“Big(ger) Data” and Statistical (Machine) Learning in R

Paul Stey

January 22, 2018

Table of Contents

Books

1. "Elements of Statistical Learning", Hastie *et. al*, (2009)
2. "Introduction to Statistical Learning", Gareth *et. al*, (2013)

Note: The authors have made PDFs of both available on the web.

Defining “big data”

Some imperfect but common definitions:

1. Doesn’t fit in RAM
2. Doesn’t fit on disk (or single node)
3. Impossible to store on any number of nodes

Solutions to “Medium Data” Problems

Let’s call problems stemming from data sets that can’t be handled in memory “medium data” problems.

These can be handled in a number of ways:

1. Use a machine with more memory
   - Some servers have up to 2 TB of memory
2. If your problem is mostly data manipulation and aggregation, consider a SQL database
3. Use an on-disk data storage format with your favorite language (e.g., *feather* package in R)

Solutions to "big data" Problems

If size is the only problem, more metal, and clever algorithms are a good-enough solution.

1. Distributed computing solutions:
   - Message Passing Interface (MPI)
   - Hadoop
   - Spark
2. Algorithmic solutions:
   - Split-apply-recombine strategy (e.g., MapReduce)
   - Find parallelism opportunities in sequential problems

The More Interesting Problem

- The more interesting problem related to “big data” is that the complexity of the data forces us to be honest about our *a priori* naïveté.
- This has fueled interest in a number of modeling approaches that make very few assumptions regarding the data generation mechanism.
- This class of modeling approaches is often called “Statistical Learning” or “Machine Learning”

Statistical Machine Learning

The goal in statistical machine learning is usually different from classical statistical methods.

1. We want to make highly accurate predictions or classification
2. We are less interested in precisely estimating effects and understanding the data-generation mechanism

Tree-Based Models

1. ID3 (Quinlan, 1986)
2. C4.5 (Quinlan, 1993)
3. C5.0 (Quinlan, 1996)
4. CART (Breiman, Friedman, Olshen, & Stone, 1984)

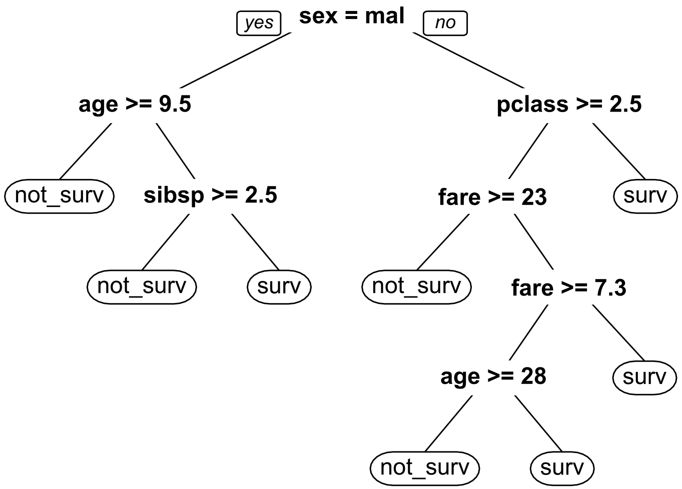
Titanic Example

1. Titantic data
2. Predicting survival (2-class problem)
3. Using demographic and other variables

Titanic Data

survived	age	sex	pclass	fare	sibsp	parch
1	29.00	female	1	211.34	0	0
1	0.92	male	1	151.55	1	2
0	2.00	female	1	151.55	1	2
0	30.00	male	1	151.55	1	2
0	25.00	female	1	151.55	1	2
1	48.00	male	1	151.55	0	0
1	63.00	female	1	26.55	1	0
0	39.00	male	1	0.00	0	0
1	53.00	female	1	51.48	2	0

Titanic Survival



CART Pseudocode

l: current minimum loss at a given node

j^*: index of column in X that is current best predictor

```
while number rows $>$ minimum-node-size
    for $j = 1 : p$
        Find threshold, $t$, of $X_j$ that best partitions data
        If $t$ of $X_j$ produces loss $< l$, $X_j$ is current best predictor
        We update $l$ and $j^*$
    end
    Split data according to whether $X_{ij^*} < t$
    Call cart() on both partitions separately
end
```

Classification and Regression Trees

1. Breiman *et al.* (1984)
2. Tree built by recursive binary splits
3. For regression, choose split that minimizes MSE
4. For classification, choose splits by minimizing node impurity (e.g., Gini index)

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

where

$$\hat{p}_{mk} = \frac{1}{N_m} \sum_{x_i \in R_m} I(y_i = k)$$

is proportion of class k observations in node m

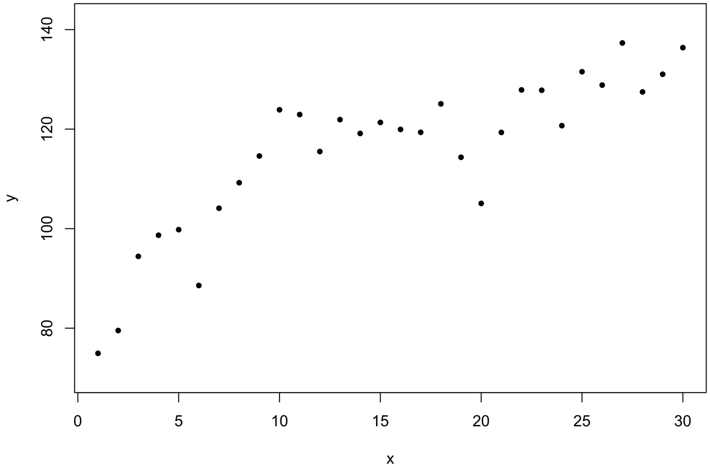
Advantages of Trees-Based Methods

1. Intuitive
2. Viable when $n \ll p$
3. Handle interactions naturally
4. Minimal assumptions
5. Handle missingness in predictors

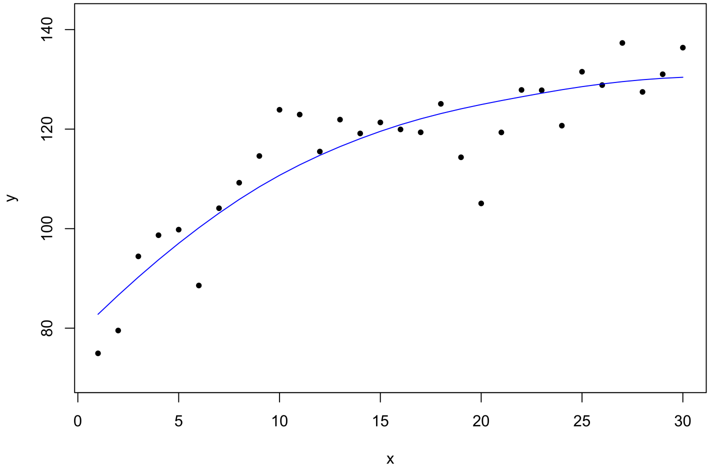
Disadvantages of Single Trees

1. High variance
2. Prone to overfitting
3. Lack of smoothness (troubling for regression)

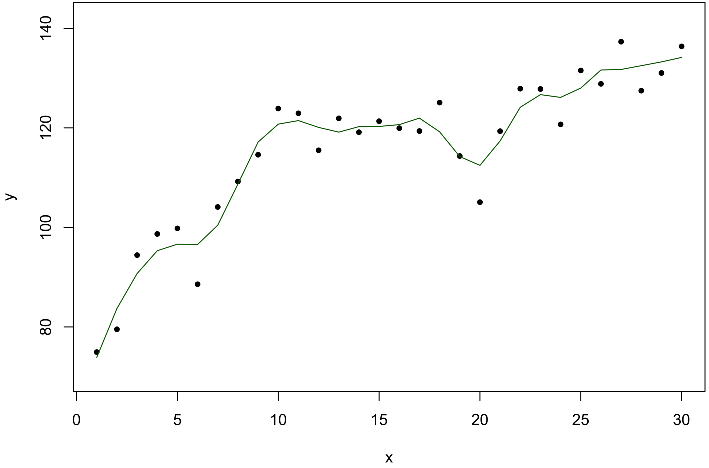
Overfitting Example



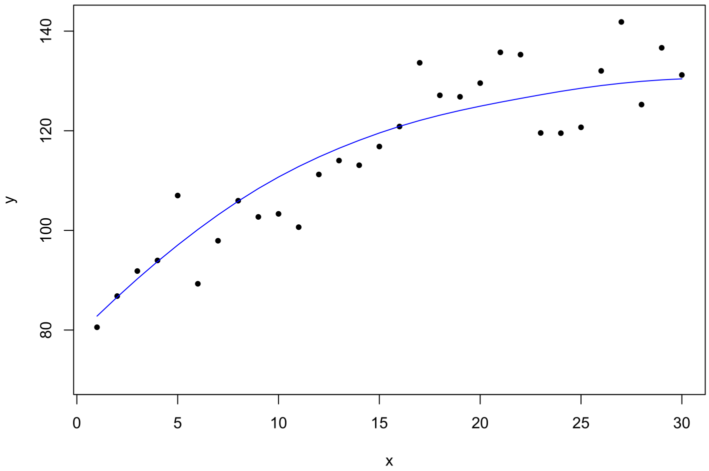
Model 1: Training Data



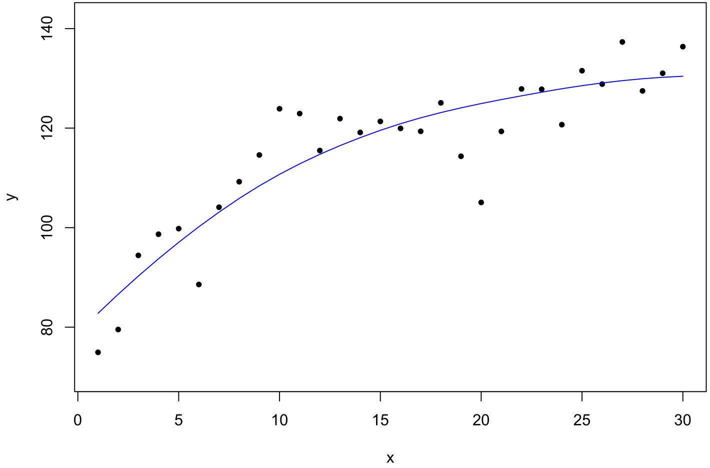
Model 2: Training Data



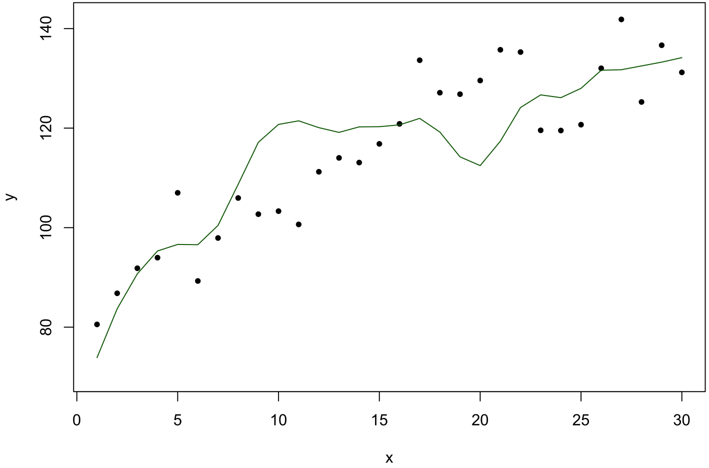
Model 1: Test Data



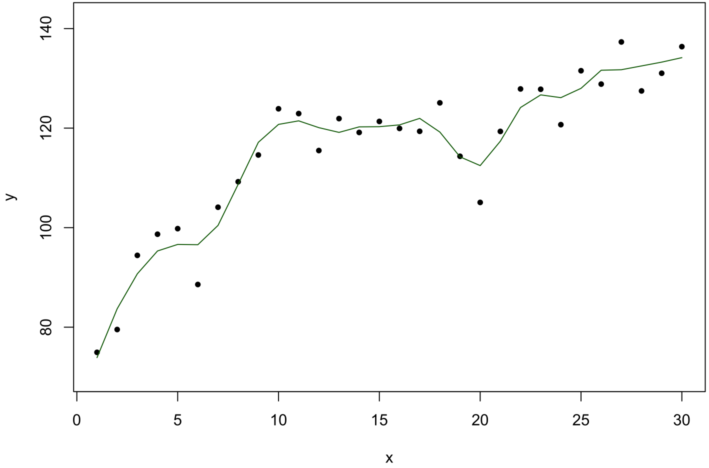
Model 1: with Training Data



Model 2: with Test Data



Model 2: Training Data



Ensemble Concept

1. Fit many simple models, then aggregate
2. Result is “committee” of learners
3. In classification problems, each casts a vote
4. Examples
    1. Boosting
    2. Bagging
    3. Random forests

Advantages of Ensemble Methods

1. Hugely powerful prediction models
2. Usually have few meta-parameters to tune
3. Very powerful “off-the-shelf”
4. Interpretability (*contra* neural networks)

Bootstrap Aggregation (Bagging)

1. Breiman (1996)
2. Extends idea of CART models
   1. Single trees overfit
   2. Stopping rules and pruning help, but only to a point
3. Take B bootstrap samples, build B trees, aggregate predictions
   1. For regression, aggregation is taking the mean of the predicted values for each y_i
   2. For classification, to aggregate means each tree casts a "vote" for each y_i

Bootstrap Samples

$x = [1, 5, 3, 7, 9]$

B^{*1}	B^{*2}	B^{*3}
3	7	5
1	1	3
9	7	3
5	3	9
9	1	1

Bagging Pseudocode

```
for $b = 1 : B$
    Sample random $N$ records with replacement
    Fit classification (or regression) tree using CART
    Add $b^{th}$ tree to ensemble
end
Aggregate predictions for input $X$
```

Advantages of Bagging

1. Big improvement in predictive performance
2. Variance decreases
3. Easy to tune
4. Embarrassingly parallel
5. Out-of-bag (OOB) error estimate (more on this later)

Random Forests

1. Ho (1995), Breiman (2001)
2. Extends idea of bagging
3. Build many trees, aggregate predictions
4. Only consider m predictors at each node
5. Improve predictive performance
    1. Reduce inter-tree correlation
    2. Further reduce variance beyond bagging

Random Forest Pseudocode

```
**for** $b = 1 : B$
    Sample random $N$ records with replacement
    Fit classification (or regression) tree
        At each node in tree, consider only $m$ predictors
        Randomly select new subset of $m$ predictors at each node
    Add $b^{th}$ tree to ensemble
**end**
Aggregate predictions for input $X$
```

Out-of-Bag (OOB) Samples

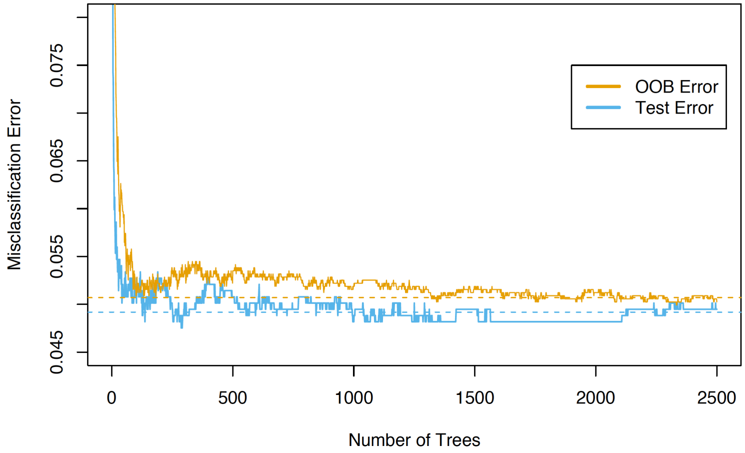
"*For each observation $z_i = (x_i, y_i)$, construct its random forest predictor by averaging* ***only*** *those trees corresponding to bootstrap samples in which z_i* ***did not*** *appear.* (Hastie *et al.* p. 593, 2009)"

OOB Samples cont.

In summary:

1. For each bootstrap iteration, we have some (y_i, x_i) that weren't used in tree building
2. Use these (y_i, x_i) OOB samples to estimate test error
3. Also use these (y_i, x_i) to generate measures of variable importance (more on this later)

OOB Error and Test Error

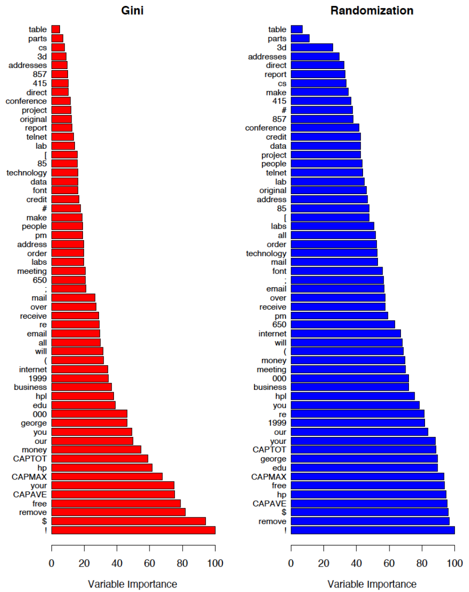


Variable Importance

Variable importance can be computed in a few different ways.

1. Gini improvement
2. OOB randomization
   1. Fit b^{th} tree
   2. Record prediction accuracy for OOB samples (or MSE in regression)
   3. Shuffle the x_i in OOB samples
   4. Compute decrease in prediction accuracy

Variable Importance



Overfitting

1. It was once (mistakenly) believed that random forests would not over fit
2. You *can* overfit by growing deep trees
3. Growing too many trees won't cause you to overfit (but it's wasteful of computing resources)
4. In general, people worry *much* less about overfitting in random forests (relative to other approaches)

Tree Depth

Tree depth can be controlled in two ways:

1. Set max number of terminal nodes (or interaction depth)
   - `maxnodes` parameter in R
   - In scikit-learn, `max_depth`
2. Specifying minimum number of observations in a node
   - `nodesize` parameter in R
   - In scikit-learn, `min_samples_split`

Noisy Data

1. In very noisy data, boosting will tend to outperform random forests
2. Due to random subset of predictors being used at each node
   - Often, only noise variables may be selected as candidates for splitting
3. This effect is mitigated when:
   - Number of non-noise variables increases even slightly
   - We set m to be larger giving us a better chance of selecting non-noise variables

Other Remarks

1. Often, boosting will tend to *slightly* outperform random forests
   - This is very case-specific, however
   - Boosting will tend to have the advantage in very noisy data
2. Bias/Variance tradeoff in random forests vs. boosting

Advantages of Random Forests

1. Powerful prediction models
2. Easy to tune
3. Difficult to overfit (more trees won't overfit)
4. Variable importance
5. Embarrassingly parallel (unlike boosting)
6. Out-of-bag (OOB) error estimate
    1. Excellent approximate of test error
    2. No need for cross validation*

Disadvantages of Random Forests

1. Viewed as “black box” approach
2. Variable importance must be interpretted cautiously
   - Less interpretable than parameter estimates in parametric models
   - Cannot meaningfully compare random forest to boosting
3. Trouble predicting severe outliers
   - In regression, training data constrains the max (or min) $\hat{y}_i$ values
   - In parametric models, the $\hat{y}_i$ values aren’t bounded in this way

Random Forest Example

`<EXAMPLES_IN_R>`

Scientific Software: Current State of the Art

- The *overwhelming* majority of performance-critical code—scientific of otherwise—is written in C, C++, or Fortran. These compiled languages have benefitted from decades of optimization.
- These languages (especially C++) are typically considered more difficult than the newer technical computing languages (e.g., R, Python, Julia, Matlab)
- A very common approach in scientific computing is to write the computationally intensive "kernel" of your algorithm in one of these lower-level languages (e.g., C, C++, Fortran), and then call that compiled code from a higher-level languages (e.g., R, Python, Matlab).

The Problem

Why is this the “two-language problem” a problem?

1. Slows development of new methods
    - Developing in C++ harder
    - Might need separate software engineer
    - If you avoid lower-level languages, your code will be slow
2. Hurts transparency of methods
    - Anyone can read Python or Matlab
    - Take a look at some C code from 80s

Julia Langauge

1. Technical computing language
2. Origin
   1. Jeff Bezanson, Stefan Karpinski, Viral Shah, & Allen Edelman
   2. Version 0.1 in 2012
   3. Current stable version is 0.6.2
   4. Expect version 1.0 by end-of-year
3. Similarities with Matlab, R, and Python
4. Free and open source (MIT license)

Julia Language Features

1. Speed
    1. Just-in-time compiler
    2. Within 1x or 2x of C and Fortran
    3. First non-C, non-Fortran language to achieve greater than 1 PetaFLOP/s on super computer (Celeste Project run on Cori)
2. Expressiveness
3. Type system
4. Integrated package manager
5. Parallelism

Performance Comparison

`<EXAMPLES IN R AND JULIA>`

Parallelism in Julia

1. Shared or distributed memory
2. Mult-threading and multi-processing
3. Functions and control structures
    1. Parallel `for` loop similar to OpenMP pragmas
    2. `pmap()` function is parallel version of `map()`
4. Data structures
    1. Distributed arrays
    2. Shared arrays
5. Add or remove processes on-the-fly from REPL
    1. `addprocs()`
    2. `rmprocs()`

Conclusion

1. Very few of us data sets so large that the problem is insurmountable
2. Usually, a beefier server or more clever algorithmic approach is all that is needed
3. Base R will do pretty well usually, since much of the "internals" are actually C, C++, or Fortran
4. If you want extremely fast code that's easy to read and write, consider Julia