The Arborist: a High-Performance Random Forest Implementation

Mark Seligman

Suiji

May 27, 2015

1 Outline

2 Introduction

3 Random Forests

4 Implementation

5 Examples and anecdotes

6 Summary and future work

Outline

- Introduction
- Random Forests
- Implementation
- Examples and anecdotes
- Summary and future work
- Q & A

1 Outline

2 Introduction

3 Random Forests

4 Implementation

5 Examples and anecdotes

6 Summary and future work

- Began as proprietary implementation of Random Forest (TM) algorithm.
- Aim was enhanced performance across a wide variety of hardware, data and workflows.
- GPU acceleration a key concern.
- Open-sourced and rewritten following dissolution of venture.
- **Arborist** is the project name.
- **Rborist** is the CRAN package.

Project design goals

- Minimize data movement.
- Language-agnostic implementation.
- Ready extensibility.
- Common source base for all spins.

Data locality

- Computers store data in hierarchy:
  - Registers.
  - Caches (L1 - L3).
  - RAM.
  - Disk.
- CPU operates on registers.
- Loading registers consumes many clock cycles, depending upon position in hierarchy.
- Performance therefore best when data is spatially (hence temporally) local.
- Similarly, loops over vectors most efficient when data in consecutive iterations separated by predictable and short(ish) strides.
- "Regular" access patterns allow compiler, and hence hardware, to do a good job.

1 Outline

2 Introduction

3 Random Forests

4 Implementation

5 Examples and anecdotes

6 Summary and future work

Binary decision trees, briefly

- Prediction method presenting a series of true/false questions about the data.
- Answer to given question determines which (of two) questions to pose next.
- Successive T/F branching relationship justifies "tree" nomenclature.
- Different data take different paths through the tree.
- Terminal (or "leaf") node in path reports score for that path (and data).
- Typically a "forest" of 100 – 1000 trees traversed and overall average (regression) or plurality (classification) reported.

Random Forests

- **Random Forest** is trademarked, registered to Leo Breiman (dec.) and Adele Cutler.
- Observational data ("predictors") can include both numerical and categorical values.
- Predictors sampled randomly throughout training - separately chosen for each node.
- Trains on randomly-selected subset of the observations ("bagging").
- Validation on held-out subset.
- Independent prediction on separately-provided test sets.

Training

- Intuitively, tree nodes represent (progressively smaller) row sections of the data.
- Root node represents data contained in all in-bag rows.
- Tree grows by successively splitting nodes according to an information metric (Gini).
- Split determines bipartition of the data into two row sections.
- If information gain below some threshhold, node does not split: leaf.
- At any point in the process, frontier represents a (highly disconnected) partition of the bagged data, as row sections.
- Leaf's score obtained directly from response values evaluated at its representative row indices.
- Implementation need not actually build these subsets at each step, but helpful to track corresponding index sets.

Node splitting

- Performed over (separately) sampled subset of the predictors.
- Methods differ for numerical and factor predictors.
- Both guided by predictor *rank* to traverse node's index set.
- Looking for predictor maximizing information metric.
- Numerical predictor p: splitting criterion is an order relation.
  - E.g., $p <= 3.2$? branch left : branch right.
  - Information metric: walk observations in *predictor* rank order.
  - $\#$ computations scales with size of index set.
- Factor predictor q: splitting criterion determined by subset membership.
  - E.g., $p \in \{3, 8, 17\}$? branch left : branch right.
  - Information metric: walk predictor *runs*, in any order, over *power set*.
  - $\#$ computations scales with $2^{\#runs}$: subsample if $\#$runs $>\sim 10$).
- Splitting criterion defines L/R row subsections for descendant nodes.

Node splitting: performance

- Splitting is "embarrassingly parallel": nodes × predictors.
- **However,** ranks corresponding to a node's row indices vary with predictor.
- Naive solution is to sort observations at each splitting step.
  - Approach used by RF package. Does not scale well.
- Could also walk pre-sorted list, accumulating per-node state by looking up row index.
  - Original Arborist approach. No data locality.
  - Large state budget.
- "Restaging": maintain separately-sorted state vectors for each predictor, by node.
  - Begin with pre-sorted list, and (stable) bipartition at each node.
  - Current Arborist approach. Data locality improves with tree depth.
  - Only modest amount of state to move: 16 bytes.
  - Splitting becomes quite regular: next datum prefetchable by hardware.
  - *intra-node* restaging parallelizable on SIMD hardware.

Aside: performance is data-dependent

- As with linear algebra, the appropriate treatment depends on the contents of the (design) matrix: SVD, for example.
- E.g., regression has regular access patterns and tends to run very quickly.
- Constraints in response or predictor values - may benefit from numerical simplification.
- Will ties play a significant role? - sparse data can train very quickly.
- Custom implementations rely heavily on the answer to such questions.
- It therefore makes sense to strive for extensibility and ease of customization.

1 Outline

2 Introduction

3 Random Forests

4 Implementation

5 Examples and anecdotes

6 Summary and future work

Organization

- Compiled code with various language front-ends.
- **R** is the driving language, but **Python** now under active development.
- Front-end "bridges" wherever possible: **Rcpp**.
- Minimal use of front-end call-backs: PRNG, sampling.
- Common code base also supports GPU version, largely as a subtyped extension.

Look and feel

- Guided by **randomForest** package. Many options the same, or similar.
- Supports only numeric and factor data: leaves type-wrangling to the user.
- Predictor sampling uses *predProb* (Bernoulli) - unlike *mtry*'s (w/o replacement).
- Breadth-first implementation; introduces specification of terminating level.
- Introduces information-based stopping criterion, *minRatio*.
- Unlimited (essentially) factor cardinality.
- Several useful **randomForest** options remain to be implemented.

Distinguishing features

- Decoupling of splitting from row-lookup: restaging + highly regular node walk.
- Both stages benefit from resulting data locality and regularity.
- Restaging maintained as stable partition (amenable to SIMD hardware).
- Training produces lightweight, serial "pre-tree".
- Rich intermediate state: e.g., frontier maps reveal quantiles.
- Internal permutation of response vector permits "loopless" (i.e., from the user's perspective) resampling.

Training wheels, early experience

- Begin with **R** front end.
- Compare performance with **randomForest**.
- "Medium" data
  - Roughly 10^3 to 10^8 rows.
  - $\#$ predictors: $>>$ $\#$ cores.
- Regular data, especially regression/numerical.
- Roughly similar cardinalities across factor predictors.
- Noticing speedups as row counts approach $500 - 1000$.

1 Outline

2 Introduction

3 Random Forests

4 Implementation

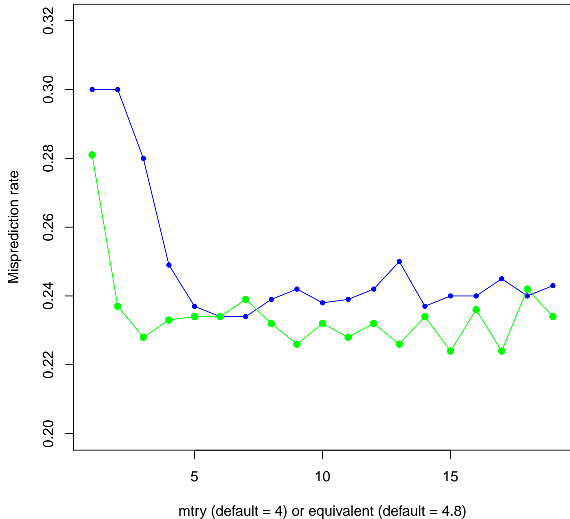
5 Examples and anecdotes

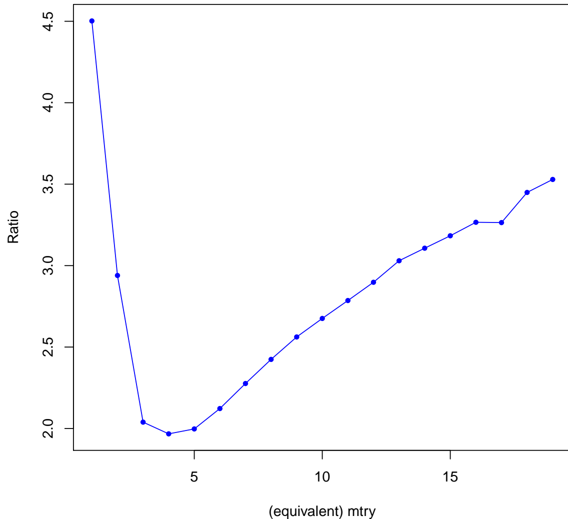
6 Summary and future work

German credit-scoring data set[1]

- Binary response, 1000 rows.
- 7 numerical predictors.
- 13 categorical predictors, cardinalities range from 2 to 10.
- Graphs to follow compare misprediction, execution times at various predictor selections.
- Accuracy as function of “predProb” (blue) appears to track that of “mtry” (green) well.
- Performance of Rborist generally 2-3 $\times$, in this regime.

[1] Lichman, M. (2013). UCI Machine Learning Repository [http://archive.ics.uci.edu/ml]. Irvine, CA: University of California, School of Information and Computer Science.


Misprediction: predProb (Rborist) vs. mtry (RF)
Misprediction rate
0.20
0.22
0.24
0.26
0.28
0.30
0.32
5
10
15
mtry (default = 4) or equivalent (default = 4.8)


Execution time ratios: randomForest / Rborist
Ratio
2.0
2.5
3.0
3.5
4.0
4.5
5
10
15
(equivalent) mtry

Instructive user account

- Disconcerting item encountered in recent R-Bloggers post:
  - "several other random forest implementations have been tested (..., Rborist R package, ...) but all of them proved slow and/or unable to scale to the larger sizes."[2]
- Subsets of airline flight-delay data set tested on various RF.
- 8 predictors; various row counts: $10^4, 10^5, 10^6, 10^7$ rows.
- Reviewer using 60GB cloud instance, 32 cores. (2 nodes?)
- Behavior reproduces on 4-core, 8GB machine with CRAN verison (0.1-0).
- Problem had already been repaired with Beta (0.1-1).
  - 10^5 rows: no swapping, all 4 cores busy.
  - 10^6 rows: swapping after about 200 trees.
  - Performance suggests on a par with Spark results reported by author.

[2] "Benchmarking Random Forest Implementations", Szilard Pafka, DataScience.LA, May 19, 2015.

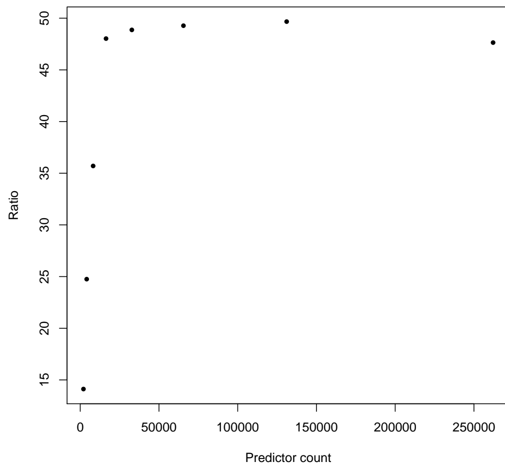
Post mortem

- Appears to have been a problem with large-cardinality sampling scheme.
- Predictor-level parallelization tops out at 8 cores: cloud server nodes often have many more.
- Spreading parallelization to predictor/node pairs would easily consume available cores.
- *n log n* approximations available for high-cardinality, binary data; Arborist currently uses (combinatorial) subsampling.
- Arborist has so far emphasized *multicore* but *multinode* not so much.
- Despite the salient problem having been repaired, this remains a meaty test case suggesting attainable improvements.

GPU: pilot study with University of Washington team

- GWAS data provided by Dept. Global Health.
  - 100 samples, up to $\sim 10^6$ predictors.
  - Binary response, purely categorical predictors with cardinality $= 3$.
- Bespoke CPU and GPU versions spun off for the data set.
- Each tree trained (almost) entirely on GPU.
- Results illustrate potential - for highly regular data sets.
- Drop-off on right is an artefact from copying *data.frame*.

CPU vs. GPU: execution time ratios of bespoke versions



New package

- *ad hoc* implementation did not scale: required rewriting to apply more generally.
- Even so, data is typically not regular.
  - Mixed predictor types and multiple cardinalities present load-balancing challenges.
  - On-GPU transpose appears necessary to support row- and column-major representations.
  - Much easier to split on CPU.
- For now, shifting focus to (very regular) task of restaging.
- Restaging now implemented as stable parition via parallel *scan*.
- Nvidia engineers concurred with this solution, anticipate good scaling.
- Software pipeline: masking transfer latency by overlapping training of multiple trees.
- Upcoming **Curborist** ($\equiv$ **Cuda Rborist**) package.

1 Outline

2 Introduction

3 Random Forests

4 Implementation

5 Examples and anecdotes

6 Summary and future work

Summary

- Only a few rigid design principles:
  - Constrain data movement.
  - Language-agnostic core implementation.
  - Common source base.
- Plenty of opportunities for improvement.
- Load balancing appears to be lowest-hanging fruit right now.

Short/medium-term goals

- Beta version 0.1-1 to CRAN; improved testing and documentation.
- Roll-out of **Curborist**.
- *Internal* support for sparse designs.
- Smarter multicore scheduling, including predictor $\times$ node.
- Restore *mtry* semantics, at least for small predictor $\#$.
- Hierarchical parallelism.
- *NA* handling.

Longer term

- Out-of-memory support.
- Generalized dispatch, fat binaries.
- Plugins for other splitting methods.
- Internalize additional workflows.

Acknowledgments

- Stephen Elston, Quanta Analytics.
- Abraham Flaxman, Dept. Global Health, U.W.
- Authors and maintainers of **Rcpp** and **RcppArmadillo**.
- Contributors to **R-bloggers**.