Taiga: overview

John Alan McDonald, Kristina Lisa Klinkner

2017-12-21

Abstract

Taiga is a machine learning library written in Clojure.

As of December 2015, Taiga offers a general pattern for random forest models, with specific implementations for least squares regression, binary classification, and binary class probability estimation. Random forest regression/classification is widely used in machine learning, performing 'remarkably well' on many data sets (p.590[17]). It is also arguably the simplest useful machine learning algorithm — so simple that a working implementation can be easily produced in a day or 2.

As of December 2015, Taiga is about 700 lines of Clojure. It is based on an earlier tutorial implementation of random forest regression, in about 200 lines of Clojure, which was in fact written over the long New Year's weekend of 2012.

Taiga represents both data and models in orders of magnitude less memory than the widely used R randomForest[25] package. Taiga is much faster than R (*TODO: some real measurements.*), running on a single core, and its speed scales roughly linearly with the number of cores.

Contents

1 Introduction

The purpose of this document is to lead you thru implementing random forest regression in Clojure.

As it happens, I'm not a fan of either random forests or Clojure, so you might reasonably ask 'Why?'.

I am a fan of tree-based methods and of Lisp. and both random forests and Clojure can be useful introductory instances of the larger class of tools they represent.

1.1 Random forests

Among tree-based methods, random forests are probably the most widely used. They perform 'remarkably well, with very little tuning required' (p.590[17]) for classification and least squares regression. Random forest regression/classification is also arguably the simplest useful machine learning algorithm — so simple that, as you will see, a working implementation can be easily produced in a day or 2.

On the other hand, random forests are usually out-performed —just a little— by methods that are —just a little— more complicated, like gradient boosting. A more important limitation is related to the fact that there is no obvious way to modify, for example, the algorithm for random forest L2 (least squared error) regression to optimize other cost functions.[1]

On the 3rd hand, random forests do work well enough for classification and least squares regression— two important problem classes.

Even if you never use the resulting code, working thru this exercise will show you exactly how random forests work, make it easier to understand the differences between random forests and alternatives, and prepare you to make informed choices between 3rd party implementations.

[1] The standard approach to L1 (least absolute error) regression grows a forest to minimize L2 cost, and then post-processes it to get an estimate of the 'median'.[**Meinshausen:2006:QRF:1248547.1248582**]

In addition, the resulting code may in fact be a useful starting point for a production-ready implementation. The most widely used open source implementations (eg [25]) have arbitrary limitations (like max 32 categories) and are slower and use much more memory than they need to, and fail to take advantage of how embarassingly easy it is to grow random forests in parallel. The implementation you produce will have none of these limitations.

1.2 Clojure

Doing statistics/machine learning in Lisp is not a new idea. Depending on how you draw the boundary between AI and machine learning, you might even argue that Lisp was invented to do machine learning. There was quite a bit of work on statistical computing in Lisp in the 1980s-1990s [9, 27, 30, 41, 42], including R [31], which began as a project to develop statistical software in Scheme [19]. R later morphed into an open-source implementation of the S language from Bell Labs, itself heavily influenced by Lisp [2, 6, 7]. More recently, the R and S authors have discusssed the limitations of those languages and suggested starting over from a modern Lisp implementation (eg [20, 42]).

The features of Lisp that make it suitable for machine learning include:

1. Every *thing*[2] is dynamic, meaning it can be created/defined/modified at run time.
2. Functions, classes, etc., are first class things. (meta-object protocol implies 'no reflection'...).
3. High performance.
4. Recursion.
5. TODO: more...

As a Lisp, Clojure [11, 12, 16, 32, 43] is something of a mixed bag. Some aspects are very well done, permitting elegant, expressive, compact, safe, fast code, especially in the context of concurrency. Other important aspects of the language are a patchwork of inconsistent afterthoughts.[3] Fortunately, like any Lisp, it's at least possible for a developer to replace/hide/work-around bad decisions by the language designer (see Exercise ??).

[2] I use 'thing' rather than 'object' to avoid 'object' = 'instance of Java class' confusion.

The features of Clojure that make it suitable for implementing random forests include:

1. The fundamental data structure is the lazy, immutable, possibly infinite sequence.
2. Software transactional memory (STM) enables safe concurrent access to shared mutable state.
3. Local type hints result in code that runs as fast as possible (on the JVM).
4. Trivial to call Java libraries from Clojure; almost as easy to call Clojure from Java.

Because of features 1 and 2, Clojure is well suited to safely expressing concurrent recursive algorithms like random forests. Because of features 3 and 4, Clojure code can be made to run as fast as the fastest Java implementation, while sacrificing a minimal amount of clarity, simplicity, and compactness.

The R randomForest package is 1626 lines of R and 2317 lines of C (using wc -l to count all lines). My Clojure version is 310 lines (using wc -l as with randomForest), or 150 lines of actual Clojure expressions.

This comparison is a bit unfair, because my code does only random forest regression. Adding support for 2-class classification is only a couple lines of code, but k-class classification could easily double the size. Support for measuring variable importance, partial influence plots, etc., might add another few hundred lines. Thus a Clojure implementation with equivalent functionality to randomForest might be 1000 lines (vs randomForest's 4000 lines of R+C).

The (unoptimized) Clojure implementation presented is faster than R randomForest regression on a single core, and get faster roughly linearly in the number of cores.

[3] Many ~~forms of Government~~ *JVM languages* have been tried, and will be tried in this world of sin and woe. No one pretends that ~~democracy~~ *Clojure* is perfect or all-wise. Indeed, it has been said that ~~democracy~~ *Clojure* is the worst ~~forms of Government~~ *JVM language,* except all those other ~~forms~~ *JVM languages* that have been tried....[8]

There are at least a couple open source machine learning libraries in Clojure: Incanter [26], which is something like 'R in Clojure' and clj-ml [14], which is something like 'Weka in Clojure' [40], so there is pre-existing code that can be used for the really difficult tasks, like drawing scatter-plots.

2 Predictive modeling with trees

An example of a predictive modeling problem is transit time estimation: For shipment planning, we need to estimate how long it will take if we ship a package, from one of a number of possible origins, using one of a number of possible shipping methods, to a given destination.

A predictive modeling problem requires [13, 17]:

- a space of the knowns $\mathrm{x} \in \mathbb{X}$, which will be the input to the prediction.

  For transit times, x is a tuple combining attributes derived from the package, origin, ship time, destination, etc.

- a space of the unknown $\mathrm{y} \in \mathbb{Y}$, which we are trying to predict.

  For transit times, y is some time quantity, like slam-to-delivery, manifest-to-clockstop, etc. It might also be a probability distribution over such times.

- a family $\mathbb{F}$ of functions $\mathbb{X} \mapsto \mathbb{Y}$, from which we choose an element $\mathrm{f} \in \mathbb{F}$ that we will use to predict $\hat{\mathrm{y}} = \mathrm{f}(\mathrm{x})$.

  For transit times, a simple (too simple, guaranteed to fail) model family would be a large table with cells containing estimated transit times, indexed by all the possible combinations of origin, ship method, and destination.

Most methods for predictive modeling can be characterized by a cost $\mathcal{C}()$ with which we measure the quality of *future* predictions. In *supervised* learning, we eventually get to observe the true value of y, so cost functions typically measure the goodness of fit of observed y and the prediction f(x). Most often this is done by simply summing pairwise distances: $\mathcal{C}(\mathrm{f}, \mathcal{S}) = \sum_{(\mathrm{x}_i, \mathrm{y}_i) \in \mathcal{S}} d(\mathrm{y}_i, \mathrm{f}(\mathrm{x}_i))$, where $\mathcal{S}$ is a set of future pairs $(\mathrm{x}_i, \mathrm{y}_i)$ and d is a distance on $\mathbb{Y}$. Examples:

L2 Regression The goal, in the transit time problem, would be to predict the mean transit time given x. $\mathbb{Y}$ is the real numbers. The standard distance is $d(\mathrm{y}, \mathrm{f}(\mathrm{x})) = (\mathrm{y} - \mathrm{f}(\mathrm{x}))^2$ and $\mathcal{C}(\mathrm{f}, \mathcal{S}) = \sum_{(\mathrm{x}_i, \mathrm{y}_i) \in \mathcal{S}} (\mathrm{y}_i - \mathrm{f}(\mathrm{x}_i))^2$.

Quantile Regression The goal is to predict $q(p)$ of the transit time given x, at some fixed surety level p. $\mathbb{Y}$ is the non-negative integers. The quantile regression distance function is $d_p\left(\mathrm{y}, \mathrm{f}(\mathrm{x})\right) = p * |\mathrm{y} - \mathrm{f}(\mathrm{x})|^{+} + (1-p) * |\mathrm{y} - \mathrm{f}(\mathrm{x})|^{-}$, where $|z|^{+} = |z|$ if $z > 0$, and zero otherwise, and $|z|^{-} = |-z|^{+}$.

Density Estimation The goal is to predict a probability measure for transit times, given x. $\mathbb{Y}$ is the space of probability measures the non-negative integers . We don't get to observe the true probability measures, but we can treat an observed transit time t_i as a probability measure that puts mass 1 on t_i : $\mathrm{y}_i = \delta(t_i)$. We can use a distance on probability measures, like Earthmover[24, 33, 34], so $\mathcal{C}\left(\mathrm{f}, \mathcal{S}\right) = \sum_{(\mathrm{x}_i, t_i) \in \mathcal{S}} W_1(\delta(t_i), \mathrm{f}(\mathrm{x}_i))$

The obvious, but naive, approach to supervised learning is to collect a set of historical training pairs $\mathcal{T} = \{(\mathrm{x}_i, \mathrm{y}_i)\}$ and choose f to minimize the cost over $\mathcal{T}$:

$$\mathrm{f} = \underset{\mathrm{f} \in \mathbb{F}}{\operatorname{argmin}} \sum_{(\mathrm{x}_i, \mathrm{y}_i) \in \mathcal{T}} d\left(, \mathrm{y}_i, \mathrm{f}(\mathrm{x}_i)\right) \tag{2.1}$$

However, this almost always leads to bad predictions Empirically, this puts too much trust in the training data, reproducing noise as well as signal. Fits that are *regularized* — biased towards models that are smoother or more conservative, in almost any reasonable way, consistently do better.

Two approaches to regularization are worth thinking about here: cross-validation and bagging. I'll describe these in the context of tree-based least squares regression models (CART and random forests) that are generally similar to the current transit time regionalization algorithm. I'll use the same artificial data, shown in figure 2.1, to illustrate both. The true y in this case is a slightly elliptical parabolic surface over 2 variables, x_0 and x_1 (which could be thought of as latitude and longitude). The training data consists of 10,000 points sampled uniformly from the $(-1.0, 1.0) \times (-1.0, 1.0)$ domain with $\sigma = 0.5$ gaussian noise added.

2.1 Cross validation (CART)

CART [4, 17] is probably the most successful single regression tree method. CART produces a binary tree which recursively partitions $\mathbb{X}$ into rectangular regions, each of which is fit by a constant value. The model function $\mathrm{f}(\mathrm{x}_i)$ finds which leaf/rectangular region of the tree x_i is in, and returns the constant associated with that leaf.

CART has 2 phases: (1) a greedy tree growing phase common to most tree-based methods and (2) a cross-validated pruning phase, which is what makes CART special.

2.1.1 Greedy tree growth

The growing phase is a optimizes the tree to minimize a cost function over a training data set. In the case of least squares regression: $\mathcal{C}\left(\mathrm{f}, \mathcal{S}\right) = \sum_{(\mathrm{x}_i, \mathrm{y}_i) \in \mathcal{S}} (\mathrm{y}_i - \mathrm{f}(\mathrm{x}_i))^2$, where $\mathcal{S} = \{(\mathrm{x}_i, \mathrm{y}_i)\}$is a set of training pairs.

To begin, we take the whole training set $\mathcal{S}$, and consider every possible way of partitioning $\mathcal{S} = \mathcal{S}_0 + \mathcal{S}_1$ using any one of the attributes in $\mathbb{X}$. For numerical attributes, this means searching over predicates like $x_1 \geq 0.74$. For categorical attributes, we need to consider all possible subsets of the categories. For any contemplated split, we compute $\mathrm{f}_{\mathcal{S}_k}() = \operatorname{argmin}_{\mathrm{f}} \mathcal{C}(\mathrm{f}, \mathcal{S}_k)$ for each of the 2 possible subsets. In the most common case, we only consider functions that are constant over the subset of the domain, and $\mathrm{f}_{\mathcal{S}_k}() = \operatorname{mean}\{\mathrm{y}_i : (\mathrm{x}_i, \mathrm{y}_i) \in \mathcal{S}_k\}$We take the split that minimizes $\mathcal{C}(\mathrm{f}_{\mathcal{S}_0}, \mathcal{S}_0) + \mathcal{C}(\mathrm{f}_{\mathcal{S}_1}, \mathcal{S}_1)$. The result of this process with our synthetic data is shown in figure 2.1. Then we repeat on the 2 leaves generated by the first split, whose result is shown in figure 2.2.

This process continues until we run out of data, where 'run out of data' means the training examples in a given node are either too few or too similar to split meaningfully. The result is shown in figure 2.3

The fit is visually noisy and more important, could be shown by experiments to give poor predictions of y given x.

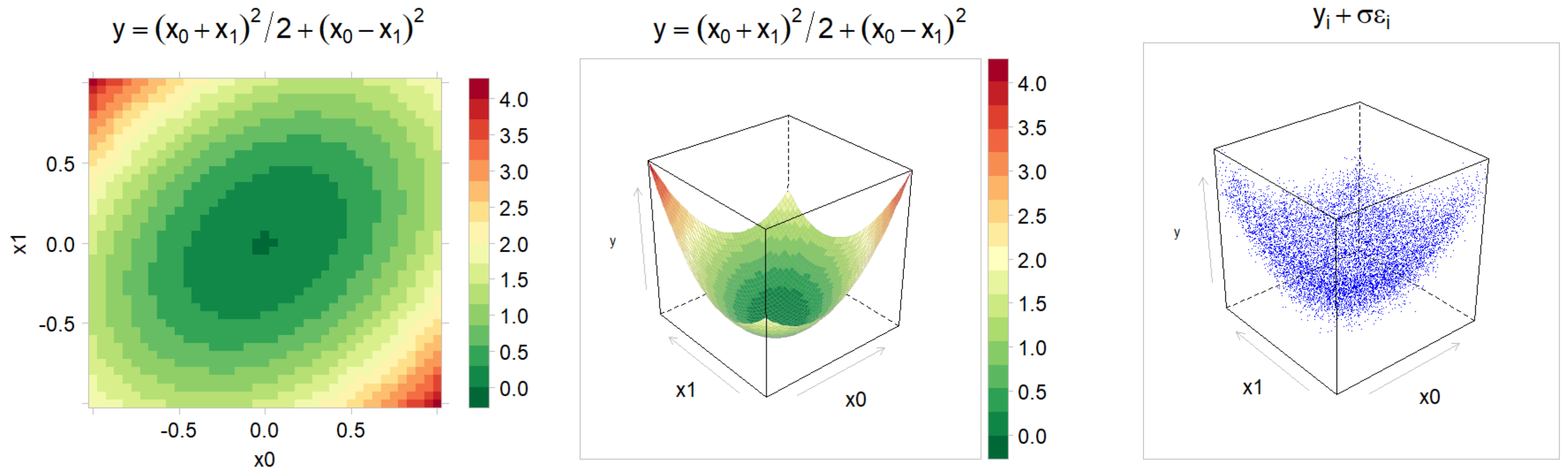


Figure 2.1: Synthetic data for regularization examples

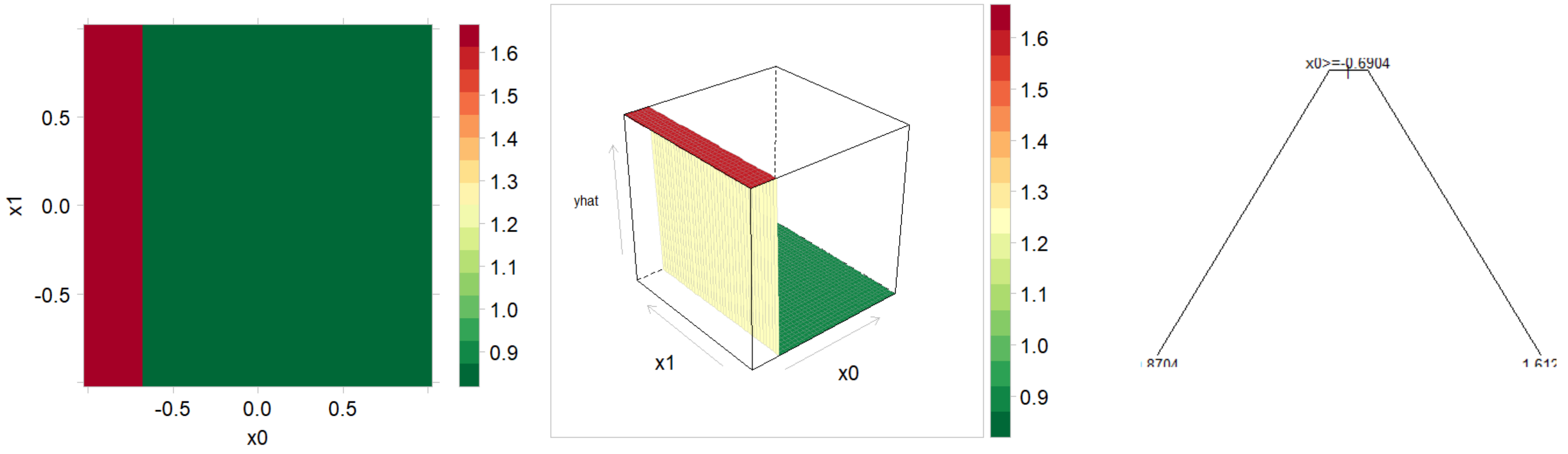


Figure 2.1: First split

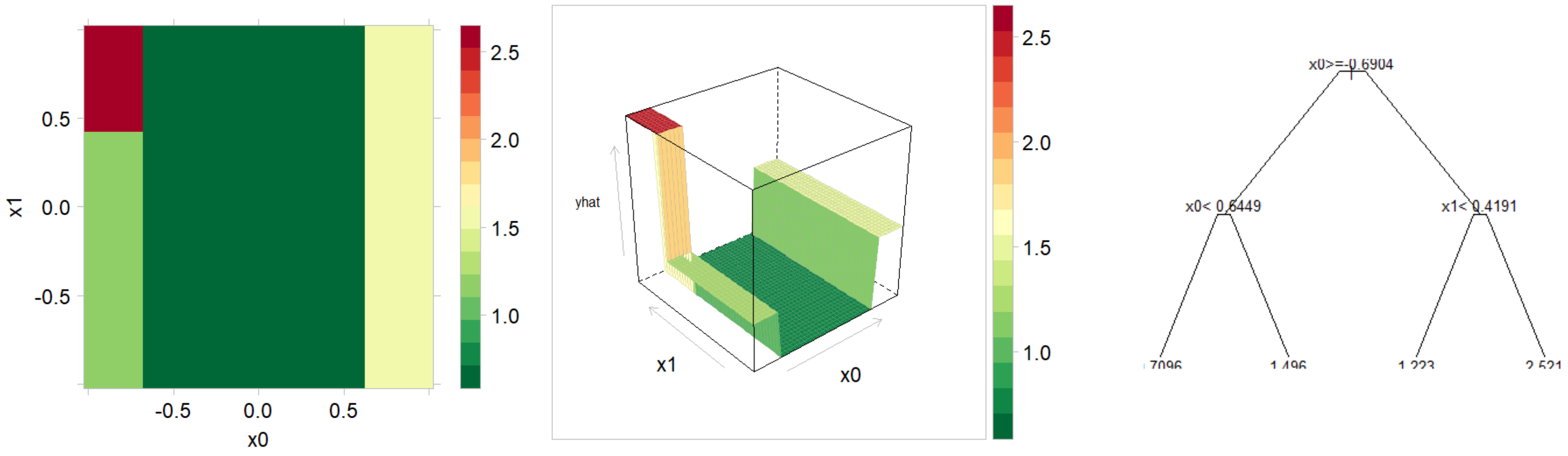


Figure 2.2: First 3 splits

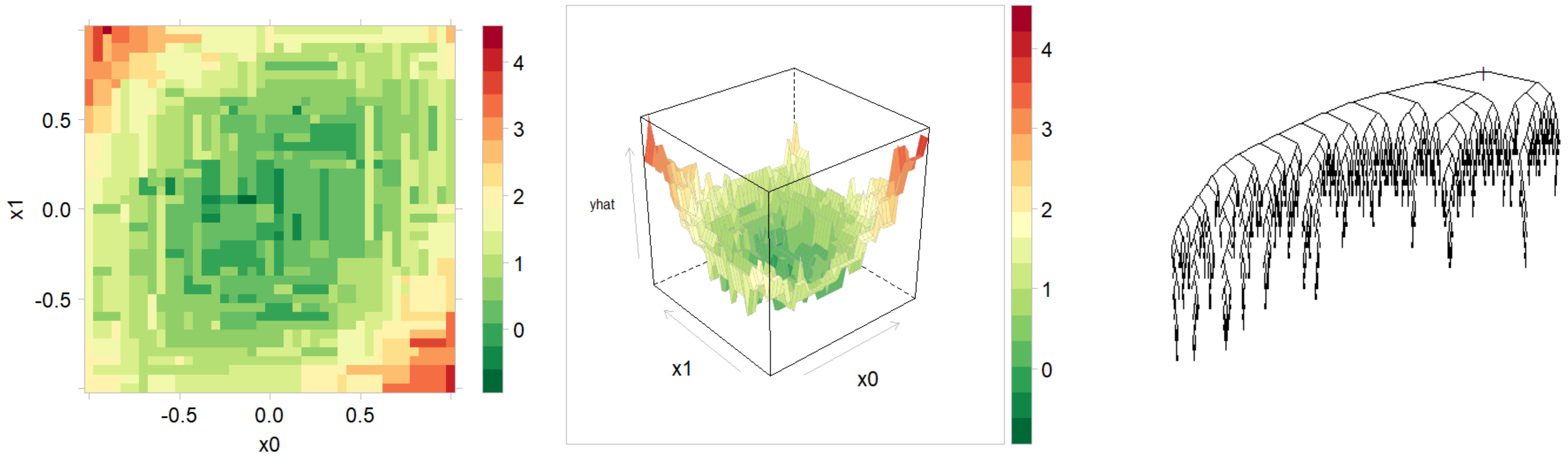


Figure 2.3: Unpruned tree

2.1.2 Cross-validated pruning

The key idea in CART was to use cross-validated pruning to fix the tree shown in figure 2.3.

Cross-validated pruning is a special case of cross-validated regularization, which can be applied to almost any predictive modeling method. The basic idea is to split the training data into 2 parts, training $\mathcal{T}_0$ and test $\mathcal{T}_1$. We add a complexity term $\mathcal{S}()$ to the cost that biases us towards simpler models. In the case of tree-based models, this may be as simple as the number of nodes in the tree. We have a parameter λ that determines the relative tradeoff between goodness of fit and complexity.

For any given λ we can find the optimal model by purely greedy optimization:

$$\mathrm{f}_\lambda = \underset{\mathrm{f}\in\mathbb{F}}{\mathrm{argmin}} \left[\lambda\mathcal{S}(\mathrm{f}) \ + \sum_{(\mathrm{x}_i,\mathrm{y}_i)\in\mathcal{T}_0} d\left(\mathrm{y}_i, \mathrm{f}(\mathrm{x}_i)\right) \right] \tag{2.1}$$

We then need to choose an optimal value for λ. We do this by optimizing the unregularized goodness of fit over the test data:

$$\mathrm{f} = \underset{\mathrm{f}_\lambda}{\mathrm{argmin}} \left[\sum_{(\mathrm{x}_i,\mathrm{y}_i)\in\mathcal{T}_1} d\left(\mathrm{f}_\lambda(\mathrm{y}_i, \mathrm{x}_i)\right) \right] \tag{2.2}$$

It's usually not feasible to actually find the optimal f_λ for all values of λ. Practical tree-based algorithms, like CART first find the naive, unregularized $\mathrm{f}_{\lambda=0}$ and then construct a heuristic sequence of smaller trees corresponding to increasing values of λ. In CART this is done by collapsing splits from the bottom up, always collapsing the split with the least improvement $\mathcal{C}(\mathrm{f}_\mathcal{S}, \mathcal{S}) - \mathcal{C}(\mathrm{f}_{\mathcal{S}_0}, \mathcal{S}_0) - \mathcal{C}(\mathrm{f}_{\mathcal{S}_1}, \mathcal{S}_1)$. The cost as a function of tree size for our example is shown in figure 2.4.

CART then picks the smallest tree such that no larger tree shows a significant improvement. The result is in figure 2.5. The resulting function is a pretty coarse approximation to the underlying parabola.

2.2 Bagging (Random Forests)

Empirically, CART regression models perform about as well as any alternative. However, there are at least 2 issues with single tree methods that led to the development of forest-based methods. One, the crudeness of the step function approximation shown in figure 2.5 is unsatisfying, especially in contexts where one believes y is a smooth function of x. Perhaps more important, single tree models are inherently unstable, in the sense that small changes in the training data result in completely different partitioning trees. The resulting step function changes much less, but small regions close to the boundaries of the steps can experience large jumps.

To prevent this, forest-based methods compute many treesf_i and then take as the final model $\mathrm{f} = \text{ average}\left\{\mathrm{f}_\mathrm{i}\right\}$, where the average operation is defined appropriately for the problem. For traditional regression problems, we just take the mean of the predictions of the trees in the forest. In the transit time problem, each f_i might be a histogram, or, equivalently, a cdf, and the average would be just the numerical mean of the cdf or histogram functions.

Random forest is a particularly simple forest-based method, combining bagging with randomly de-optimized tree growth.

The basic idea of bagging is to simulate what we might see in future data by collecting n independent bootstrap (with-replacement) samples $\mathcal{T}_i$ from the complete training data $\mathcal{T}_r$. We fit a model f_i via some base learner to each bootstrap sample set $\mathcal{T}_i$, and average the results.

In practice, bagging alone is not very successful, unless it's coupled with some further randomization or constraint that makes the individual f_i less greedy.

Random forests [3, 17] uses the same greedy optimization as CART to grow its trees, except that for each split it restricts the search to a random subset of the attributes. Random forests are

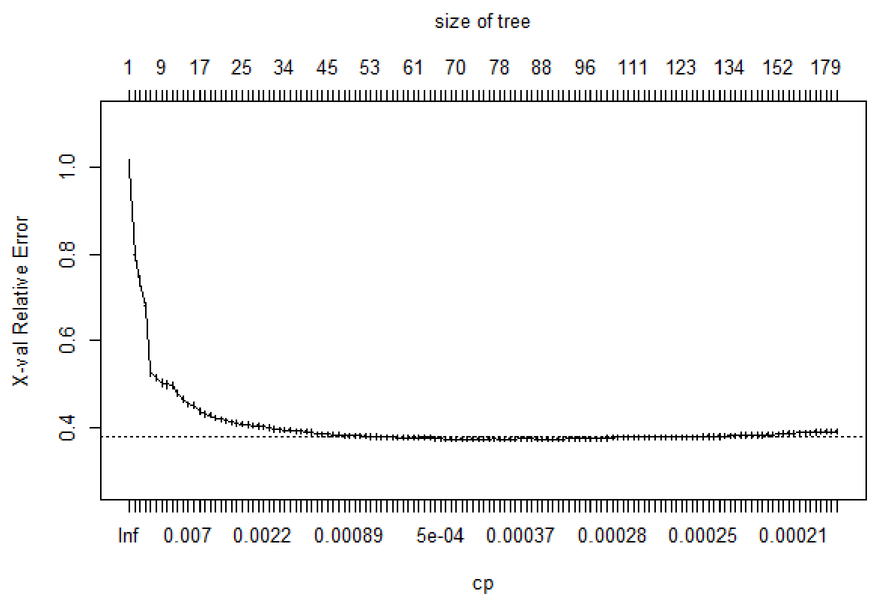


Figure 2.4: Cross-validated accuracy vs tree size

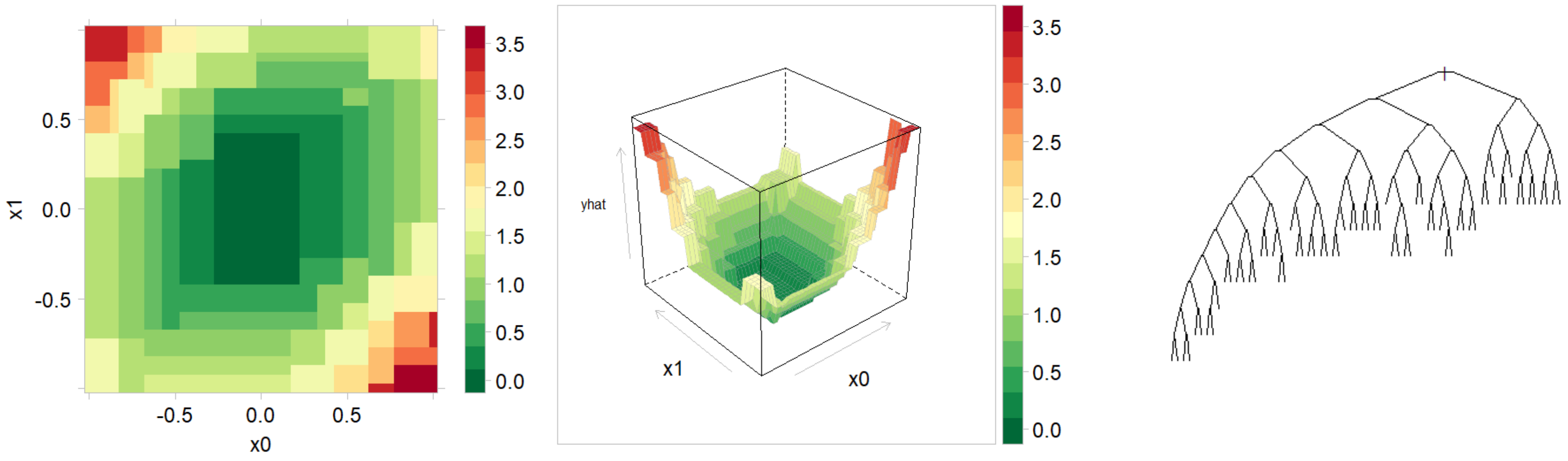


Figure 2.5: Pruned tree

generally regarded as the most successful method for regression and classification, but the reasons for their success are not fully understood.

Figures 2.1 thru 2.3 illustrate the process of creating a random forest on the same synthetic data. Figure 2.4 shows how prediction accuracy depends on the number of trees in the forest.

The final fit is much smoother than the CART model, but the increase in overall accuracy over CART is relatively small. One reason people tend to prefer random forests is that the error in random forest models tends to be evenly distributed over the domain, whereas in CART models it's more likely to be concentrated near the boundaries of the leaf rectangles.

3 Implementing random forest regression

The goal of this section is to walk you thru an implementation of random forest regression.

If you are comfortable with Clojure and generally familliar with decision trees, you should be able to get a working version in a day or 2. At this point, we are looking for simplicity and generality, rather than speed. The resulting code should be not much longer than the high level description in Algorithm 3 (essentially Algorithm 15.1 from [17]), except for steps 2b (section 3.4) and 2c (section 3.3), where the details have obviously been omitted.

```
Random Forests for Regression or Classification.
Training data $\mathcal{T} = \{(\mathrm{x}_i, y_i);\ i = 1 \ldots N_{\text{records}}\}$.
For $k = 1$ to $N_{\text{trees}}$:
  1. Draw a bootstrap sample $\mathcal{T}_k$ from the training data.
  2. Grow a random-forest tree $f_k$ from each bootstrap sample $\mathcal{T}_k$, by recursively repeating the
     following steps for each terminal node of the tree, until the minimum node size $N_{\text{min}}$ is
     reached.
       (a) Select $m$ attributes at random from the total $p$ attributes.
       (b) Pick the best attribute/split-point among the $m$.
       (c) Split the node into two daughter nodes.
Return the ensemble $\mathcal{F} = \{f_k\}$ .
Prediction at x:
  1. Regression: Use ensemble mean $f_{\mathcal{F}}(\mathrm{x}) = \frac{1}{\#\mathcal{F}} \sum_k f_k(\mathrm{x})$.
  2. Classification: Use ensemble mode $f_{\mathcal{F}}(\mathrm{x}) = \mathrm{argmax}_c\left(\#\{f_k(\mathrm{x}) = c\}\right)$
  3. Probability: Use ensemble vote fraction.
```

We are going to do this backwards/top-down, so we start with general ensemble (map-reduce) models:

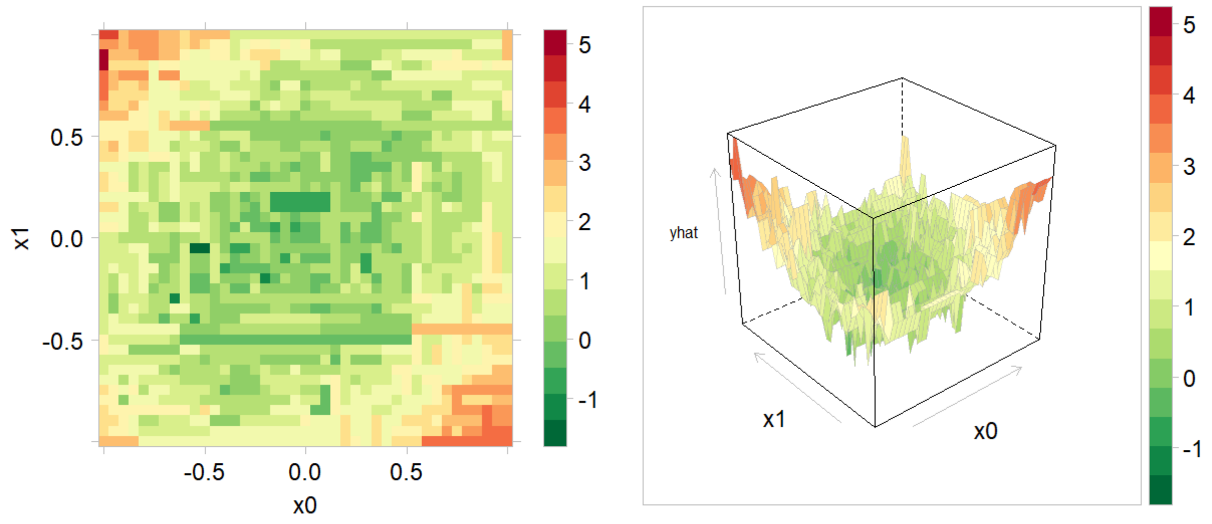


Figure 2.1: One tree random forest

3.1 Ensemble models

Random forests are a special case of additive models: $f(\mathrm{x}) = \sum_k f_k(\mathrm{x})$. In the random forest case, the terms, f_k, happen to all be regression trees, but there's no reason to impose that restriction on our additive model implementation. A simple additive model implementation in Clojure is just a few lines of code (Listing 1), but it demonstrates fundamental features of Lisp: higher order functions, lexical scoping, and closures.

```
(defn sum-model [terms]
  (let [f (apply juxt terms)]
    (fn [x] (reduce + (f x)))))
```

Listing 1: Sum additive model

`sum-model` takes a sequence of term functions as its single argument, and returns a new function that computes the sum of the values of the terms applied to the same argument `x`. Functions that create new functions from old are sometimes referred to as 'higher order functions', though there is really nothing special about them.

We first use the Clojure (higher order) function `juxt`, which creates a function that returns the sequence of the values of the functions that are its arguments. `juxt` is defined for calls like

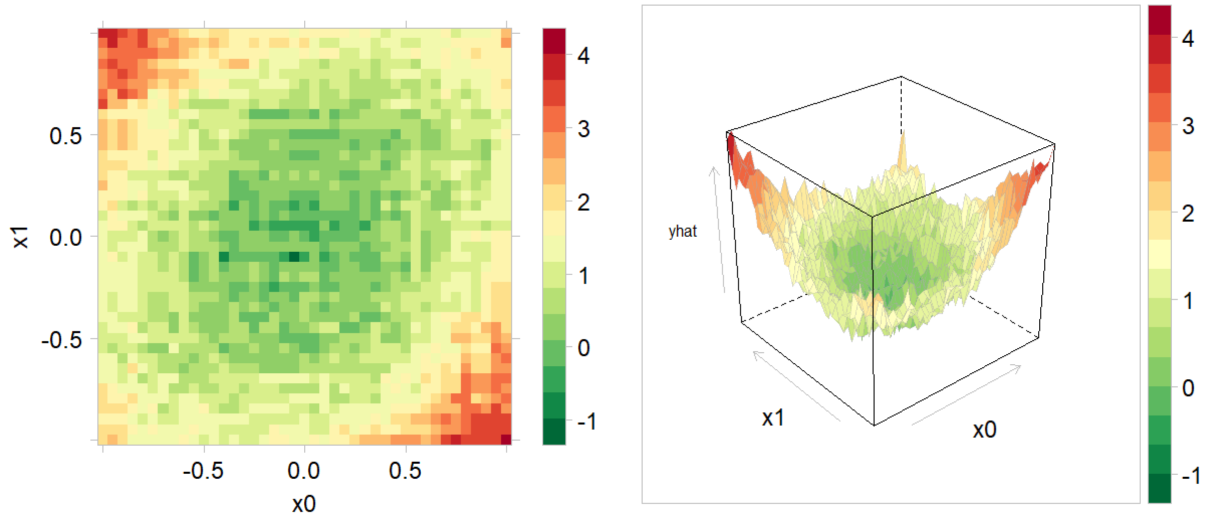


Figure 2.2: Four tree random forest

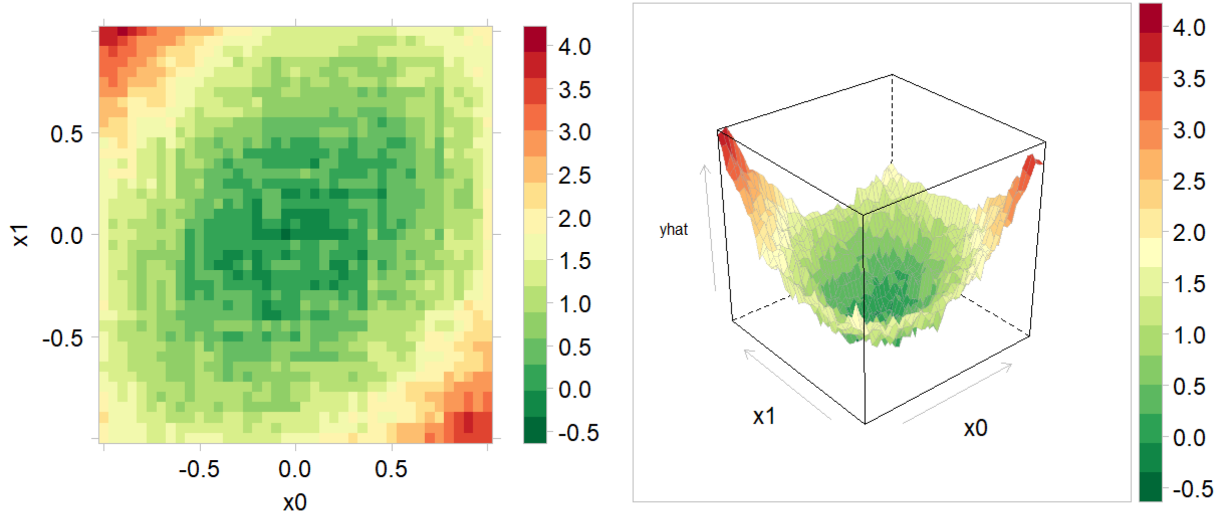


Figure 2.3: Complete random forest

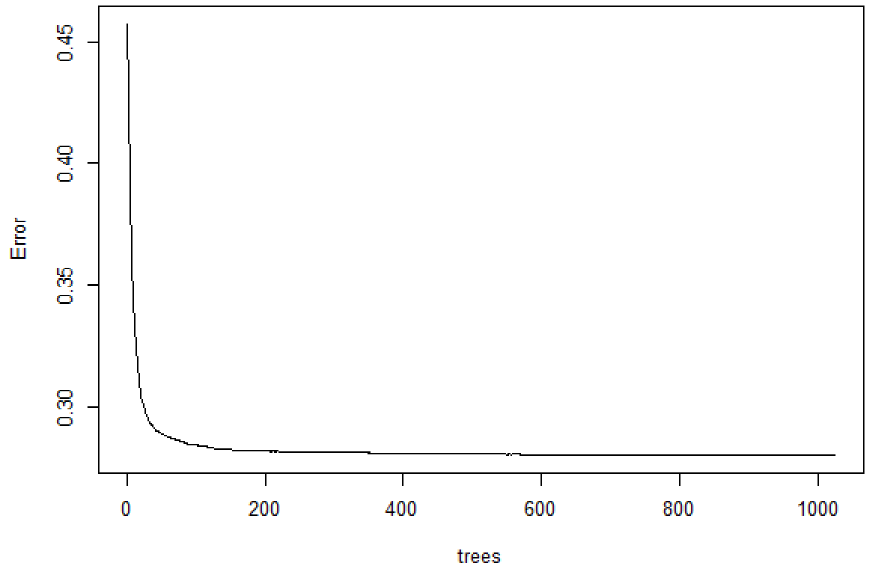


Figure 2.4: Error on test data by number of trees.

`((juxt f g) x) -> [(f x) (g x)]`, so we use `apply` to use it on a sequence of functions.

The value of the `juxt` call is bound to the symbol `f`, which is referenced in the anonymous function, defined by the `(fn [x] (reduce...))` expression, which use reduce to compute the sum of the sequence of values returned by `f`.

The anonymous function returned by `sum-model` is an example of what's often called a 'closure', because it uses the value of a binding visible in the lexical scope at function definition time, even though, in some sense that binding disappears once the call to `sum-model` returns.

3.2 Bagging

One way to generate an additive model is by *bagging* (bootstrap aggregation). We generate an ensemble of slightly different models by applying a base learner to bootstrap samples from the training data, and then combine predictions from the individual models to get the ensemble prediction.

Note that bagging doesn't require the base learner to return decision trees, or regression models, etc. Our implementation shouldn't make those assumptions either.

Bootstrap sample just means sampling training records with replacement, generating a new data set that has on average something like 2/3s the distinct records from the original data set, with exponentially decaying numbers of records that are replicated once, twice, three times, etc.

Listing 2 shows code from the supporting `stat.random` library:

```
(defn sample-with-replacement [data prng]
  (let [n (count data)]
    (repeatedly n (fn [] (nth data (.nextInt ^Random prng n))))))
```

Listing 2: Bootstrapping

Note 1: `sample-with-replacement` returns a lazy sequence, meaning the samples from `data` won't be drawn until somebody looks at them. This has a perhaps not-so-obvious implication: Until it is fully realized, the lazy sequence will retain a reference to `data`, so it won't be available for garbage collection.

Note 2: `^Random` is Clojure syntax for a local type hint. `(.nextInt ^Random prng n)` is equivalent to the Java expression `((Random) prng).nextInt(n)`.

Here's code for bagging a sequence of learners:

```
(defn bagging [data prng learners combine]
  (let [bags (repeatedly
               (fn [] (sample-with-replacement data prng)))
        terms (map (fn [l b] (l b)) learners bags)]
    (combine terms)))
```

Listing 3: Bagging

`data`: A collection of training pairs.

`prng`: A pseudo-random number generator.

`learners`: A (lazy) sequence of learners, each a function that takes collection of training pairs and returns a (predictive model) function.

`combine`: A function (like `mean-model` from exercise ??) that combines a (lazy) sequence of model functions into a single model.

`bags`: A lazy infinite sequence of bootstrap samples from `data`.

`terms`: A lazy sequence of model functions (eg trees) produced by applying each learner to the corresponding bag. This will be an infinite sequence if `learners` is infinite, in which case it would by up to `combine` to determine how many terms to actually compute.

3.3 Decision trees

A general binary decision tree consists of

- internal *split* nodes, each containing a predicate that determines whether a record goes to the left or right child of that node.
- terminal *leaf* nodes, each containing a leaf model function whose value is the tree's prediction for any record that ends up in that node.

3.3.1 clojure.zipper

The implementation here uses the `clojure.zip` package for growing trees. The problem it solves for us is the following:

To be able to safely use concurrency in random forest learning and prediction, we need to rigorously minimize the amount of mutable shared state. In particular, we want our trees, once grown, to be immutable. But growth implies change. One approach is to have 2 representations for trees, a mutable one to use while growing, which gets replicated in an immutable representation when growing is done. Besides more than doubling the complexity the tree representation, this approach is problematic because tree growth itself is one of the places we'd like to be free to use concurrency.

`clojure.zip` manages traversing, and most important for us, 'editing' generic immutable tree-like data structures. clojure.zip has a purely functional interface:
`(clojure.zip/zipper fertile? get-children get-node root)`
which returns a edit location that can be moved around a tree with `(down loc)`, `(left loc)`, `(next loc)`, etc., and that can be used to 'modify' an immutable tree with `(replace loc node)`, `(insert-left loc node)`, etc.

In the call to `zipper`, `root` is an instance of our tree node representation, the root of the initial tree.

The first 3 arguments to `zipper` are functions with the signatures below, where `node` is an instance of our tree node representation, and `children` is a `seq` of instances of our node representation.

`(fertile? node)` Answers the question: 'Can this node have children?' (as opposed to 'Does this node have children?').[4]

`(get-children node)` What it says.

`(get-node node children)` Return a node that has `children` as its children, possibly using `node` to initialize the returned node.[5]

3.3.2 defrecord

The tree implementation described here uses several cooperating `defrecord`s to implement the zipper API. This is only one of many reasonable choices (and, given the hacky/afterthought/work-in-progress feel of `defrecord`, `defprotocol`, etc., perhaps not the best one).

Advantages include:

- compact syntax for dynamic creation of Java classes.
- all fields final, so instances are immutable if all field values are.
- general purpose Clojure functions are defined for each method, so that methods can be passed to `map`, `comp`, `juxt`, etc. (which isn't true for methods defined in ordinary Java classes)[6] .
- the generated Java class automatically implements the Clojure key-value map API:

[4]Referred to as '`branch?`' in the Clojure documentation, one of many examples of the Clojure authors' difficulty with the English language.

[5]Referred to as '`make-node`' in the Clojure documentation, which even more misleading than '`branch?`'. Most tree representations in Clojure will be immutable. If so, then there's no reason to construct a new node if we already have one that has the specified children, and that would most likely be true of `node` itself.

[6]This, of course, raises the question: Why not do this, as needed, for all Java classes?

  - faster and more compact, especially if type hints are added to fields, than using a general key-value map.
  - clone-able immutable prototypes via `(merge instance {:field value})`

The disadvantages all reflect a number of mystifyingly bad design choices:

- record definitions are not first class:
  - no anonymous record definitions.
  - record definitions 'public', with global dynamic scope.
  - record definition can only be done thru macro call, no corresponding function call.
  - serialization broken as of Clojure 1.3.
  - no meta-object protocol, inspection requires cumbersome Java reflection, and so on.
- only methods specified by a pre-existing Java interface can be defined, imposing exactly the kind of type constraint you don't want in a dynamic language.
- fields are all public, making encapsulation impossible.
- field access via keywords rather than method/function needlessly complicates field <-> method refactoring.
- default constructor doesn't do any input validation, no mechanism to add validation.
- no inheritance, no code reuse.

3.3.3 Static trees and prediction

We create the required Java interface with `defprotocol`:

```
(defprotocol Node
  (fertile? [node])
  (get-children [node])
  (get-node [node children]))
```

Listing 4: Tree node protocol

This just specifies the 3 functions/methods we will pass to the zipper API.

We need representations for terminal leaf nodes and internal split nodes:

```
(defrecord Leaf [model]
  Node
  (fertile? [leaf] false)
  (get-children [leaf] nil)
  (get-node [leaf children] leaf))
```

Listing 5: Leaf nodes

A `Leaf` has one field, whose value is the model function to be applied to any case that ends up in that leaf. Because leaves are terminal, they aren't fertile (can't have children). We are assuming that the model is immutable, so it's safe to return the leaf itself from the `get-node` method.

```
(defrecord Split [predicate true-child false-child]
  Node
  (fertile? [branch] true)
  (get-children [branch] (seq [true-child false-child]))
  (get-node [branch children]
    (Split. predicate (first children) (second children))))
```

Listing 6: Split nodes

A Split has 3 fields, whose values are the `predicate` function determining which cases go to which child, and the children corresponding to whether the predicate returns `true` or `false`.

```
(defn predict [node x]
  (if (instance? Split node)
    (if ((:predicate node) x)
      (recur (:true-child node) x)
      (recur (:false-child node) x))
  ((:model node) x)))

(defn tree-model [root] (fn [x] (predict root x)))
```

Listing 7: Generic prediction

Prediction is done by the simple recursive `predict` function in listing 7[7]. To get a model function we can, for example, use as a term in an additive mode, we need only construct a one-line closure as in `tree-model`.

The key thing to note is that so far, this tree implementation doesn't depend on what kind of tree model it is, regression, classification, survival, etc. It also doesn't depend on the nature of `x`, which might be a tuple of primitive fields (as in most implementations of decision trees), but could easily be a time series, a bag of terms, a polygon in a spatial mesh, or anything else.

[7]One of the quirks of Clojure is that it requires the programmer to identify opportunities for tail-call optimization and exploit manually them by calling `recur` in place of the function name, even though `defn/fn` could fairly easily do the equivalent for you.

3.3.4 Tree growing

To support learning trees from training data, I add a 3rd implementation of `Node`, called `Bud`. The job of a `Bud` is to carry shrinking subsets of training pairs down the tree, and 'sprout', creating either a `Leaf` node, or a `Split` node with 2 new `Bud`s as children.

```
(defrecord Bud [leaf-learner split-learner pairs]
  Node
  (fertile? [bud] false)
  (get-children [bud] nil)
  (get-node [bud children] bud))
```

Listing 8: Buds

The fields:

`leaf-learner`: A function that returns a model given a set of training pairs. For L2 regression, this would be something like `(fn [pairs] (constantly (mean y pairs)))` where `(mean y pairs)` is the mean of the `y` attribute in pairs, and `(constantly z)` returns a constant function, that is `((constantly z) x)` is `z` regardless of `x`.

`split-learner`: A function that returns a split predicate function given a set of training pairs. More detail in section 3.4.

`pairs`: A collection of `RegressionPairs`.

Functions that operate on `Bud`s:

```
(defn sprout [bud]
  (let [predicate ((:split-learner bud) (:pairs bud))]
    (if predicate
      (split bud predicate)
      (leaf bud))))
```

Listing 9: (sprout bud)

`sprout` applies the bud's split learner to its pairs, returning either a predicate function, or `false` if the learner believes the pairs shouldn't be split any further. Then either divides the training data between 2 new `Buds` or ends the tree growth by returning a `Leaf`.

```
(defn split [bud predicate]
  (let [groups (group-by (comp predicate :x) (:pairs bud))]
     (Split. predicate
             (merge bud {:pairs (get groups true)})
             (merge bud {:pairs (get groups false)}))))
```

Listing 10: (split bud)

`split` constructs an instance of `Split` from the bud and the predicate. Note the use of `merge` to 'clone' 2 new buds, updating only the pairs field.

```
(defn leaf [bud]
  (let [model ((:leaf-learner bud) (:pairs bud))]
    (Leaf. model)))
```

Listing 11: (leaf bud)

`leaf` constructs an instance of `Leaf`, getting the model function by applying the bud's leaf learner to its pairs.

3.3.5 Managing tree learning with clojure.zip

We create a zipper from a Bud with:

```
(defn zipper [root]
  (z/zipper fertile? get-children get-node root))
```

Listing 12: Create a zipper for growing a tree

This will create a `clojure.zip` edit location object that will use our definitions of `fertile?`, `get-children`, and `get-node` to navigate and edit the tree starting from the supplied `root` bud[8].

These are functions that use `clojure.zip` to walk over the current, terminal buds, sprouting into leaves or splits with new buds, where appropriate.

```
(defn grow-loc [loc]
  (cond (z/end? loc) (z/root loc)
        (z/branch? loc) (recur (z/next loc))
        :else (recur (z/next (sprout-loc loc)))))
```

Listing 13: Growing a tree using a zipper

Notes:

`grow-loc` a recursive function that walks the edit location `loc` depth first over the growing tree, finding buds to sprout.

[8] '`z/`' is short for '`clojure.zip/`', and assumes we have done something like `(:require [clojure.zip :as z])` in defining our namespace.

`z/end?` true if we've reached the end of a depth first traversal of the tree — the `loc` doesn't point to a node, but to a marker for end-of-iteration.

`z/root` returns the node at the root of the tree (not an edit location pointing to the root).

`z/branch?` true if the node pointed to by `loc` can have children, ie, is a `Split`.

`z/next` returns an edit location pointing to the next node in a depth first traversal of the tree.

```
(defn sprout-loc [loc]
   (let [bud (z/node loc)
         new-node (sprout bud)]
     (z/replace
       loc
       (z/make-node loc new-node (get-children new-node)))))
```

Listing 14: Sprouting a bud using a zipper

`sprout-loc` the `loc` must point to a `Bud`. We then edit the tree, replacing the bud with the new `Split` or `Leaf` returned from `sprout`.

`z/node` returns the `Node` pointed to by an edit location `loc`.

3.3.6 External interface for learning tree models

```
(defn learn [leaf-learner split-learner pairs]
  (let [root-loc (zipper (Bud. leaf-learner split-learner pairs))
        tree (grow-loc root-loc)]
    (tree-model tree)))
```

Listing 15: Tree learner

Note the use of `tree-model` from listing 7, to convert `tree` to a function.

3.4 Greedy split optimization

At this point, we've implemented the major conceptual content of random forests, in about 70 fairly short lines of Clojure. None of the code so far depends on the fact that we are doing L2 numerical regression, and could be used without change for classification, vector regression, etc.

In this section I will describe split optimization, concretely: how to implement the `split-learner` passed to `learn` in listing 15. This will require about as much code as we've written so far. Though not strictly necessary, I will for the first time introduce code that depends on the fact that L2 numerical regression (for speed).

3.4.1 Finding the best attribute

The entry point for split optimization is `best-split`, which returns a predicate function that defines the split.

```
(defn best-split [cost-factory acceptable? score pairs
                  attributes]
  (let [split (partial attribute-split
                       cost-factory acceptable? score pairs)
        cost (comp :cost splitter)
        best-attribute (argmin cost attributes)
        s (split best-attribute)]
    (:predicate s)))
```

Listing 16: Split optimization entry point

`cost-factory`: A function of no arguments that returns an instance of `Accumulator` (see listing 31). Each `Accumulator` is a cost function (such as the sum of squared deviations from the mean) that can be efficiently updated/downdated by adding/removing double precision numbers.

`acceptable?`: A predicate used to evaluate whether a given split is allowed. For example, it may the constrain Leaf nodes to have at least some minimum number of training records.

`score`: A double-valued function applied to groups of training records, used to sort the groups corresponding to discrete attributes. In L2 regression, the score function is just the mean of the y-values in the group. In 2-class classification, it's the fraction of training records in class 1.

`pairs`: Training data.

`attributes`: Attribute functions available for splitting.

`split`: Finds the best split for a given attribute. `partial` constructs a function of one argument that calls `attribute-split` (see listing 17) with the first 4 arguments set to the given values (this is often referred to as 'currying').

`cost`: A function the returns the cost of the best split on a given attribute. `comp` returns a function that is the composition of `:cost` and `splitter` — this is assuming that splitter returns a Map with the value of the best split it finds in the `:cost` field.

`best-attribute`: The best attribute to split on.

`s`: The best split on the best attribute, recomputed. (The re-computation could be avoided at the cost of a slightly more complicated version of `argmin`.) Assumed to be a Map with a `:predicate` field containing the split predicate.

3.4.2 The best split on a given attribute

```
(defn attribute-split [cost-factory acceptable? score pairs
                       attribute]
  (let [split (if (numerical? attribute)
                numerical-split
                categorical-split)]
    (split cost-factory acceptable? score pairs attribute)))
```

Listing 17: Best split for a given attribute

`attribute-split` just dispatches to the appropriate method for numerical versus categorical attributes.

3.4.3 Numerical splits

To optimize a numerical split, we

1. to handle ties correctly, group the training pairs on the value of the x attribute.
2. sort the groups by increasing x.
3. replace the groups of pairs by groups of corresponding y values.
4. add all the y values to the right hand cost function.
5. move each group from right to left
6. check whether the resulting split is acceptable.
7. update, if needed, the minimizing cost and x value.

When done we return the minimum cost and the corresponding predicate function in a map.

Every split is tested for acceptability, and, if no acceptable splits are found, infinite cost and a nil predicate are returned.

```
(defn numerical-split [cost-factory acceptable? score pairs
                       attribute]
  (let [groups (group-by :x pairs)
        groups (sort-by key groups)
        groups (map (fn [[x pairs]] [x (map :y pairs)]) groups)
        ^Accumulator cost0 (cost-factory)
        ^Accumulator cost1 (cost-factory)]
    (doseq [[_ ys] groups] (.addAll cost1 ys))
    (loop [gs groups
           xmin Double/NaN
           cmin Double/POSITIVE_INFINITY]
      (if-not (empty? gs)
        ; continue
        (let [[[x ys] & rgs] gs
              cost (double (+ (.deleteAll cost1 ys)
                              (.addAll cost0 ys)))]
          (if (and (acceptable? cost0 cost1)
                   (< cost cmin))
            (recur rgs (double x) cost)
            (recur rgs xmin cmin)))
        ; done
        (let [p (if-not (== Double/POSITIVE_INFINITY cmin)
                  (fn [x] (<= (attribute x) xmin)))]
          {:predicate p :cost cmin})))))
```

Listing 18: Optimal split on a numerical attribute

Note the use of a couple Clojure features:

destructuring bind: Instead of just binding a name to a value, Clojure also supports binding expressions containing names to values. The value must have structure homolgous to the name expression, and Clojure automatically takes apart the value, binding its pieces to the corresponding pieces of the name expression. Concretely, `(let [[[x ys] & rgs] gs…]…)` takes the first element of the sequence `gs`, which must itself be a sequence whose first element is a sequence, binds `x` to the first element of the first element of `gs`, binds `ys` to the second element of the first element of `gs`, and binds `rgs` to the rest of `gs`.

`loop` special form: `loop` does not actually do iterative looping as in C/Java's `for`, but is really syntatic sugar equivalent to defining a recursive local function, and then calling that function.[9]

3.4.4 Categorical splits

Many implementations of random forest regression either don't support categorical attributes, or have arbitrary restrictions on the number of categories (due mostly likely to the fact that they are derived from the original implementation in Fortran, which is impoverished in data structures compared to most modern languages).

The implementation here relies on the fact that, for numerical regression and 2-class classification, we don't need to consider all possible partitions of the categories into 2 subsets. Instead we can sort the categories by the value of a score function, and only consider simple numerical splits on the score.

For regression, the score is the mean of the y values for all the training cases with the given category.

For 2-class classification, it's the fraction of the training cases with the given category whose y value is class 0.

[9] I'm not sure whether the `loop` syntatic sugar is helpful or just confusing, but it would almost certainly be better to call it something else. The `for` macro is definitely perverse, at least its name.

```
(defn categorical-split [cost-factory acceptable? score pairs
                         attribute]
  (let [groups (group-by :x pairs)
        groups (sort-by (comp score val) groups)
        groups (map (fn [[x pairs]] [x (map :y pairs)]) groups)
        ^Accumulator cost0 (cost-factory)
        ^Accumulator cost1 (cost-factory)]
    (doseq [[x ys] groups] (.addAll cost1 ys))
    (loop [gs groups
           gmin groups
           cmin Double/POSITIVE_INFINITY]
      (if-not (empty? gs)
        ; continue
        (let [[[x ys] & rgs] gs
              cost (double (+ (.deleteAll cost1 ys)
                              (.addAll cost0 ys)))]
          (if (and (acceptable? cost0 cost1)
                   (< cost cmin))
            (recur rgs rgs cost)
            (recur rgs gmin cmin)))
        ; done
        (let [p (if-not (== Double/POSITIVE_INFINITY cmin)
                  (let [cats (set (map first gmin))]
                    (fn [x] (contains? cats (attribute x)))))]
          {:predicate p :cost cmin} )))))
```

Listing 19: Optimal split on a categorical attribute

3.4.5 Split constraints

A simple split acceptability predicate is shown in

```
(defn mincount-split? [mincount cost0 cost1]
  (and (<= mincount (.count ^Accumulator cost0))
       (<= mincount (.count ^Accumulator cost1))))
```

Listing 20: Minimum amount of training data in a leaf

This will ensure that every leaf has at least `mincount` training cases.

3.5 Cost functions for L2 numerical regression

The cost function for L2 regression is the sum of squared deviations from the mean: $L_2(\mathcal{T}) = \sum_{y \in \mathcal{T}} (y - \bar{y}(\mathcal{T}))^2$, where $\bar{y}(\mathcal{T}) = \frac{1}{\#\mathcal{T}} \sum_{y \in \mathcal{T}} y$. Computing this accurately in an online fashion, allowing for the updating/downdating needed for fast split optimization, requires some care. An implementation that does this efficiently and accurately is provided in `stat.stats/MeanSSE`.

However, a little bit of algebra will let us use an even simpler alternative to get the same splits. Any split partitions the training y-values $\mathcal{T} = \{y\}$ into left and right subsets: $\mathcal{T} = \mathcal{L} \uplus \mathcal{R}$. The split cost is:

$$
\begin{aligned}
c(\mathcal{L}, \mathcal{R}) =& L_2(\mathcal{L}) + L_2(\mathcal{R}) \\
=& \sum_{y \in \mathcal{L}} (y - \bar{y}(\mathcal{L}))^2 + \sum_{y \in \mathcal{R}} (y - \bar{y}(\mathcal{R}))^2 \\
=& \sum_{y \in \mathcal{L}} \left[y^2 - 2\bar{y}(\mathcal{L})\, y + \bar{y}(\mathcal{L})^2 \right] + \sum_{y \in \mathcal{R}} \left[y^2 - 2\bar{y}(\mathcal{R})\, y + \bar{y}(\mathcal{R})^2 \right] \\
=& \sum_{\mathcal{L} \uplus \mathcal{R}} y^2 - \frac{\left(\sum_{\mathcal{L}} y\right)^2}{\#\mathcal{L}} - \frac{\left(\sum_{\mathcal{R}} y\right)^2}{\#\mathcal{R}}
\end{aligned}
$$

Since $\sum_{\mathcal{L} \uplus \mathcal{R}} y^2$ doesn't depend on the split, minimizing $c(\mathcal{L}, \mathcal{R})$ is equivalent to minimizing $-\left[\frac{\left(\sum_{\mathcal{L}} y\right)^2}{\#\mathcal{L}} + \frac{\left(\sum_{\mathcal{R}} y\right)^2}{\#\mathcal{R}}\right]$, so we can use $\frac{-\left(\sum_{\mathcal{T}} y\right)^2}{\#\mathcal{T}}$ as our cost function in split optimization.. An implementation of this cost is provided in `stat.stats/MSSN`.

3.6 Random forest split optimization

Breiman's original idea was to apply bagging to decision trees, to get smoother predictions than those returned by the best single decision tree models like CART[4]. (See appendix 2 for examples.) However, bagged CART forests had worse prediction accuracy than single CART trees in most problems, not to mention being $\approx 10^2$ times more expensive to compute. Replacing CART with cheaper alternative approaches to regularized greedy trees, like restricting the depth or the number of leaves, reduced computational cost, but didn't help accuracy.

The change that made (what was eventually named) random forests work well enough to be useful, was to replace regularized greedy trees with unregularized trees using randomized, not-quite-greedy split optimization.

Greedy split optimization considers all the attributes in the training data, choosing to split on the attribute whose best split most reduces the cost function.

Random forest split optimization selects a (different) random subset of the attributes at each node, and chooses the best split, restricted to that random subset.

If this seems somewhat mysterious to you, you are in good company. The arbitrariness of this is further emphasized by the fact that it introduces a tuning parameter, the size of the random subset of the attributes, without any real guidance for choosing the value of that parameter. Breiman and Cutler recommended $m = \left\lfloor \frac{p}{3} \right\rfloor$ for regression problems and $m = \left\lfloor p^{\frac{1}{2}} \right\rfloor$ for classification problems, based on experience with a small number of small data sets. (m is the subset size and p is the total number of attributes.) However, varying m away from the recommended values often greatly improves prediction accuracy (see for example [17] figure 15.4).

```
(defn random-forest-splitter [attributes cost-factory splittable?
                              score m prng]
  (fn [pairs]
    (let [a (sample m attributes prng)]
      (best-split cost-factory splittable? score pairs a))))
```

Listing 21: Random forest split optimizer

`random-forest-splitter` returns a function that does random forest split optimization. Every time it's called, it selects a new subset of the original list of attributes, and then applies the greedy `best-split` function to choose the best split from the attribute subset.

3.7 Generic random forests

Here's a generic random forest implementation — generic in the sense that it will do either classification or regression, depending on the choices of `leaf-learner`, `cost-factory`, `score`, and `combine`.

```
(defn random-forest [pairs attributes leaf-learner cost-factory
                     splittable? score combine ntrees m]

 (let [bag-prng (mersenne-twister-generator)
       prngs (repeatedly mersenne-twister-generator)
       make-splitter
         (partial random-forest-splitter
                  attributes cost-factory splittable? score m)
       splitters (map make-splitter prngs)
       learners (map (fn [s] (partial tree/learn leaf-learner s))
                      splitters)]

    (bagging pairs bag-prng (take ntrees learners) combine)))
```
Listing 22: Generic random forests

`bag-prng` a pseudo-random number generator used for bootstrap samples.

`prngs` a lazy infinite sequence of random number generators, for randomized split optimization in each tree.

`make-splitter` a splitter factory that curries out the first 4 arguments to `random-forest-splitter`.

`splitters` a lazy infinite sequence of random forest splitters.

`learners` a lazy infinite sequence of random forest tree learners.

I keep the random forest finite by only passing the first `ntrees` learners to `bagging`.

3.8 Random forest regression

```
(defn regression [pairs attributes ntrees mincount]

  (let [score (fn [d] (mean :y d))
        leaf-learner (fn [d] (constantly (score d)))
        splittable? (partial mincount-split? mincount)
        m (max 1 (int (/ (count attributes) 3)))]

    (random-forest data leaf-learner mssn splittable? score
                   mean-model ntrees m)))
```
Listing 23: Random forest regression

3.9 Examples

3.10 Boston Housing (2)

The Boston Housing Data[10], though tiny (506x19) by contemporary standards, has been a standard test set for regression since the 1970s. Code that parses a tab-separated file containing the data into appropriate regression training pairs, using the tsv library described in section B.2.3, is in listing 24.

[10] A little known fact: this data was originally collected by my older brother, while an undergraduate. He's since gone on to bigger and better things [28].

```
(defrecord Record
   [^double lon ^double lat ^double crim ^double zn
    ^double indus ^double nox ^double rm ^double age
    ^double dis ^double rad ^double tax ^double ptratio
    ^double b ^double lstat ^boolean chas])
```

Listing 24: Boston housing record class

```
(defn parser-factory [header]
  (let [da (fn [key]
             (let [ta (tuple-accessor header key)]
               (fn [tuple]
                 (Double/parseDouble (ta tuple)))))]
    (fn [tuple]
      (RegressionPair.
        (Record.
          ((da :lon) tuple) ((da :lat) tuple) ((da :crim) tuple)
          ((da :zn) tuple) ((da :indus) tuple) ((da :nox) tuple)
          ((da :rm) tuple) ((da :age) tuple) ((da :dis) tuple)
          ((da :rad) tuple) ((da :tax) tuple) ((da :ptratio) tuple)
          ((da :b) tuple) ((da :lstat) tuple)
          (zero? (Integer/parseInt
                   ((tuple-accessor header :chas) tuple))))
        ((da :cmedv) tuple)))))
```

Listing 25: Boston housing parser

```
(def pairs
  (parse transformer-factory "data/BostonHousing2.tsv")
```

Listing 26: Reading the Boston housing data

```
(def cd {:codomain Double/TYPE})

(def attributes
  [(with-meta (fn [^Record r] (.lon r)) cd)
   (with-meta (fn [^Record r] (.lat r)) cd)
   (with-meta (fn [^Record r] (.crim r)) cd)
   (with-meta (fn [^Record r] (.zn r)) cd)
   (with-meta (fn [^Record r] (.indus r)) cd)
   (with-meta (fn [^Record r] (.nox r)) cd)
   (with-meta (fn [^Record r] (.rm r)) cd)
   (with-meta (fn [^Record r] (.age r)) cd)
   (with-meta (fn [^Record r] (.dis r)) cd)
   (with-meta (fn [^Record r] (.rad r)) cd)
   (with-meta (fn [^Record r] (.tax r)) cd)
   (with-meta (fn [^Record r] (.ptratio r)) cd)
   (with-meta (fn [^Record r] (.b r)) cd)
   (with-meta (fn [^Record r] (.lstat r)) cd)
   (with-meta (fn [^Record r] (.chas r))
     {:codomain Boolean/TYPE})]))
```

Listing 27: Boston housing attribute functions

Listing 27 defines the x attribute functions I will pass to random forests. They are just field accessors, annotated with Clojure meta-data about the possible values, which is used to determine if a function represents a numerical or categorical attribute.

```
(def forest (random-forest/regression pairs attributes 128 4)
(println ntrees mincount "bias" (bias forest pairs))
(println ntrees mincount "rmse" (rmse forest pairs))

; Should print something close to:
; 128 4 bias 0.055305656
; 128 4 rmse 1.9137275128
```

Listing 28: Boston housing forest on training data

A Clojure

A.1 Installation

In developing this kit, I used the Oracle Java 6 JVM[21], Clojure (core) 1.3 [18], Leiningen 1.6.2[15] (which may need Maven [39]), and Eclipse 3.7 (Indigo) plus the Counterclockwise Clojure Eclipse plugin [38]. I developed on Windows 7.

Of these, strictly speaking, you need only ensure you have a Java JDK installed; the Clojure jar plus other required jars (eg uncommons-maths [10]) are included with the kit.

Whether you use Eclipse plus Counterclockwise — as of Counterclockwise version 0.5.0, adequate but not exciting — or some other IDE, or just a simple text editor plus a REPL launcher, is up to you.

A simple Clojure launcher for Windows is provided with the kit as `clj.bat` in the top folder. It will either run the Clojure script in the file name you provide, or launch a modestly interactive REPL, if no file name is provided. Modifying this to launch Clojure on your operating system should be easy. (You may want to experiment with the best JVM options for your particular environment.)

I don't expect the random forest code to be very sensitive to the the exact versions of any of these components (eg Java 7 should work fine). I recommend either using what comes with the kit, or installing the latest versions and fixing what doesn't just work.

A.2 Lisp overview

See also: [1, 23, 29, 36, 37]

A.3 Clojure overview

TODO: critical review of web sites, books [11, 12, 16, 32].

Two key entities in Clojure programs

Functions: First class[11], lexically scoped.

Sequences: Mostly immutable, mostly lazy.

Most expressions in Clojure code perform:

Mapping: Apply a function to the items in one or more sequences returning a new (lazy immutable) sequence, for example:

`(map model dataset)` returns a sequence of the predictions produced by applying the model function to each item in the data set.

Reduction: Apply a function to a (lazy immutable) sequence returning some more atomic object, for example:

`(reduce + predictions)` returns the sum of the predicted values.

Function creation: Apply a (higher order) function to other functions creating a new function, for example:

`(juxt f0 f1 f2)` returns a new function g such that `(g x)` == `[(f0 x) (f1 x) (f2 x)]`.

[11] Actually, not quite true in Clojure, perhaps its most important failing.

A.3.1 Testing

It's generally good practice to create tests as, or even before, we write any significant chunk of code. The standard Clojure distribution supports unit testing thru the `clojure.test` package. Here's a simple (maybe too simple) test of `mean-model`:

```
(deftest test-mean-model
  (testing
    "mean-model"
    (let [f (fn [x] (+ x 17))
          g (fn [x] (Math/cos x))
          m0 (fn [x] (/ (+ (f x) (g x)) 2))
          m1 (mean-model [f g])]
      (doseq [x (repeatedly 5 (fn [] (rand (Math/PI))))]
        (is (== 0 (- (m0 x) (m1 x))))))))
```

Listing 29: Trivial unit test for `mean-model`

Assuming this code is in the `rfrk.test.additive-models` namespace, this, and any other tests in that namespace, can be run by evaluating `(run-tests 'rfrk.test.additive-models)`.

B Libraries

TODO:

B.1 stat: probabillity and statistics.

B.1.1 stat.stats

Collection of *records*

Attributes as annotated functions

Regression (x,y) pairs and delegation.

Weighting and delegation again.

B.1.2 stat.random: reproducible random numbers and concurrency

Random forests decomposes nicely, in a variety of ways, into independent tasks that can be computed concurrently. The most obvious example is the fact that each tree in the forest can be grown in parallel with the other trees.

Random forests is a randomized algorithm. For testing, at least, we want to be able to seed the pseudo-random number generation to ensure reproducible results. In order to support concurrency correctly, we need to take a little extra care with how we seed and generate the pseudo-random numbers that drive tree growth.

One approach would be to have all the tasks share the same generator. Even if the generator is thread-safe, this will fail to give us reproducible results, in a concurrent implementation, because the tasks' requests to the generator will be interleaved non-deterministically.

So we need to give every task its own pseudo-random number generator, and every task's generator must have its own seed. (If the tasks shared the seed, they would all see the sequence of pseudo-random numbers.)

If we generate the seeds from the same pseudo-random number generator algorithm used by the tasks, then every task will see essentially the same sequence of random numbers, only shifted by 1,2,3... This may be ok, but it is difficult to prove that this won't introduce dangerous correlations. Using a different pseudo-random number generator algorithm to generate the seeds might help, but, again, it's difficult to verify.

A simple alternative is to collect a large enough cache of 'true' random numbers to use for seeds. This is easy to do using `DefaultSeedGenerator` from uncommons-maths [10]. An example of how to do this is in `stat.random`:

```
(def mersenne-twister-seeds
     (ref ["A8ECFAFD28968DF24F3308151EB62826" ...]))

(defn mersenne-twister-generator []
  (dosync
    (let [seed (first (deref mersenne-twister-seeds))]
      (alter mersenne-twister-seeds next)
      (assert (not (nil? seed)) "Out of mersene twister seeds!")
      (MersenneTwisterRNG.
(BinaryUtils/convertHexStringToBytes seed))))
```

Listing 30: Independent random number generators

To make it safe for concurrent tasks to pull seeds from from the cache, we wrap the vector containing the seeds in a `ref`, which is one of Clojure's 4 mechanisms for managing mutable state under concurrency (see, for example, Table 6.2 in [32]). `dosync`, `deref`, and `alter` ensure that only one task will use each seed.

B.1.3 stat.stats

Some basic statistics over collections, eg, `(mean function data)`, which computes the mean of the value of `function` applied to the elements of `data`.

```
(defprotocol Accumulator
  (clear [a])
  (add [a x])
  (addAll [a xs])
  (remove [a x])
  (removeAll [a xs])
  (count [a])
  (value [a]))
```

Listing 31: Updatable cost functions

The Accumulator protocol defines an interface for incremental statistics, used in random forests for fast split optimization. Implementations are provided for mean, variance, and related quantities.

B.2 General utility code

B.2.1 util.core

A few utility functions that might have been in clojure.core:

`(argmin function collection)` return an element `ci` of the collection with the minimum value over the collection of `(function ci)`.

`(mapmap function key-value-map)` Like `(map function sequence)`, returning a new key-value map with the values transformed by `function`.

B.2.2 util.gz

Readers and writers for zip/gzip compressed files, or uncompressed files, based on the file name.

B.2.3 util.tsv

An initial stab at reading and writing tab-separated files with header, vaguely similar to R's `read.table` and `write.table`.

`(parse parser-factory filename)` Returns a fully realized (not lazy) `ArrayList`, where each item in the `ArrayList` corresponds to one line in the file named by `filename`.

`parser-factory` A function that converts the sequence of header tokens into the actual transformer function that used to parse each data line. (Not as confusing in practice as it may sound.)

`filename` the path to the file to be parsed.

`(tuple-accessor header key)` A function used in defining transformer factories passed to `parse`. It basically looks up the position of the key in the header to get a positional accessor for the corresponding token in each of the data lines.

`(write-records records filename)` Assumes `records` is a collection of instances of a single Java class. Uses Java reflection to discover the instance fields defined by that class, and writes a tab-separated file with header with the values of all the discovered fields.

C Solutions to some exercises

Exercise ??:

```
(defn mean-model [terms]
  (let [f (apply juxt terms)
        n (count terms)]
    (fn [x] (/ (reduce + (f x)) n))))
```

Listing 32: Mean additive model

Exercise ??:

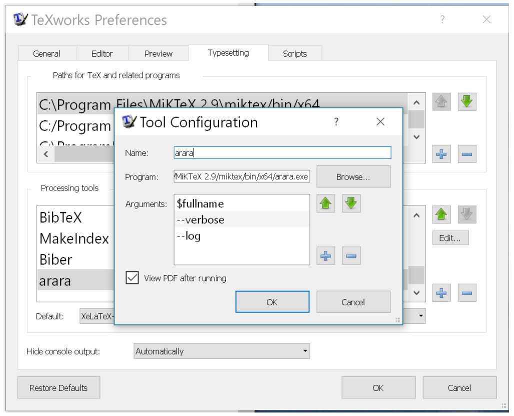
```
(defn make-canonizer
  "Return␣a␣closure␣containing␣a␣HashMap␣used␣to␣de-dup␣its␣argument."
  ([]
    (let [canon (HashMap.)]
      (fn [item]
        (or (.get ^HashMap canon item)
            (.put ^HashMap canon item item)
            item))))

  ([n]
    (let [canon (HashMap. (int n))]
      (fn [item]
        (or (.get ^HashMap canon item)
            (.put ^HashMap canon item item)
            item)))))
```

Listing 33: Canonizer factory

D Typesetting

This document was typeset using MikTEX 2.9 [35] and TEXworks 0.6.1 [22] on WINDOWS 10. I used `arara` [5] to run `xelatex`, `biber`, `makeglossaries`, and `makeindex`. I believe only MikTEX and TEXworks are Windows specific; the actual typesetting tools should be usable on Linux and MacOS as well.



Paths for TeX and related programs
C:\Program Files\MiKTeX 2.9\miktex/bin/x64
C:/Program

Processing tools
BibTeX
MakeIndex
Biber
arara

Default: XeLaTeX-

Hide console output: Automatically

Name: arara
Program: MiKTeX 2.9/miktex/bin/x64/arara.exe
Arguments:
```
$fullname
--verbose
--log
```
[✓] View PDF after running

Figure D.1: Configuring TEXworks for `arara`.

References

[1] H. ABELSON, G.J. SUSSMAN, and J. SUSSMAN.
Structure and interpretation of computer programs.
MIT Electrical Engineering and Computer Science.
MIT Press, 1996.
ISBN: 9780262510875.

[2] Richard A. BECKER, John M. CHAMBERS, and Allan R. WILKS.
The New S Language.
London: Chapman and Hall, 1988.

[3] Leo BREIMAN.
"Random Forests".
In: *Machine Learning* 45.1 (2001), pages 5–32.

[4] Leo BREIMAN, Jerome H. FRIEDMAN, Richard OLSHEN, and Charles J. STONE.
Classification and Regression Trees.
Wadsworth, 1984.

[5] Paulo Roberto Massa CEREDA, Marco DANIEL, Brent LONGBOROUGH, and Nicola Louise Cecilia TALBOT.
`arara`: The cool TEX automation tool.
URL: https://github.com/cereda/arara (visited on 2017-03-13).

[6] John M. CHAMBERS.
Programming with Data.
ISBN 0-387-98503-4.
New York: Springer, 1998.
URL: http://cm.bell-labs.com/cm/ms/departments/sia/Sbook/.

[7] John M. CHAMBERS and Trevor J. HASTIE.
Statistical Models in S.
London: Chapman and Hall, 1992.

[8] Winston CHURCHILL.
PARLIAMENT BILL Debate.
1947-11.
URL: http://hansard.millbanksystems.com/commons/1947/nov/11/parliament-bill#column_206.

[9] David DONOHO.
"A Tool for Research in Data Analysis".
PhD thesis. Harvard, 1982.

[10] Daniel W. Dyer.
Uncommons Maths.
[Online; accessed 2012-01-05].
2012.
URL: http://maths.uncommons.org/.

[11] C. Emerick, B. Carper, and C. Grand.
Clojure Programming.
O'Reilly Media, 2011.

[12] M. Fogus and C. Houser.
The Joy of Clojure: Thinking the Clojure Way.
Manning Pubs Co Series.
Manning Publications, 2011.
ISBN: 9781935182641.

[13] Jerome H. Friedman.
Recent advances in predictive (machine) learning.
Technical report.
Stanford University, Dept. of Statistics, 2003-11.
URL: http://www-stat.stanford.edu/~jhf/#reports.

[14] Antonio Garrote.
clj-ml.
[Online; accessed 2012-01-05].
2012.
URL: http://antoniogarrote.github.com/clj-ml/.

[15] Phil Hagelberg.
Leinigen.
2012.
URL: https://github.com/technomancy/leiningen.

[16] S. Halloway.
Programming Clojure.
Pragmatic Programmers.
Pragmatic Bookshelf, 2009.
ISBN: 9781934356333.

[17] Trevor Hastie, Rob Tibshirani, and Jerome H. Friedman.
The elements of statistical learning.
2nd.
Springer-Verlag, 2009.
URL: http://www-stat.stanford.edu/~tibs/ElemStatLearn/.

[18] Rich Hickey.
clojure.org.
2012.
URL: http://clojure.org.

[19] Ross Ihaka.
R: Lessons Learned, Directions for the Future.
2010.

[20] Ross Ihaka and Duncan T. Lang.
"Back to the Future: Lisp as a Base for a Statistical Computing System".
In: *COMPSTAT 2008* (2008), pages 21–33.
DOI: 10.1007/978-3-7908-2084-3_2.
URL: http://dx.doi.org/10.1007/978-3-7908-2084-3%5C_2.

[21] *Java JDK 6.*
2012.
URL: http://www.oracle.com/technetwork/java/javase/overview/index-jsp-136246.html.

[22] Jonathan Kew and Stefan Löffler.
TEXworks: A simple interface for working with TEX documents.
URL: https://github.com/texworks/ (visited on 2017-03-13).

[23] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow.
The art of metaobject protocol.
Cambridge, MA, USA: MIT Press, 1991.
ISBN: 0-262-61074-4.

[24] Elizaveta Levina and Peter Bickel.
"The earth mover's distance is the Mallows distance: some insights from statistics".
In: *Proceedings of ICCV 2001.*
Vancouver, Canada, 2001,
Pages 251–256.

[25] Andy Liaw and Matthew Wiener.
randomForest: Breiman and Cutler's random forests for classification and regression.
Version 4.5-36.
2010-07.
URL: http://cran.r-project.org/web/packages/randomForest/.

[26] David Edgar Liebke.
Incanter.
2011.
URL: http://incanter.org.

[27] John Alan McDonald and Jan Pedersen.
"Computing Environments for Data Analysis III: Programming Environments".
In: *SIAM Journal on Scientific and Statistical Computing* 9.2 (1988), pages 380–400.
DOI: 10.1137/0909025.
URL: http://link.aip.org/link/?SCE/9/380/1.

[28] Robert N. McDonald.
Boston Housing Data.
1974.
URL: http://judgepedia.org/index.php/Robert_N._McDonald.

[29] P. Norvig.
Paradigms of artificial intelligence programming: case studies in common lisp.
Artificial intelligence programming languages.
Morgan Kaufman Publishers, 1992.
ISBN: 9781558601918.

[30] R. W. Oldford and S. C. Peters.
"DINDE: towards more sophisticated software environments for statistics".
In: *SIAM Journal on Scientific and Statistical Computing* 9.1 (1988-01), pages 191–211.

[31] R Development Core Team.
R: A Language and Environment for Statistical Computing.
R Foundation for Statistical Computing. 2012.
URL: http://www.R-project.org.

[32] A. Rathore.
Clojure in Action.
Manning Pubs Co Series.
Manning Publications, 2011.
ISBN: 9781935182597.

[33] Yossi Rubner, Carlo Tomassi, and Leonidas Guibas.
"A metric for distributions with applications to image databases".
In: *IEEE International Conference on Computer Vision.*
1998,
Pages 59–66.

[34] Yossi Rubner, Carlo Tomassi, and Leonidas Guibas.
The earth mover's distance as a metric for image retrieval.
Technical report STAN-CS-TN-98-862.
Computer Science, Stanford University, 1998-09.

[35] Christian Schenk.
MikTEX: typsetting beautiful documents.
URL: https://miktex.org/ (visited on 2017-03-13).

[36] Andrew Shalit, Jeffrey Piazza, and David Moon.
Dylan an object-oriented dynamic language.
Apple Computer Eastern Research and Technology, 1992.

[37] Guy L. Steele Jr.
Common LISP: the language (2nd ed.)
Newton, MA, USA: Digital Press, 1990.
ISBN: 1-55558-041-6.

[38] Counterclockwise Team.
Counterclockwise Eclipse Clojure plugin.
2012.
URL: `http://code.google.com/p/counterclockwise/`.

[39] Maven Team.
Apache Maven.
2012.
URL: `http://maven.apache.org/`.

[40] Weka Team.
Weka 3: Data Mining Software in Java.
2005-10.
URL: `http://www.cs.waikato.ac.nz/~ml/weka/`.

[41] Luke Tierney.
LISP-STAT: an object oriented environment for statistical computing and dynamic graphics.
New York, NY, USA: Wiley-Interscience, 1990.
ISBN: 0-471-50916-7.

[42] Luke Tierney.
"Some Notes on the Past and Future of Lisp-Stat".
In: *Journal of Statistical Software* 13.9 (2005-01), pages 1–15.

[43] L. VanderHart and S. Sierra.
Practical Clojure.
Apress Series.
Apress, 2009.
ISBN: 9781430272311.