_of_checking_account,      data = germa
               Type of random forest: classification
                     Number of trees: 500
No. of variables tried at each split: 1

        OOB estimate of  error rate: 28.9%
Confusion matrix:
     good bad class.error
good  641  59  0.08428571
bad   230  70  0.76666667
```

Random forests

This has slightly better performance than the trees (if we believe in the OOB error estimate).

How can we "understand" the model?

```
R> importance(rf_s)
```

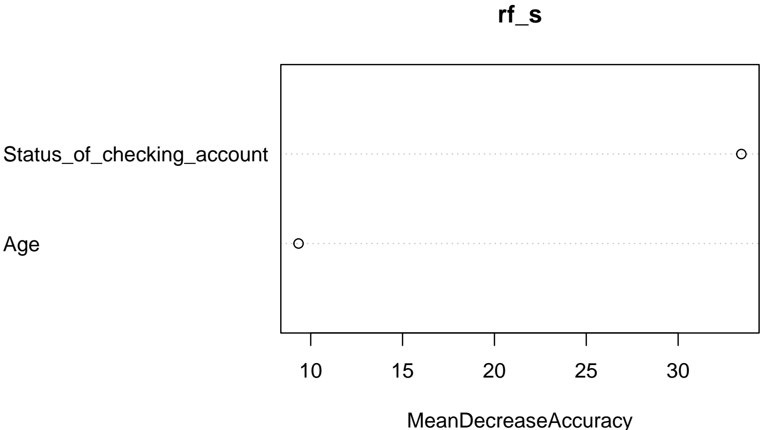
```
                                good       bad MeanDecreaseAccuracy
Age                         5.390686  8.684654             9.336894
Status_of_checking_account 25.918689 33.219812            33.441953
                           MeanDecreaseGini
Age                                31.56906
Status_of_checking_account         56.24369
```

Actually computes a conditional measure of variable importance: for each tree in the forest, compare OOB errors before and after permuting each predictor, and average over all trees.

For accuracy (classification error), this is further broken down by outcome class.

Random forests

```
R> varImpPlot(rf_s, type = 1)
```



Random forests

We can easily make the random forests use all available predictors:

```
R> rf_a <- randomForest(Class ~ ., data = german, importance = TRUE)
R> rf_a

Call:
 randomForest(formula = Class ~ ., data = german, importance = TRUE)
               Type of random forest: classification
                     Number of trees: 500
No. of variables tried at each split: 4

        OOB estimate of  error rate: 23.7%
Confusion matrix:
     good bad class.error
good  636  64  0.09142857
bad   173 127  0.57666667
```

Typically, this gives very good predictive models.

Random forests

To "understand":

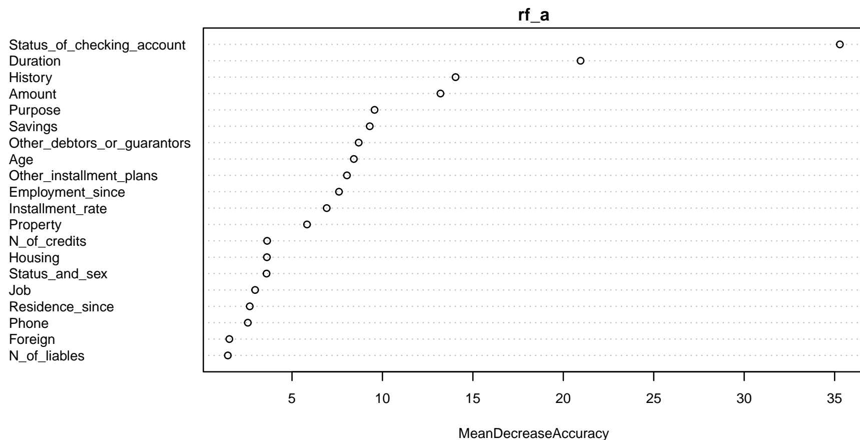
```
R> importance(rf_a, type = 1)

                             MeanDecreaseAccuracy
Status_of_checking_account              35.289803
Duration                                20.953907
History                                 14.042440
Purpose                                  9.563782
Amount                                  13.208536
Savings                                  9.298375
Employment_since                         7.597639
Installment_rate                         6.918505
Status_and_sex                           3.592131
Other_debtors_or_guarantors              8.685955
Residence_since                          2.663559
Property                                 5.829667
Age                                      8.419170
Other_installment_plans                  8.037367
Housing                                  3.612210
```

(and more ...).

Random forests

```
R> varImpPlot(rf_a, type = 1)
```



Random forests

Remember that bagging (random forest) trees can be performed with $m = p$:

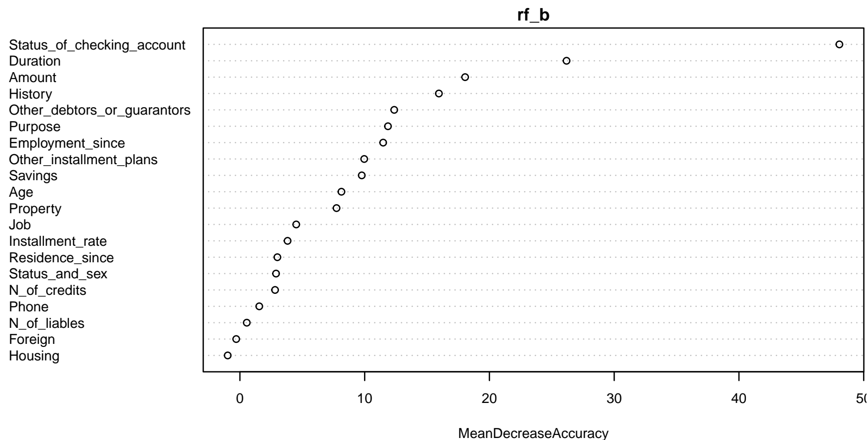
```
R> rf_b <- randomForest(Class ~ ., data = german, importance = TRUE, mtry = 20)
R> rf_b

Call:
 randomForest(formula = Class ~ ., data = german, importance = TRUE,      mtry = 2(
               Type of random forest: classification
                     Number of trees: 500
No. of variables tried at each split: 20

        OOB estimate of  error rate: 23.3%
Confusion matrix:
     good bad class.error
good  615  85   0.1214286
bad   148 152   0.4933333
```

Random forests

```
R> varImpPlot(rf_b, type = 1)
```



Boosting

AdaBoost.M1

Freund and Schapire (1997).

Consider a binary classification problem, with responses ± 1. Algorithm:

1. Initialize observation (case) weights $w_i = 1/n$, $i = 1, \dots, n$.
2. For $m = 1, \dots, M$:
   1. Fit a classifier G_m to the training data using weights w_i.
   2. Compute

      $$\mathrm{err}_m = \frac{\sum_{i=1}^n w_i I(y_i \neq G_m(x_i))}{\sum_{i=1}^n w_i}, \qquad \alpha_m = \log \left(\frac{1 - \mathrm{err}_m}{\mathrm{err}_m} \right).$$

   3. Set $w_i \leftarrow w_i \exp(\alpha_m I(y_i \neq G_m(x_i)))$, $i = 1, \dots, n$.
3. Output $G(x) = \mathrm{sign} \left(\sum_{m=1}^M \alpha_m G_m(x) \right)$.

AdaBoost.M1

One can show: turns weak classifiers (performance $1/2 + \epsilon$) to strong classifiers (performance $1 - \epsilon$).

I.e., **boosts** the performance of classifiers.

Initially believed never to overfit (not even for M very large): not true.

Breiman (1996): when uses with trees (e.g., decision "stumps", i.e., trees with a single split), "best off-the-shelf classifier in the world".

Mystery?

Additive models

Prediction rules of the form

$$f(x) = \sum_{m=1}^{M} \beta_m b(x, \gamma_m)$$

fit additive expansions in a set of "elementary" basis functions. E.g., linear models, single hidden layer neural networks, multivariate adaptive regression splines, trees, ...

Ideally, parameters would be obtained by solving

$$\min_{\{\beta_m, \gamma_m\}_{m=1,\dots,M}} \sum_{i=1}^{n} L\left(y_i, \sum_{m=1}^{M} \beta_m b(x_i, \gamma_m)\right)$$

(writing the discrepancy measure d as the loss function L).

Typically hard optimization problem, so try "greedy" variant by optimizing one (β, γ) pair at a time in one forward pass: **forward stagewise additive modeling**.

Additive models

Best explained for squared error loss $L(y, f(x)) = (y - f(x))^2$:

Already having fitted $m-1$ (β, γ) pairs, we have an expansions f_{m-1} using $m-1$ summands. The loss when adding one more term is

$$
\begin{aligned}
L(y_i, f_{m-1}(x_i) + \beta b(x_i, \gamma)) &= (y_i - (f_{m-1}(x_i) + \beta b(x_i, \gamma)))^2 \\
&= ((y_i - f_{m-1}(x_i)) - \beta b(x_i, \gamma))^2 \\
&= (r_{im} - \beta b(x_i, \gamma))^2
\end{aligned}
$$

where $r_{im} = y_i - f_{m-1}(x_i)$ is the residual of the current model on observation i.

I.e., greedy approximation can be performed by successive least squares fitting to the "current" residuals r_{im}.

Basic idea of "least squares" regression boosting.

AdaBoost.M1

Consider the loss function ("**exponential loss**")

$$L(y, f(x)) = \exp(-yf(x)).$$

One can show: AdaBoost.M1 is equivalent to forward stagewise additive modeling using exponential loss.

Why? At stage m, we need to find (β, G) to minimize

$$\sum_{i=1}^{m} \exp(-y_i(f_{m-1}(x_i)+\beta G(x_i))) = \sum_{i=1}^{m} w_{im} \exp(-\beta y_i G(x_i)), \quad w_{im} = \exp(-y_i f_{m-1}(x_i)).$$

Can rewrite as

$$e^{-\beta} \sum_{i:y_i=G(x_i)} w_{im} + e^{\beta} \sum_{i:y_i \neq G(x_i)} w_{im} = (e^{\beta} - e^{-\beta}) \sum_{i=1}^{n} w_{im} I(y_i \neq G(x_i)) + e^{-\beta} \sum_{i=1}^{n} w_{im}.$$

AdaBoost.M1

If $\beta > 0$, the best G is clearly obtained as

$$\arg\min_G \sum_{i=1}^n w_{im} I(y_i \neq G(x_i)),$$

i.e., by minimizing the weighted misclassification error.

With this as G_m, one then shows that choosing

$$\beta_m = \frac{1}{2} \log \frac{1 - \mathrm{err}_m}{\mathrm{err}_m}$$

gives the minimum over β, with err_m the weighted misclassification error rate.

Updating the approximation as $f_{m-1}(x) + \beta_m G_m(x)$, the weights for the next iterations are

$$w_{im} e^{-\beta_m y_i G_m(x_i)} = w_{im} e^{2\beta_m I(y_i \neq G_m(x_i))} e^{-\beta_m},$$

with $e^{-\beta_m}$ a common factor which has no effect.

Hence, with $\alpha_m = 2\beta_m$ we indeed get AdaBoost.M1.

AdaBoost.M1

Why exponential loss? (And not misclassification error?)

If y is binary with values ± 1,

$$\mathbb{E} e^{-yf} = e^{-f}\mathbb{P}(y=1) + e^{f}\mathbb{P}(y=-1).$$

To minimize over f, set derivative wrt. f to zero:

$$-e^{-f}\mathbb{P}(y=1) + e^{f}\mathbb{P}(y=-1) = 0 \quad \Leftrightarrow \quad e^{2f} = \frac{\mathbb{P}(y=1)}{\mathbb{P}(y=-1)} \quad \Leftrightarrow \quad f = \frac{1}{2}\log\frac{\mathbb{P}(y=1)}{\mathbb{P}(y=-1)}.$$

I.e., AdaBoost.M1 approximates half the log-odds of $\mathbb{P}(y=1|x)$.

If y is binary with values ± 1 and f is half the log-odds of $p = \mathbb{P}(y=1)$,

$$\frac{1}{1+e^{-2yf}} = \frac{1}{1+\left(\frac{1-p}{p}\right)^{y}} = p^{I(y=1)}(1-p)^{I(y=-1)}.$$

I.e., the binomial negative log-likelihood (deviance) is $\log(1+e^{-2yf(x)})$, which has the same minimizer as exponential loss!

Loss functions for classification

When classifying to the “more probable” (ignore ties):

$$\hat{y} = 1 \quad \Leftrightarrow \quad \mathbb{P}(y = 1|x) > 1/2 \quad \Leftrightarrow \quad f(x) = \frac{1}{2} \log \frac{\mathbb{P}(y = 1|x)}{\mathbb{P}(y = -1|x)} > 0,$$

i.e.,

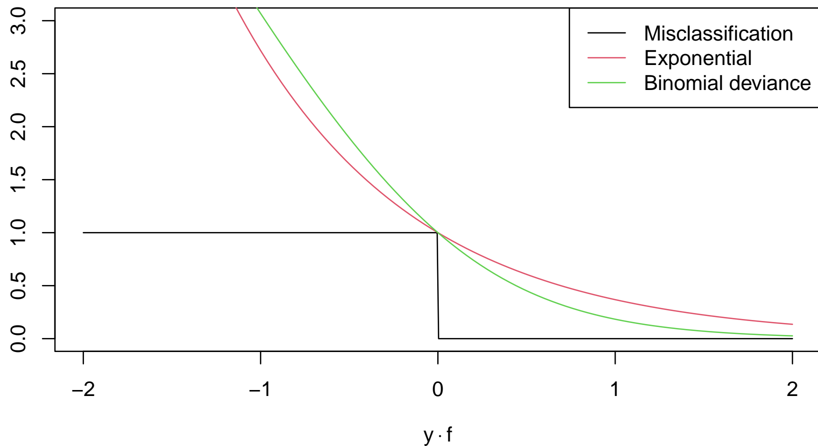
$$\hat{y} = \mathrm{sign}(f(x)).$$

Misclassification loss then becomes

$$L(y, f(x)) = I(yf(x) < 0).$$

Thus: misclassification loss, binomial deviance loss and exponential loss are all functions of the **margin** $yf(x)$.

Loss functions for classification



Gradient boosting

How should we perform forward stagewise additive modeling for arbitrary loss functions L?

In principle, solve

$$\arg\min_{\beta,G} \sum_{i=1}^{n} L(y_i, f_{m-1}(x_i) + \beta G(x_i)).$$

But this may be hard.

Idea: mimic gradient descent. I.e., use a Taylor expansion about the current predictions, and fit the next term to the **working response**.

This gives **gradient boosting** (Friedman, 2001).

Gradient boosting

1. Initialize $f_0(x) = \arg\min_\beta \sum_{i=1}^n L(y_i, \beta)$.
2. For $m = 1, \dots, M$:
   1. Compute the working responses
   $$r_{im} = - \left. \frac{\partial L(y_i, f(x_i))}{\partial f(x_i)} \right|_{f = f_{m-1}}.$$
   2. Fit a regression model G with covariates x_i and responses r_{im}.
   3. Choose the gradient descent step size as
   $$\arg\min_\beta \sum_{i=1}^n L(y_i, f_{m-1}(x_i) + \beta G(x_i))$$
   4. Update $f_m(x) = f_{m-1}(x) + \beta G(x)$.
3. Output $f_M(x)$.

Gradient boosting

Typically, one uses regression trees to fit the regression models.

By controlling the number of splits for the trees one can control the interaction depth K of the models (one split: no interaction, two splits: single interaction, ...).

This gives one **hyperparameter** that may need tuning.

In R, reference implementation is in package `gbm` ("Generalized Boosted Regression Models"). Very popular for reasons we will see below.

This actually also performs sub-sampling of the observations when fitting the trees, and additional shrinkage on the the predictions: two more hyperparameters.

And of course, how should we choose M?

gbm::gbm

Package gbm fits "Generalized Boosted Regression Models".

```
R> require("gbm")
```

The basic function is gbm() with arguments:

```
R> args(gbm)
```

```
function (formula = formula(data), distribution = "bernoulli",
    data = list(), weights, var.monotone = NULL, n.trees = 100,
    interaction.depth = 1, n.minobsinnode = 10, shrinkage = 0.1,
    bag.fraction = 0.5, train.fraction = 1, cv.folds = 0, keep.data = TRUE,
    verbose = FALSE, class.stratify.cv = NULL, n.cores = NULL)
NULL
```

So by default, this performs classification using the binomial deviance loss.

However, for this it needs the responses in $\{0, 1\}$: need to transform Class accordingly.

gbm::gbm

As `Class` is a factor with two levels, we can use `as.numeric(Class) - 1` (but don't show this to your kids):

```
R> with(german, table(Class))

Class
good  bad
 700  300

R> with(german, table(as.numeric(Class) - 1))

  0   1
700 300
```

Hence:

```
R> german1 <- transform(german, Class = as.numeric(Class) - 1)
```

(For consistency, we do not overwrite the original data.)

gbm::gbm

```
R> gbm1 <- gbm(Class ~ ., data = german1, distribution = "bernoulli",
+              interaction.depth = 2)
R> gbm1
```

```
gbm(formula = Class ~ ., distribution = "bernoulli", data = german1,
    interaction.depth = 2)
A gradient boosted model with bernoulli loss function.
100 iterations were performed.
There were 20 predictors of which 18 had non-zero influence.
```

We can use `summary()` to inspect the relative influence of each variable in the fitted model, both numerically and graphically.

(In principle, gives the reduction attributable to each variable in sum of squared error in predicting the gradient on each iteration.)

gbm::gbm

```
R> summary(gbm1, plotit = FALSE)
                                                    var   rel.inf
Status_of_checking_account Status_of_checking_account 22.742252
Purpose                                         Purpose 13.240326
Amount                                           Amount 12.604897
Duration                                       Duration 11.444306
History                                         History  8.439337
Savings                                         Savings  6.157153
Age                                                 Age  5.933471
Property                                       Property  4.018939
Employment_since                       Employment_since  3.596744
Status_and_sex                           Status_and_sex  2.982185
Other_installment_plans         Other_installment_plans  2.211325
Installment_rate                       Installment_rate  2.003878
Housing                                         Housing  1.278961
```

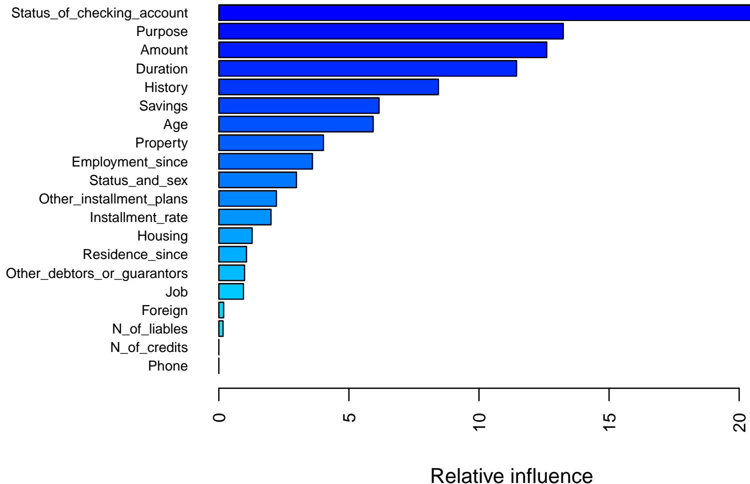
gbm::gbm

By default `summary()` also creates a plot.

Typically, this will have unreadable variable names. To improve, one can use something like the following.

```
R> op <- par(mar = c(5, 10, 1, 1))
R> summary(gbm1, las = 2, cex.names = 0.7)
R> par(op)
```

gbm::gbm



gbm::gbm

How good is this model?

To predict, we need to decide which M to use.

In the simplest case, we determine the best M using the performance on the OOB samples (remember that sub-sampling is performed, by default using 50% of the observations):

```
R> best_M <- gbm.perf(gbm1, method = "OOB")
R> best_M
```

```
[1] 33
attr(,"smoother")
Call:
loess(formula = object$oobag.improve ~ x, enp.target = min(max(4,
    length(x)/10), 50))

Number of Observations: 100
Equivalent Number of Parameters: 8.32
Residual Standard Error: 0.0008951
```

gbm::gbm

By default, predictions have type "link" (i.e., are for log-odds in the Bernoulli case).

Using type "response" will transform back to the response scale (i.e., give probabilities in the Bernoulli case).

```
R> yhat <- predict(gbm1, n.trees = best_M) > 0
R> with(german1, table(Class, yhat))
```

```
     yhat
Class FALSE TRUE
    0   665   35
    1   192  108
```

To compute misclassification error:

```
R> MCE <- function(y, yhat) mean(y != yhat)
R> with(german1, MCE(Class, yhat))
```

```
[1] 0.227
```

gbm::gbm

Not bad, but as the outputs say, choosing the optimal number of trees using OOB error estimation is usually too optimistic.

Better to use CV (which takes quite some time ...) [or a test set]:

```
R> gbm2 <- gbm(Class ~ ., data = german1, distribution = "bernoulli",
+              interaction.depth = 2, cv.folds = 5)
```

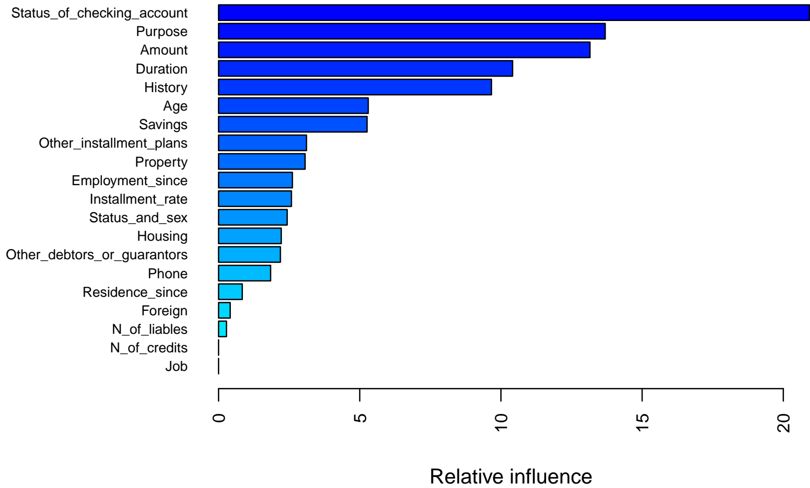
Now we can determine the best M using CV instead of OOB:

```
R> best_M <- gbm.perf(gbm2, method = "cv")
```

And use the best M for summary and prediction:

```
R> summary(gbm2, n.trees = best_M)
```

gbm::gbm



gbm::gbm

Classification performance:

```
R> yhat <- predict(gbm2, n.trees = best_M) > 0
R> with(german1, table(Class, yhat))
```

```
     yhat
Class FALSE TRUE
    0   653   47
    1   142  158
```

```
R> with(german1, MCE(Class, yhat))
```

```
[1] 0.189
```

Note that this is the apparent error rate, and not a cross-validated one. Now `? gbm.object` says that the gbm object returned actually has elements `train.error` and (if CV was used) `cv.error`: but the loss function is binomial deviance loss and not misclassification loss!

gbm::gbm

One does get “cross-validation predicted values” on the scale of the linear predictor (i.e., the predictions on the hold-out folds).

```
R> with(german1, MCE(Class, gbm2$cv.fitted > 0))
```

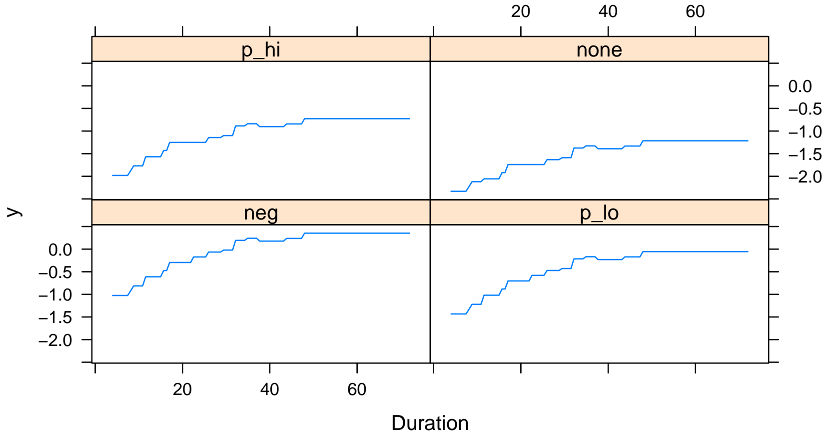
```
[1] 0.244
```

Hmm: not so good anymore?

One can further inspect results via **partial dependence plots**: These plot the marginal effect of the selected variables by “integrating” out the other variables (as described in Friedman (2001)).

```
R> plot(gbm2, i.var = c("Status_of_checking_account", "Duration"),
+       n.trees = best_M)
```

gbm::gbm



gbm::gbm

Note that these only show **averages**. E.g.,

```
R> with(german1, table(Status_of_checking_account, yhat))
```

```
                          yhat
Status_of_checking_account FALSE TRUE
                      neg    150  124
                      p_lo   193   76
                      p_hi    61    2
                      none   391    3
```

Model-based boosting

One does not have to use trees as base learners for boosting.

Bühlmann and Hothorn (2007) introduced **model-based** boosting. For boosting GLMs, this uses sparse linear models as base learners (more precisely: linear predictors using only one covariate).

This conveniently features variable selection (paths) similar to the LASSO.

Model-based boosting

We use package mboost:

```
R> require("mboost")
R> glm1 <- glmboost(Class ~ ., data = german, family = Binomial("glm"))
```

Printing is somewhat verbose:

```
R> glm1
```

and shows the following regression coefficients:

```
R> coef(glm1)
```

```
                 (Intercept) Status_of_checking_accountnone
                 -0.13830858                    -0.82964720
                    Duration                      HistoryA34
                  0.01305658                    -0.09629369
attr(,"offset")
[1] -0.4054651
```

Model-based boosting

In fact, a full main-effects model was investigated:

```
R> betas <- coef(glm1, which = "")
R> length(betas)
```

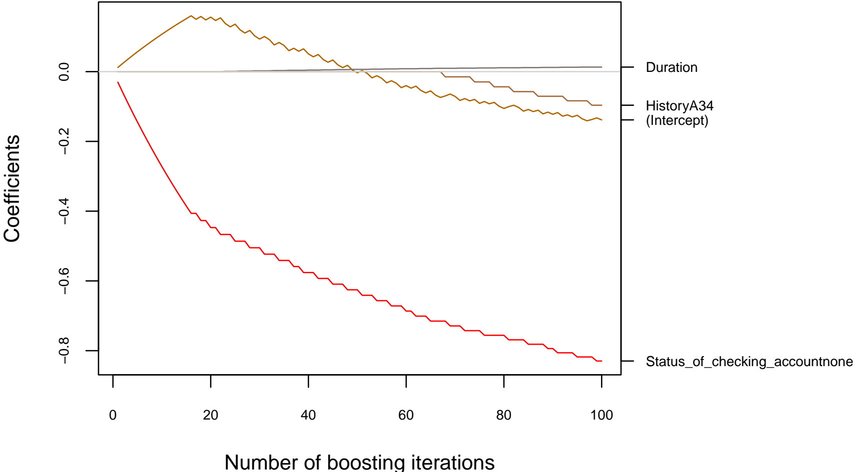
```
[1] 49
```

but only a very few predictors were selected:

```
R> sum(betas != 0)
```

```
[1] 4
```

Model-based boosting



Model-based boosting

However, the obtained model is not very good:

```
R> yhat <- predict(glm1, type = "response") > 1/2
R> with(german, table(Class, yhat))
```

```
      yhat
Class   FALSE TRUE
  good    683   17
  bad     265   35
```

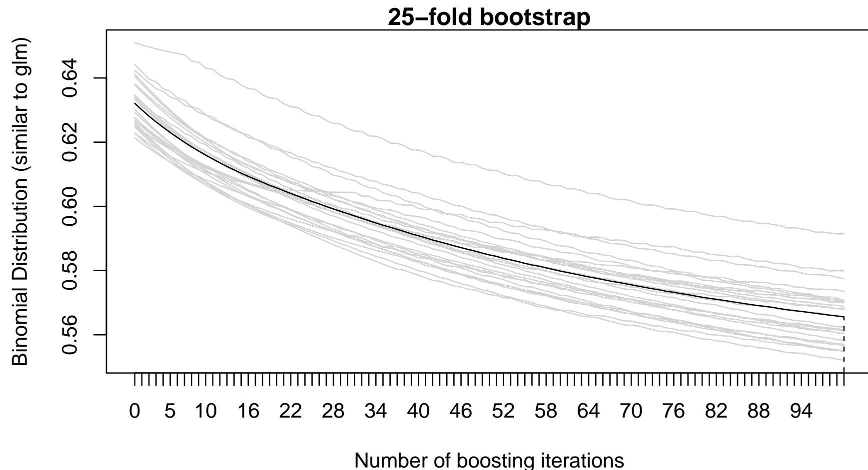
The problem seems to be one of early stopping (opposite of overfitting): the cross-validated prediction errors kept decreasing when the iteration limit M was reached:

```
R> cvm <- cvrisk(glm1)
```

(By default, this uses 25 folds.)

Model-based boosting

```
R> plot(cvm)
```



Lessons learned

- Many of the "modern" modeling methods have hyperparameters which need tuning for optimal performance.
- Often, there is built-in functionality for tuning some of these using cross-validation.
- Performance is not necessarily optimized for the measures of interest in the context of default prediction models (misclassification error or AUC).
- Obtaining suitable estimates of these performance measures is not straightforward.

Learning **frameworks** try to help with the above.

In particular, `caret` (Classification and Regression Training, older, likely more popular) and `mlr` (Machine Learning in R, newer and likely more comprehensive).

However: using these efficiently requires to learn how to work with these ...