MSA220/MVE440 Statistical Learning for Big Data

Lecture 3

Rebecka Jörnsten

Mathematical Sciences
University of Gothenburg and Chalmers University of Technology

Classification - selection of tuning parameters

As we saw from examples in the previous lecture, classification methods can be tuned via method-specific parameters.

- kNN - choose the size neighborhood, k
- CART - choose number of rectangular regions or, equivalently, the number of data splits in the classification tree.

How do we choose the best value for the tuning parameter? We could try minimizing the error rate $= \frac{1}{n}\sum_{i=1}^{n} 1\{y_i \neq \hat{c}(x_i)\}$.

Using the error rate is *not* going to work.
If we train and tune/validate the method *on the same data* we will always pick the most complex method (e.g. k=1 neighbors, or many rectangular regions) so that all the observations are correctly predicted by our method.
While such a method may look good at first glance, it is unlikely to work for prediction on future data. Why? The data we train on is subject to chance variation/measurement noise on the features. *Such chance variation is not going to be replicated on future data*.
If we train a method to fit chance variation perfectly we tune the method to patterns that are unrelated to the classes and this may lead to spurious class boundaries and many misclassifications on future data.
The key is *generalizability* - we want to tune our method so that it works well for predicting class labels on future data.

Validation

There are 2 ways to getting an idea of the generalizability of a method:

- Use a test data set.
  - Learn the rule from *training data*
  - Apply the learnt rule to *test data* and compute the *test error rate*, TE
  - Choose the tuning parameter value (k, or number of CART splits) that minimizes the test error.
- Compute the *expected loss*, e.g. expected error rate on future data.
  - This is a thought experiment. We envision all possible test data sets and training data sets and compute the Expectation of the test error (or the expectation of the difference between the (training) error rate and the test error rate).
  - This approach leads to model selection criteria you might recognize from the regression class, e.g. AIC, BIC and Cp.

Bias-Variance Trade-off

Depending on the tuning parameter value, a classification rule can be thought of as *local* or *global*.

local	global
use subset of data	use all data
flexible	more rigid
allow for complex boundaries	assume an underlying model for the data distribution
example: kNN with small k	example: logistic regression
need a lot of data to train on	requires less data in general

Bias-Variance Trade-off

We have a whole spectrum of methods than span from the most local to the most global.
Where on this spectrum we should be depends on a lot of things: the true shape of the class boundary, the amount of data we have to train on, the complexity of the method we are using, the dimensionality of the problem.
We use *test data* to figure out how local or global we should be for the situation at hand.

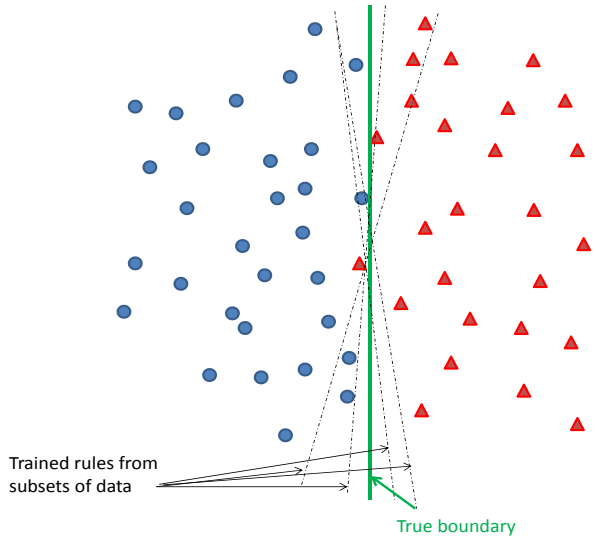
Bias-Variance Trade-off

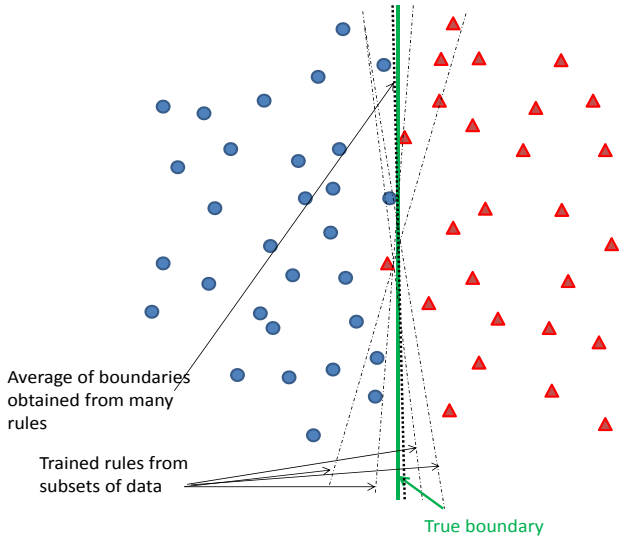
Why aren't the most flexible methods always the best?
"Flexibility" is usually thought of as a positive quality.
In statistics, flexibility can be bad since being flexible means *variable* - the method is sensitive to small changes to the data and those can be random noise. That is, flexible methods may be too sensitive to small perturbations in the data and lead to a high-VARIANCE estimate (unstable).

Bias-Variance Trade-off

Why aren't global methods always the best then?
Global methods assume that class boundaries are simple and smooth. If this is not true, then the method is oversimplifying the situation. These overly simple class boundaries cannot be remedied by more data since our assumption when using a global method is a smooth or simple boundary.
That means that we are not learning enough from the data, or adaptive enough. This problem is called BIAS.

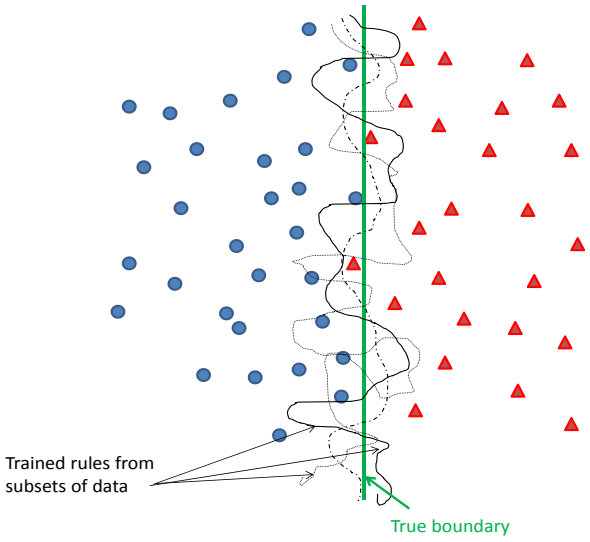
Global rule - simple boundary

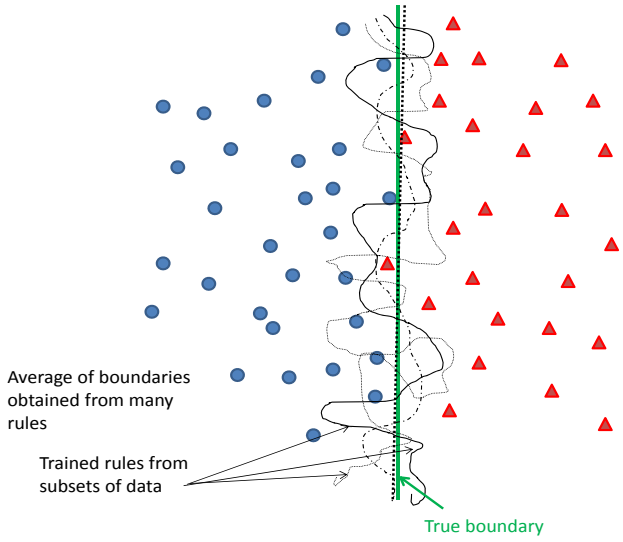




Here is a cartoon example of what happens when you train a simple rule (linear boundary) on a data where a simple rule is the correct assumption.
Simple rules are constrained, here to be linear, so cannot vary much from data set to data set - illustrated by thin black lines. LOW VARIANCE.
If you average all the rules you get another linear boundary (fat black line) that is almost exactly overlapping with the true boundary (green line). LOW BIAS.

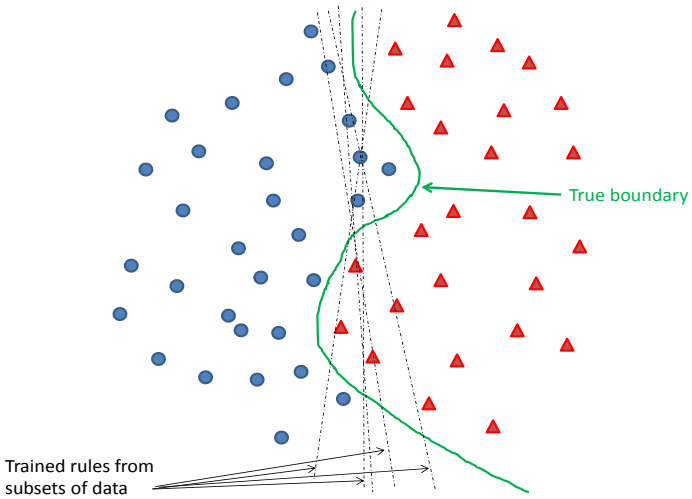
Local rule - simple boundary

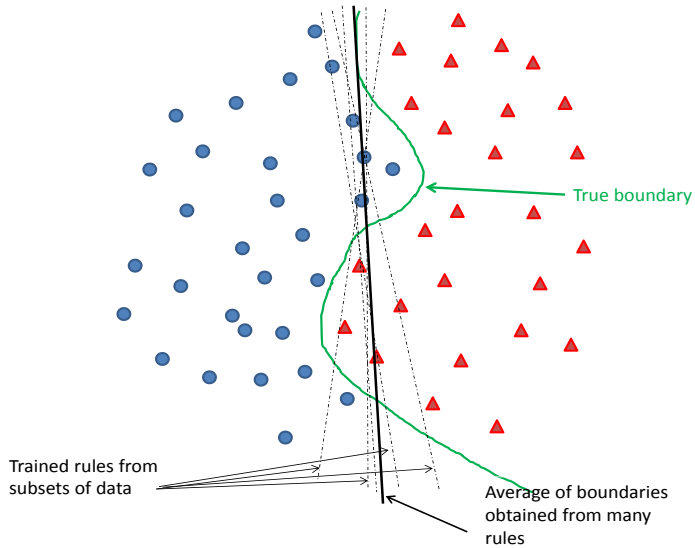




What if you train a local rule, like kNN with small k, to a data set where the true boundary is simple?
Due to the flexible nature of the rule, estimated rule boundaries can vary a lot from training data to training data, illustrated by the thin black lines. Each trained rule can thus be quite far from the true boundary. HIGH VARIANCE.
If you average enough of these wiggly lines all the randomness of training cancels out and you get a rule boundary that coincides with the true boundary. LOW BIAS.

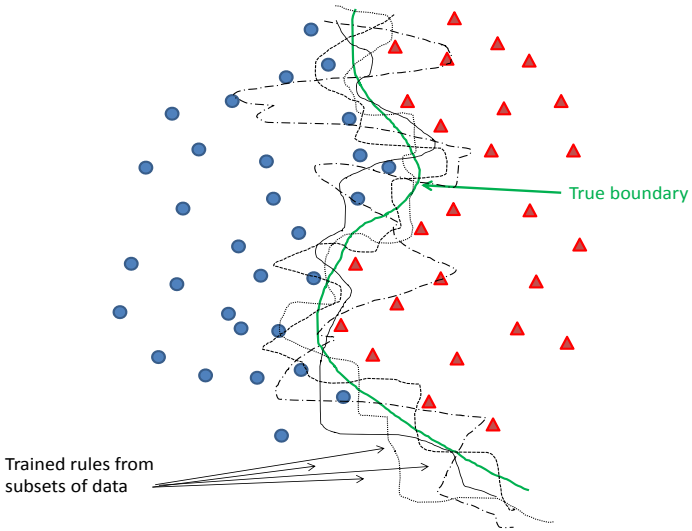
Global rule - complex boundary

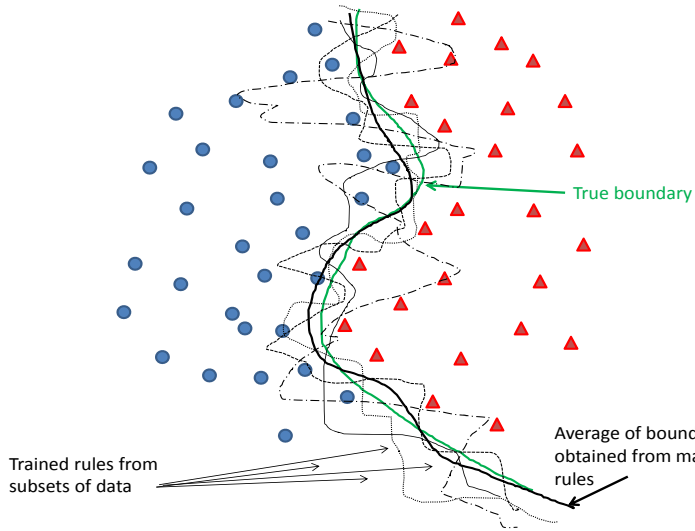




What if you trained a global rule with simple, linear boundary on a data set where the true boundary is complex?
Well,as before, the simple rule is constrained to be linear so cannot vary much from training data to training data. LOW VARIANCE.
If you average out all the randomness of training you get another linear boundary that is clearly an overly simple boundaries that will make a lot of mistakes. LARGE BIAS.

Local rule - complex boundary





If you train a local rule to a data where the true boundary is complex you will again see a lot of variability across different training data. HIGH VARIANCE.
However, if you average out all these rules you get a very good approximation of the true boundary. LOW BIAS.

Local or Global?

The point is of course that in real life situations you won't know what the true boundary looks like!
A local rule *can* capture a complex boundary (LOW BIAS) but you don't know if the rule you get from one particular training data is far from this ideal, average boundary (HIGH VARIANCE).
A global rule *may* have oversimplified the boundary (LARGE BIAS) but at least the rule won't radically change if small changes are made to the data (some extra observations, some noisy observations,...) (LOW VARIANCE).
The best rules control both BIAS and VARIANCE by trading off one against the other. We may be willing to accept a bit of bias if that reduces the estimation variance a lot and vice versa.

Validation

Since we don't know the true boundary we can't compute bias. All we have to go on is the performance of each method in terms of classification errors.
As we discussed, a good measure of performance is the total risk. The error rate on training data is NOT a good estimate of the total risk since we are using the data both to train the rule and evaluate it. This leads to an overly optimistic estimate of risk called the *optimism of training*.
It wouldn't matter so much if all methods suffered from the same level of optimism, but the training error rate is more optimistic for complex methods than simple ones and will always lead to the most complex method considered to be selected.

Validation

A better estimate of total risk is obtained from *test data*.
TRAIN: $\{y_i, x_i\}_{i=1}^{n} \rightarrow$ construct a rule, $\hat{c}(.)$.
TEST: $\{y_i^{new}, x_i\}_{i'=1}^{n'} \rightarrow$ apply rule $\hat{c}(.) \rightarrow$ compute TE, Test Error rate.

$$TE = \frac{1}{n'} \sum_{i'=1}^{n'} 1\{\hat{c}(x_i) \neq y_i^{new}\}$$

Note, we are summing over observations in the test data. The rule $\hat{c}$ has been learnt from the training data and is not updated in any way from test data before it is applied for class prediction.

Validation

In practice we divide the data set into TRAIN and TEST.
How big should n and n' be, respectively?

- small n, large n': poor training, good test error estimation
- large n, small n': good training, poor test error estimation

Cross-validation

It is really better to treat the data symmetrically. Everyone should take a turn to be TRAIN or TEST. These treatment of data is called cross-validation.

1. Divide the data into B equal parts, labeled $b = 1, \cdots, B$, each comprising n/B (closest integer) observations.
2. For $b = 1, \cdots, B$:
   1. Let TEST be the b-th data part and TRAIN all the other $B - 1$ data parts.
   2. Learn the rule from TRAIN and apply it to predict class labels on TEST.
   3. Compute the b-th test error rate, $TE^b = \sum_{i \in b-th\ part} 1\{\hat{c}(x_i) \neq y_i\}$.
3. Compute the total risk estimate, $TE = \sum_b TE^b/n$.

To use this to select between methods, loop over the methods also.

1. Divide the data into B equal parts, labeled $b = 1, \cdots, B$, each comprising n/B (closest integer) observations.
2. For $b = 1, \cdots, B$:
   1. Let TEST be the b-th data part and TRAIN all the other $B - 1$ data parts.
   2. Consider methods $m = 1, \cdots, M$:
      1. Learn the rule m from TRAIN and apply it to predict class labels on TEST.
      2. Compute the b-th test error rate for the m-th method, $TE^{m,b} = \sum_{i \in b-th\ part} 1\{\hat{c}_m(x_i) \neq y_i\}$.
3. Compute the total risk estimate for methods $m = 1, \cdots, M$: $TE^m = \sum_b TE^{m,b}/n$.
4. Choose the method $m^* = \arg\min_m TE^m$

Here, m can refer to kNN with different number of neighbors, CART with different number of regions, or any other methods you want to compare!

How many data parts should we construct?
The most commonly used values for B are n and 10.

- $B = n$ is a special case called *leave-one-out cross-validation*.
  It is very popular because many methods have a short-cut where one can compute all leave-one-out test errors without having to re-learn the rule
  Leave-one-out saves the most data possible for training and one can show that this results in an *unbiased estimate of the true test error* rate. However, because we only leave one data point out each time we train all training data sets are quite similar and this leads to a *highly variable estimate* of the test error rate.

- Small B leaves out a large chunk of data each time for testing. The drawback with this is that you are making training more difficult. This can mean that you get an overly pessimistic estimate of the test error rate, especially for complex methods. On the other hand, because the training data sets are now quite different each time you train, randomness due to training is better cancelled out when you average all the B test error estimates and you get a lower variance estimate of the test error rate than you do with $B = n$.
- There are papers discussing leave-one-out versus leave-k-out crossvalidation (Kerchau Li, Annals of Statistics papers from the 90s).
- In practice, it is a good idea to try a few different values of B to check if method selection is very sensitive to this choice (which really means you should be a bit cautious about interpreting or trusting the selection results).

CART

CART stands for Classification And Regression Trees.
We partition $X - space$ into rectangular regions and assume a constant expected value, or constant posterior probability of a certain class, within each region. This leads to a region-specific class label prediction.
For a CART with M regions, the rule can be written as

$$\hat{c}(x) = \arg\max_{c} \sum_{m=1}^{M} 1\{x \in R_m\} \left(\sum_{i \in R_m} 1\{y_i = c\} \right),$$

where R_m denotes the m-th rectangular region and the expression inside the ( ) brackets computes the class proportion within the region. For regression trees the rule is the mean of y within the region R_m.

CART

We don't allow for any kind of rectangular region because the search for that is too complex (too many choices). Instead, we form the rectangular regions sequentially via binary splits of data. (There are some extensions to CART that allow for more flexible regions, e.g. A. Molinaro).

CART algorithm

1. Consider each feature j and split the data into do parts:
   - data 1 $\{i : \ x_{ij} > T_j\}$
   - data 2 $\{i : \ x_{ij} \leq T_j\}$

   where the threshold T_j is selected to make the data sets as "pure" as possible in terms of class labels.
2. Choose the feature j that is the best in terms of splitting the data into pure parts.
3. Repeat steps 1-2 on data 1 and data 2 separately (i.e. treat each data split as a new data set).

CART algorithm

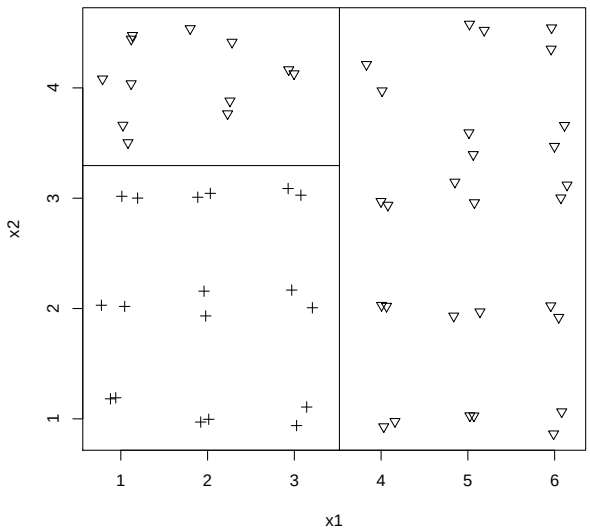
You keep building the CART rule by iterating the algorithm, splitting the data into smaller and smaller parts.
Stopping criteria:

- The number of data splits exceeds M, e.g. 30
- The data set in each region comprises less than o observations, e.g. 5.
- The error rate improvement is below some cutoff, e.g. 1% of the error rate without any data splitting.

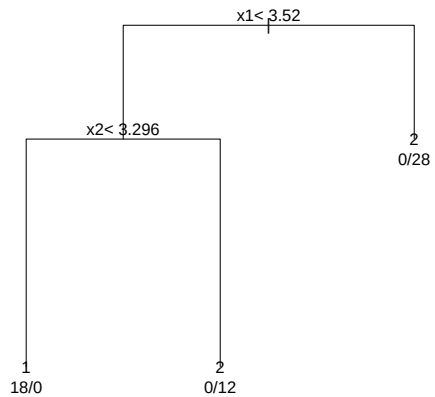
Running CART until one of these stopping criteria kicks in is called the *growing phase* and generates a so-called *max tree*

Visualizing CART

CART is very popular because the data splits that constitute the rule can be visualized as a decision tree. The length of the branches in the tree reflect how much that decision improves the classification error rate.



(a)



(b)

(a): the data splits illustrated. (b): the first data split reduces the error rate from approx 30 to 20%, the second data split reduces it to 0% as is therefore shown as a longer branch.

Splitting criterion

As mentioned above, you split the data into two parts with the goal of making each part as pure as possible in terms of class labels. You can measure this in different ways. One common criterion is error rate (number of mistakes you make). A more popular criterion is the so-called Gini index. This is geared at minimizing the "variance" in each region, specified as follows;

$$GI_m = \sum_{c=1}^{C} \hat{p}(y = c \mid x \in R_m)\hat{p}(y \neq c \mid x \in R_m)$$

where $\hat{p}(y \neq c \mid x \in R_m) = 1 - \hat{p}(y = c \mid x \in R_m)$, and $p * (1 - p)$ is the variance of a binomial random variable. The more mixed region R_m is, the larger the Gini index is. You split on feature j at threshold T_j to minimize the Gini index. In general, GI splitting is very "aggressive" in trying to form pure (single class) regions quickly.

Validation of CART

As with kNN or any other rule, it's usually better to not let the rule be too flexible (too many splits of data). A large tree with many splits corresponds to a local rule that is highly data adaptive and runs the risk of adaptive to noise, i.e. variations in data that is not reproducible on future data.

Short branches toward the bottom of the tree is often an indication that you have over-trained your rule.

To construct a more robust rule we *prune* the tree by cutting short branches (i.e. removing data splits and merging regions).

Does this sound like you're undoing work done in training? Not really, you may need to perform data splits with short branches to get to ones that pay off (long branches). It is only after building the max tree that you can identify unnecessary data splits.

Validation of CART

We have discussed cross-validation. This is a more complex operation for CART. Each validation data can result in a very different max tree with different features used. How do we compare the different trees from the different validation data sets? We have to come up with a measure to identify which pruned max trees to compare for different validation data sets. This is achieved via a complexity cost function and a pruning parameter that controls how much you prune.

Validation of CART

Consider a max tree. Now consider pruning this tree by cutting a branch at the bottom of the tree. There are many such bottom branches to consider, each resulting in a particular error rate.
We define the cost function of a particular tree, T, with $\mid T \mid$ regions as

$$C_\alpha(T) = \sum_{m=1}^{|T|} 1\{y_i \neq \hat{c}(x_i)\} + \alpha \mid T \mid$$

Now, for $\alpha = 0$, the max tree will clearly minimize this cost function (have the smallest error rate. For each $\alpha > 0$ there will be a unique pruned tree that minimizes the cost function.
The point of this: α will be how we "match" different trees to each other in order to compare across validation data sets.

Validation of CART

1. Split the data into B parts, or folds, for cross-validation
2. Build a max tree on each of the folds, holding out the b-th data part for testing.
3. Prune the shortest branches sequentially to minimize the cost-complexity function

$$C_\alpha(T) = \sum_{m=1}^{|T|} 1\{y_i \neq \hat{c}(x_i)\} + \alpha \mid T \mid$$

   This generates a sequence of trees $\{T_\alpha^b, \alpha = 0, \cdots\}$
4. For each α, apply T_α^b to predict data in the b-th test set, resulting in error rate TE_α^b
5. For each α, compute the average error rate $TE_\alpha = \frac{1}{B}\sum_{b=1}^{B} TE_\alpha^b$
6. Identify α^* that minimizes the average error rate: $\alpha^* = \arg\min_\alpha TE_\alpha$

Validation of CART

7. Build the max tree on *all* the original data
8. Prune this max tree using sequence $\alpha = 0, \cdots$ used on the CV data sets, resulting in a sequence of trees T_α
9. Final rule: T_{α^*}, that is, the max tree on all the data pruned to minimize the cost-complexity function with $\alpha = \alpha^*$

So, CV is not used to identify a particular type tree in terms of size or features. CV is used to identify the α^* that tells you how to optimally balance tree size and error rate and matches this balance on the tree from all the data.

Cautionary remarks

- Trees are notoriously unstable - meaning small changes to the data can change the appearance (size and features used) of the tree substantially. Be careful not to read too much into the tree.
- Trees are inappropriate if the true class boundaries are well approximated by linear combinations of x-features. Then discriminant analysis is a better option or extension of CART called MARS.
  You can spot this problem fairly easily - if your tree splits on the same feature again and again, this is a tell-tale sign.
- Check the error rates of each class. Are we obtaining a low error rate because we are simply mislabeling one class completely?

Good things about trees

- CART can easily be adapted to the case of missing values, either by including "missingness" as another feature that you can split on or by using substitute variables.
- Even though CART is unstable, if you exploit this instability by using multiple trees as an ensemble you can improve on the error rate of each single tree. Build not one rule but 100s from random subsets of data. Use a majority vote based on all the trees. This is called *bagging* and is an example of an *ensemble method* (more later).

Random Forest

Random Forest (RF) is a classification and regression method that works for both big p and big n!
It's an example of an *ensemble classifier*

RF algorithm

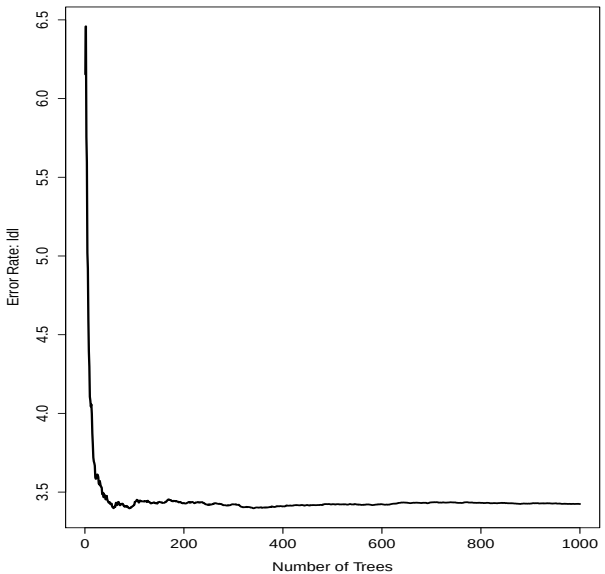
1. For $b = 1, \cdots, B$, draw a subsample of data. The non-sampled data is called the *Out-of-bag* sample, OOB_b
2. Grow a CART tree on the bth subsample as follows:
    1. Draw *mtry* variables at random
    2. Choose the best of the *mtry* variables to split on
    3. Repeat the above 2 steps until a *maxtree* is grown.
3. Compute the overall OOB-error rate for each observation by feeding the observations through each of the b trees where the observation was a member of OOB_b.
4. Final predictions is a majority vote or mean prediction from the B trees.

Random Forest tuning

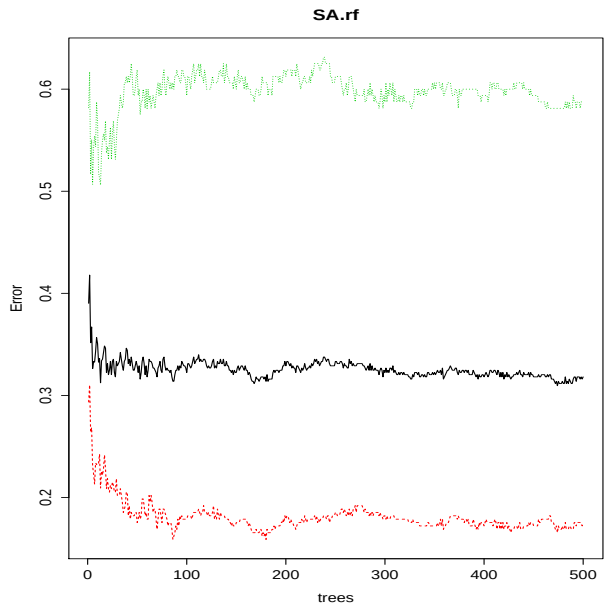
- How trees are built?
  - Often GINI index or MSE
  - Newer implementations include log-rank tests to counter the selection bias for variables with many levels if categorial or continuous variables
- How subsamples are constructed
  - Original RF: sample with replacement
  - Big n: often sample less than m
  - Sample without replacement can help against selection bias
- How we aggregate the trees
  - Original RF: mean or majority vote
  - Imbalanced data: use weighted voting
  - New research into dynamic voting (adaptive).

Random Forest

The OOB error rate is usually a very good estimate of the Prediction error (the performance on future data).
If we plot OOB versus the number of trees we grow we can see that OOB usually levels off - RF doesn't appear to be sensitive to overfitting!
Example: SouthAfrican heart disease data. Predicting cholesterol or chronic heart disease.



LDL regression tree



CHD classification tree

Variable Importance

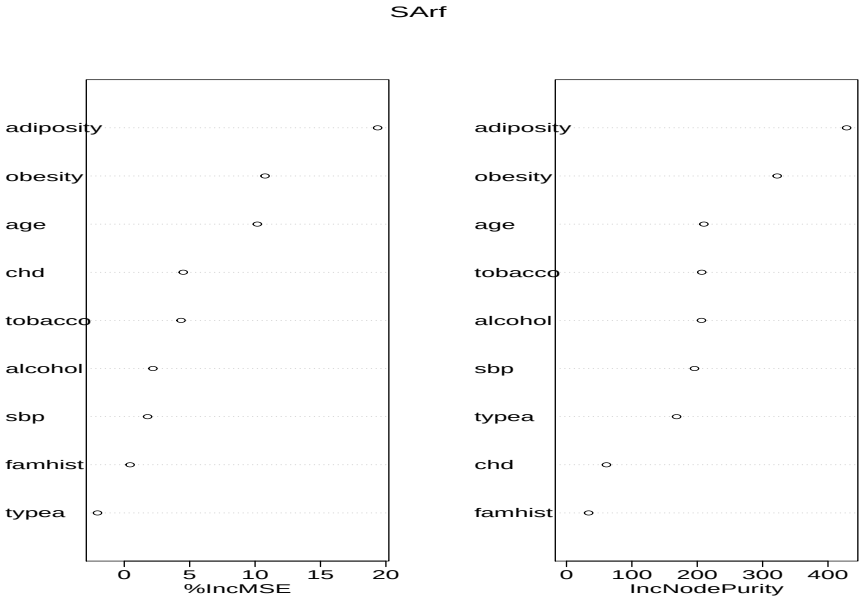
In addition to the OOB error, RF provides several measures of predictor ranking/importance.

Variable importance measures

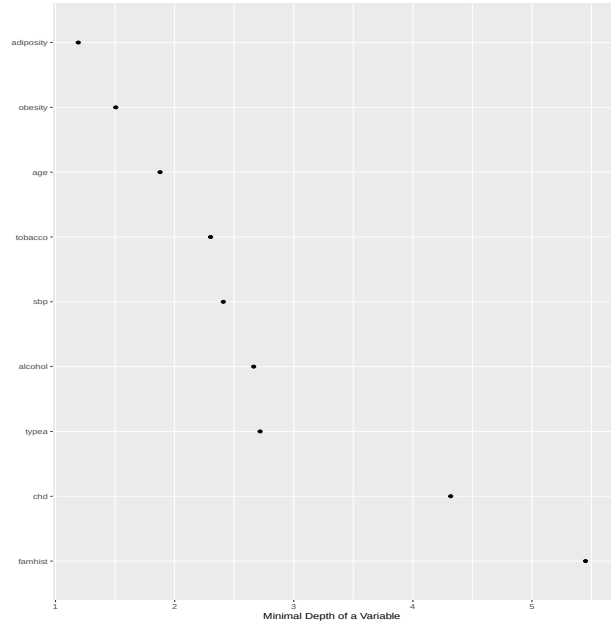
- Gini-index: Sum of the splitting values over all trees for the variable in question. Sensitive to number of levels of a variable or the scale.
- Permutation based
  - Permute variable j values
  - Send observations through the tree
  - Check the OOB error before and after permutation
  - The OOB increase in response to permutation is a measure of the variable importance
  - Some variants of this exists: joint permutations of pairs of features, or daughter nodes, etc.
- Minimum depth: At what level of the tree is the variable used for splitting? The higher up the more important it is.
- And more...

Variable importance

Predicting cholesterol



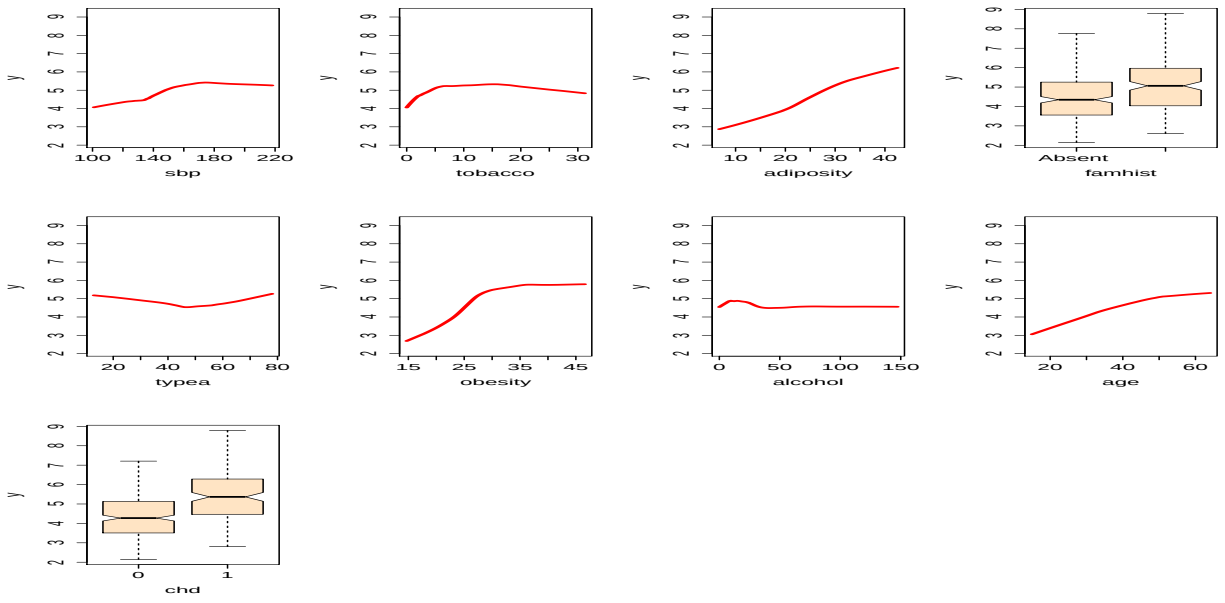
Permutation and Gini



Minimum Depth

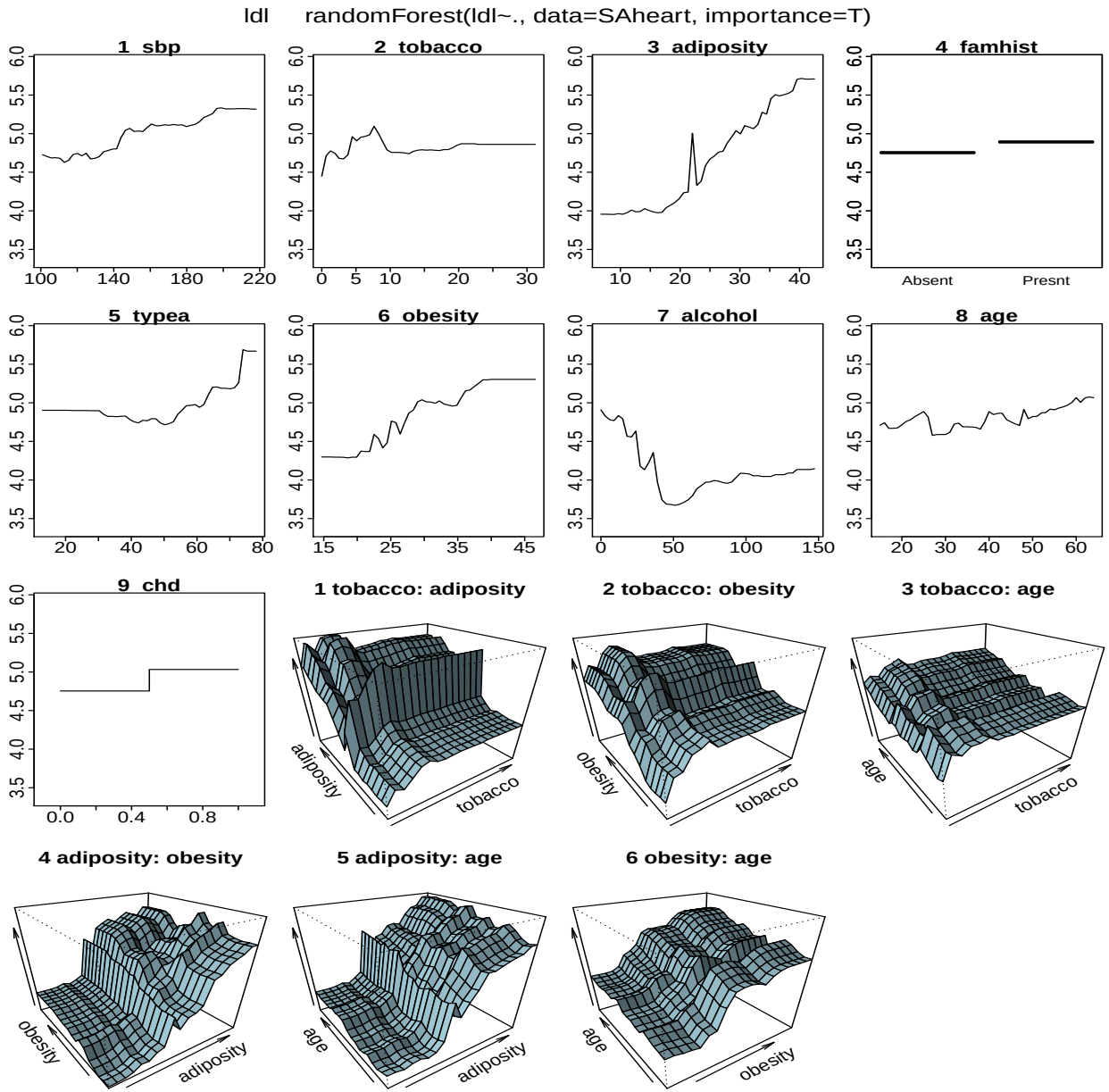
Complex models built via simple steps

Since CART can build complex non-linear models including interactions, so can RF. We can plot the predicted values (smoothed) to see what kind of relationships RF has found.



This is a great RF feature. Remember, each CART model only produces step-wise constant functions. By averaging several trees we get these smooth models that capture the $y - x$ dependencies.

Finding interactions and understanding the fit



Tons of more plots

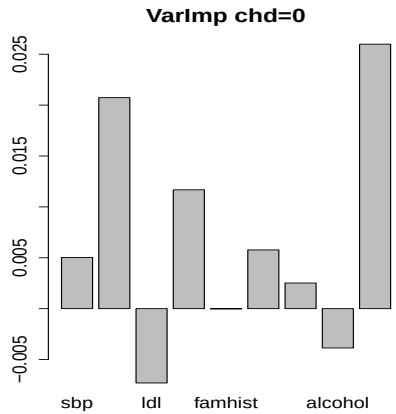
- Partial plots, accounting for other variables
- new packages appear all the time: `forestFloor`, `ggRandomForest`, ...

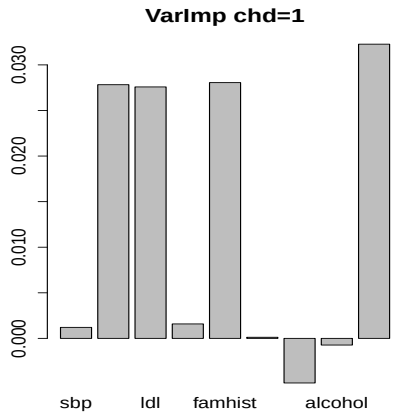
More on classification

Another summary measure we get out of RF is the *proximity*. For each pair of observations we compute the number of times they appear in the same terminal leaf in the each maxtree. If the frequently co-terminate, they are very similar. We can use proximity to:

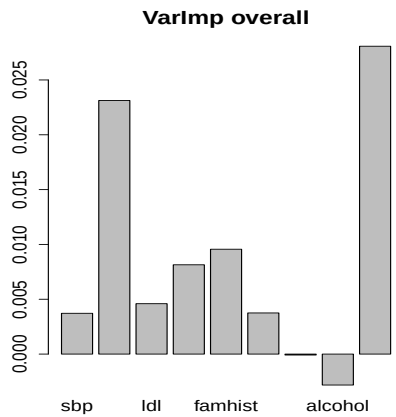
- Plot the data in a lower dimensional space to understand how classes are separated. Feed proximity into e.g. multidimensional scaling or SVD clustering (more later)
- We can identify outliers as observations that don't have close proximity with respect to its own class
- Can use this to adapt RF to do *clustering*!

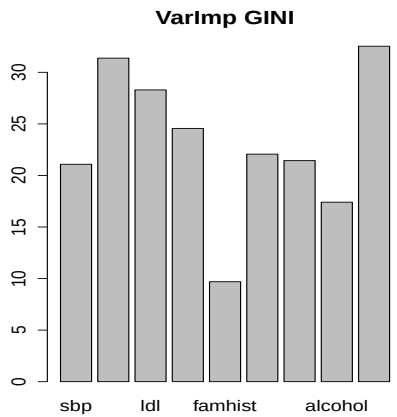
Predicting heart disease status: variable importance for each class



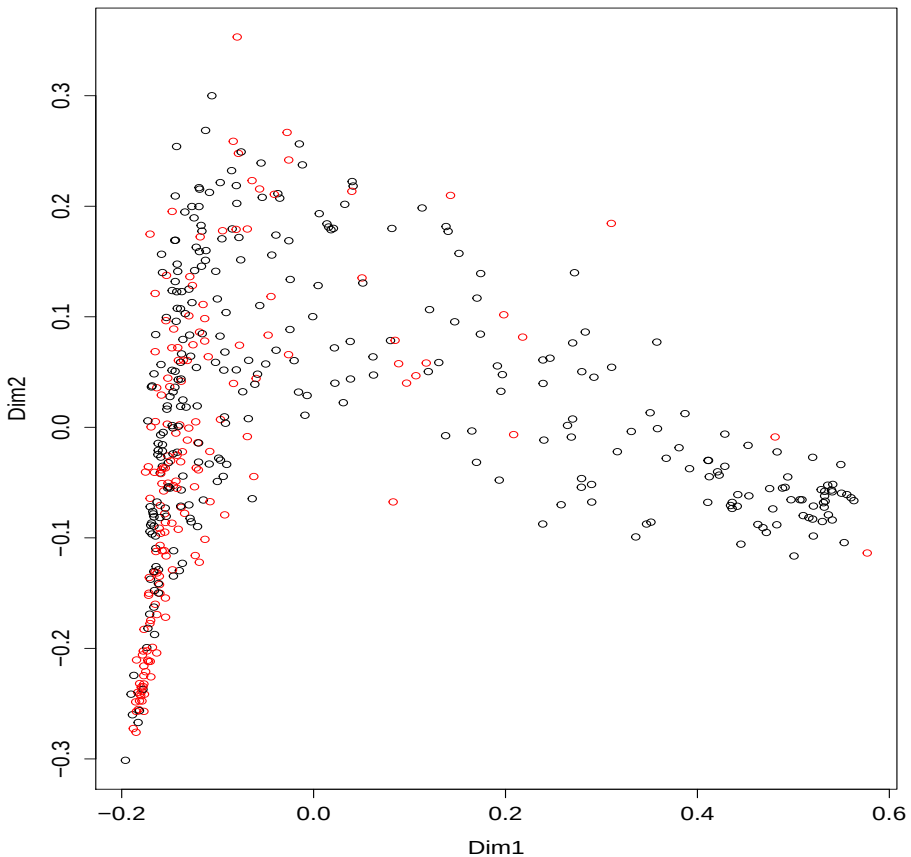


and overall

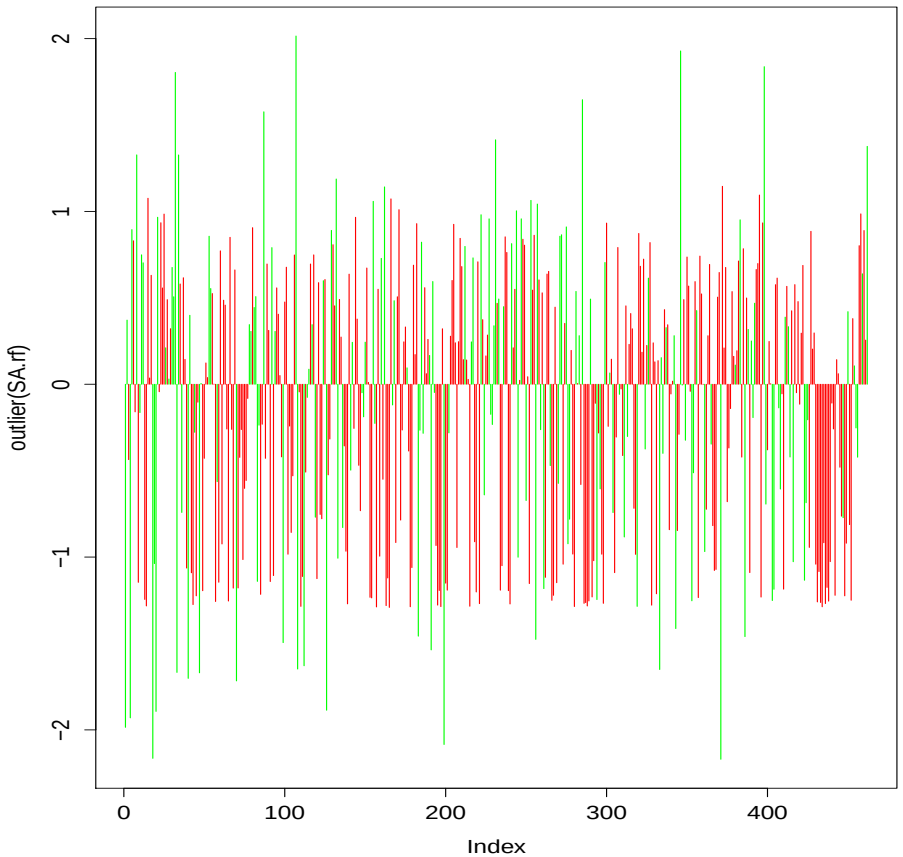




Predicting heart disease status: summarizing proximities



Predicting heart disease status: outlier detection



Unsupervised RF

Generate a *Synthetic data set* by e.g. permuting all the variables in the data set you want to cluster. Call the observations in the original data set for one class and the synthetic data as another class.

- The idea is that the synthetic data set spreads out around the different clusters
- When we try to separate real from synthetic data, terminal leaves will generally form for each of the true clusters is such exist. (See class demo).
- The proximity measure for the real data observations can be used as input into a clustering scheme to identify the data groups, and variable importance measures for clustering obtained.
- `randomForestSRC` package

Tuning RF

- Tends to be stable wrt choices for number of trees and *mtry* (number variables chosen at each node)
- BUT... you need to check for every data scenario (depends on p, n, data structure)
- Bootstrap without replacement is recommended by most recent developers (selection bias reduced)
- Imbalanced data is a problem! If you have small classes among big ones, make sure you grow each tree big enough to capture the small class! You can also try different weighting or subsampling schemes to reduce the dominance of big classes.
- Can aggregation be improved? Recent work on tuning the weights of each tree, or even learn optimal weights.

Big Data

- New packages to scale up and speed up RF.
- RF is not completely parallelizable: OOB computed by a second run on different data through the tree. Variable importance requires a second run through the tree also.
- `Bigrf`, `ranger` and many more