How to Grow Distributed Random Forests

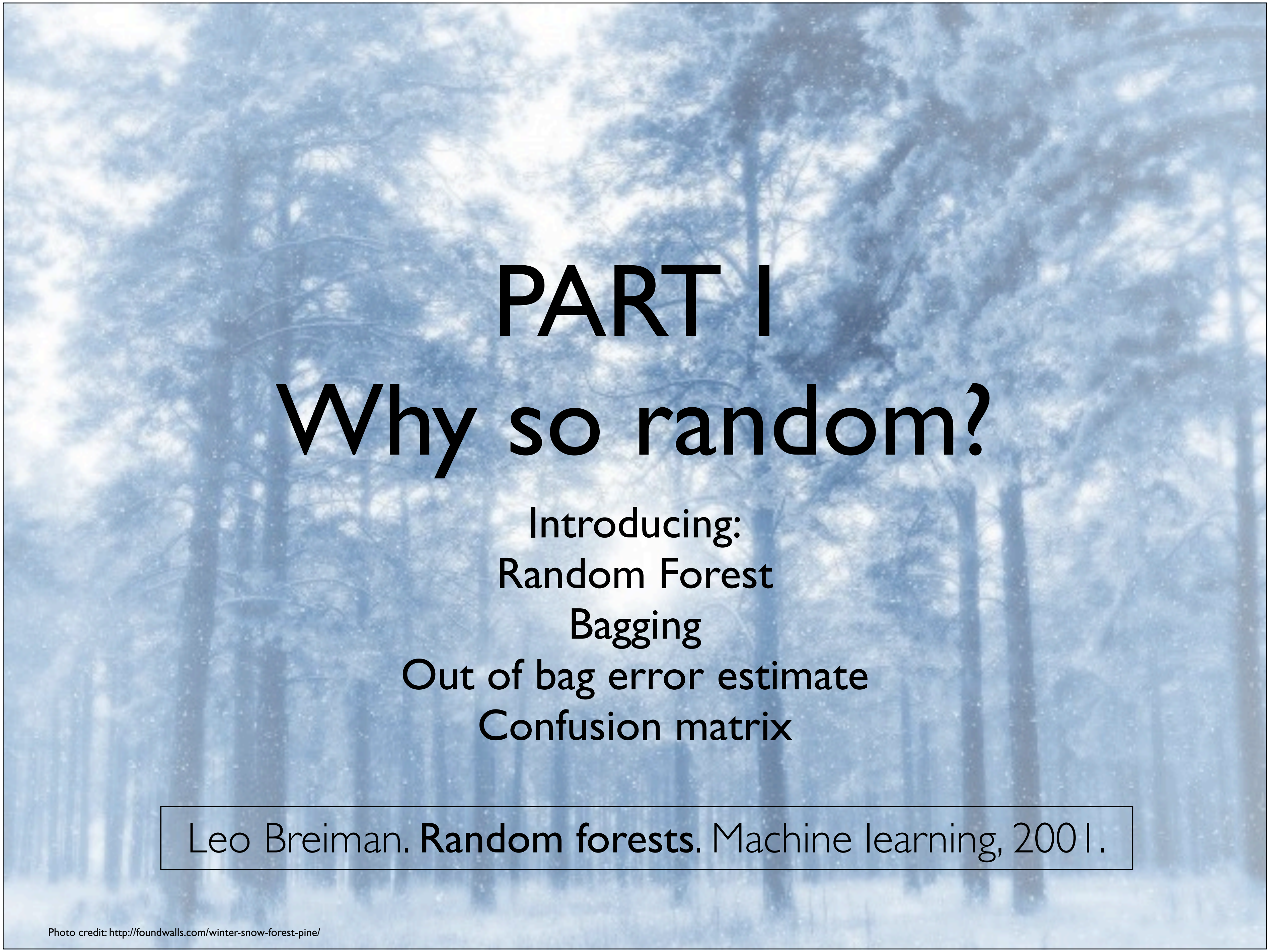
Jan Vitek

Purdue University
on sabbatical at 0xdata

Photo credit: http://foundwalls.com/winter-snow-forest-pine/

Overview

- Not data scientist…
- I implement programming languages for a living…
- Leading the FastR project; a next generation R implementation…
- Today I'll tell you how to grow a distributed random forest in 2KLOC



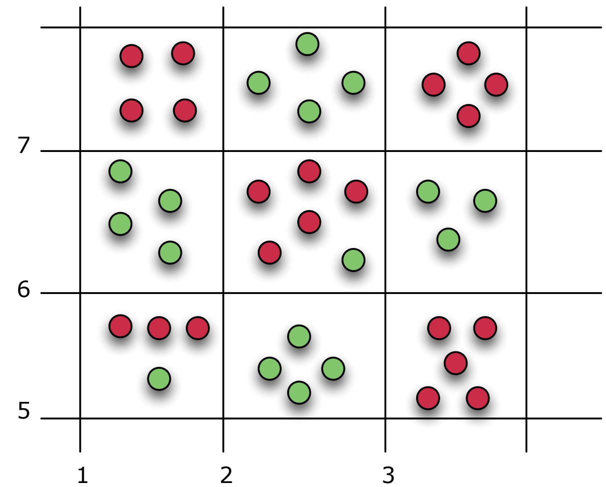
PART I
Why so random?

Introducing:
Random Forest
Bagging
Out of bag error estimate
Confusion matrix

Leo Breiman. **Random forests**. Machine learning, 2001.

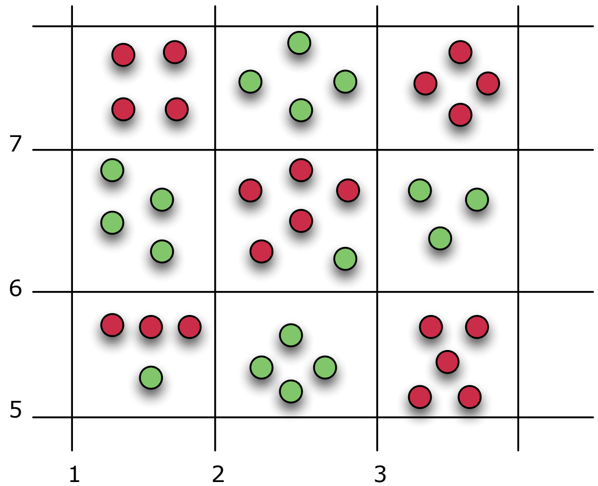
Photo credit: http://foundwalls.com/winter-snow-forest-pine/

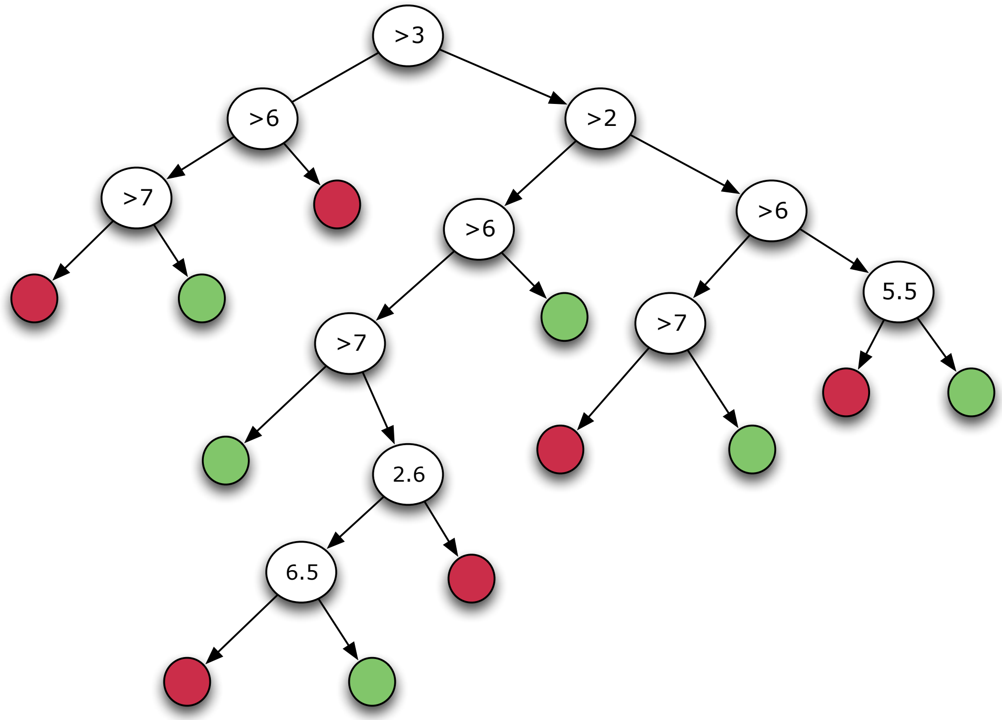
Classification Trees



- Consider a supervised learning problem with a simple data set with two classes and the data has two features **x** in [1,4] and **y** in [5,8].
- We can build a classification tree to predict classes of new observations

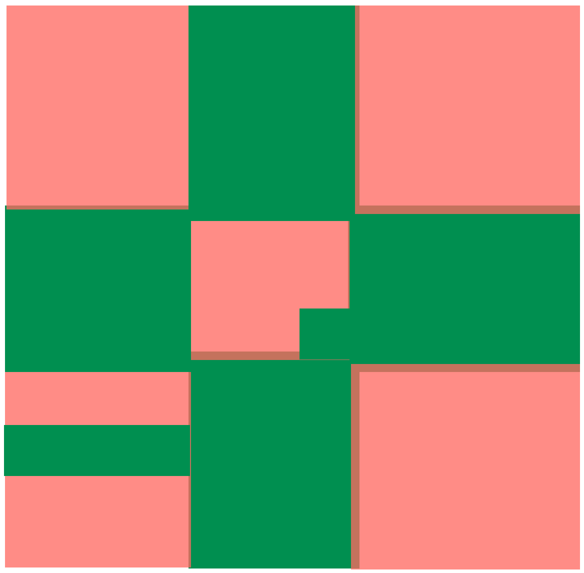
Classification Trees

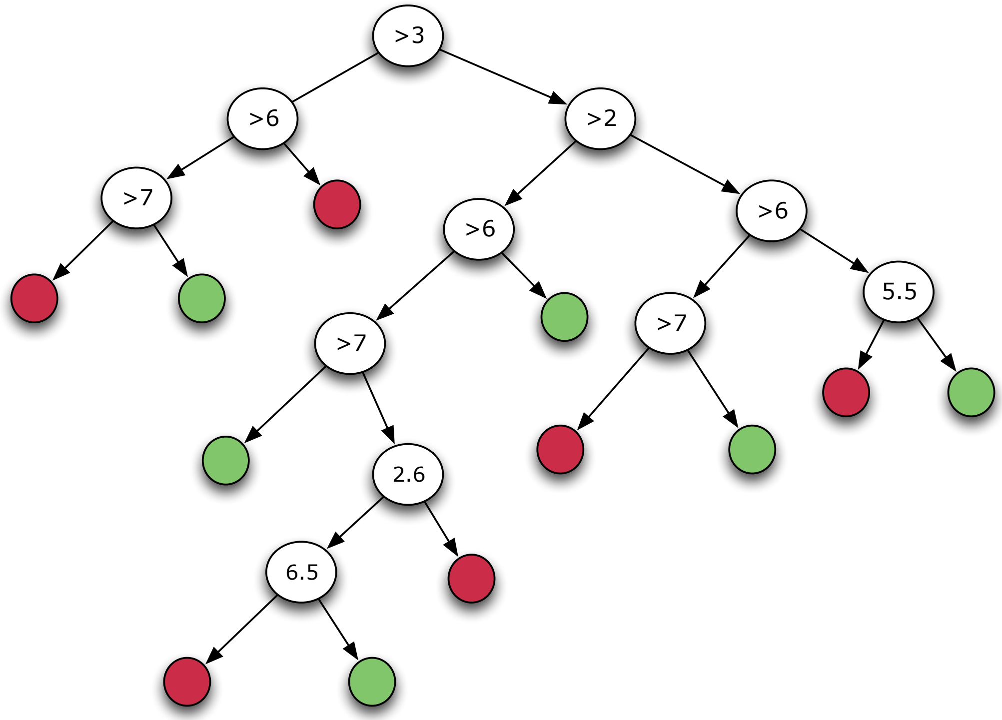




- Consider a supervised learning problem with a simple data set with two classes and the data has two features **x** in [1,4] and **y** in [5,8].
- We can build a classification tree to predict classes of new observations

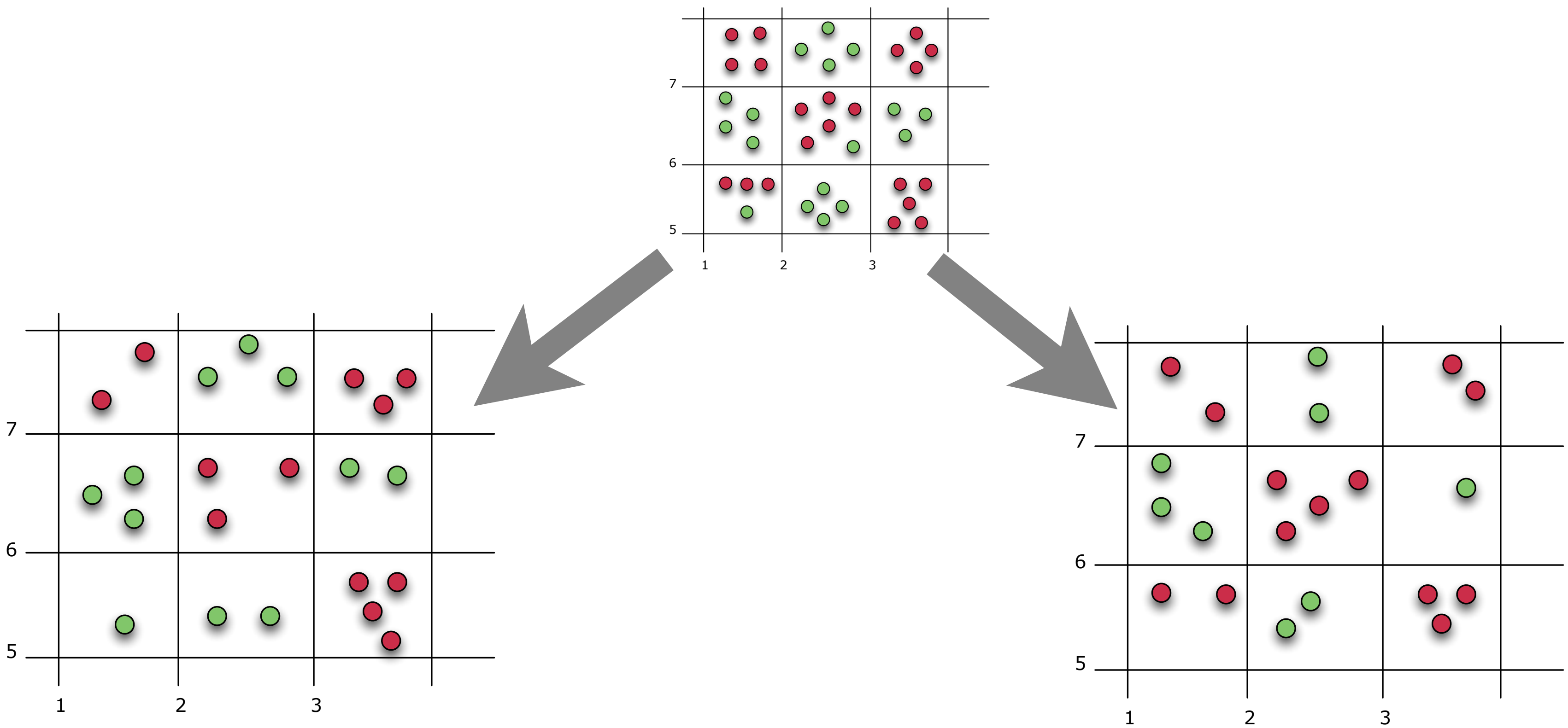
Classification Trees





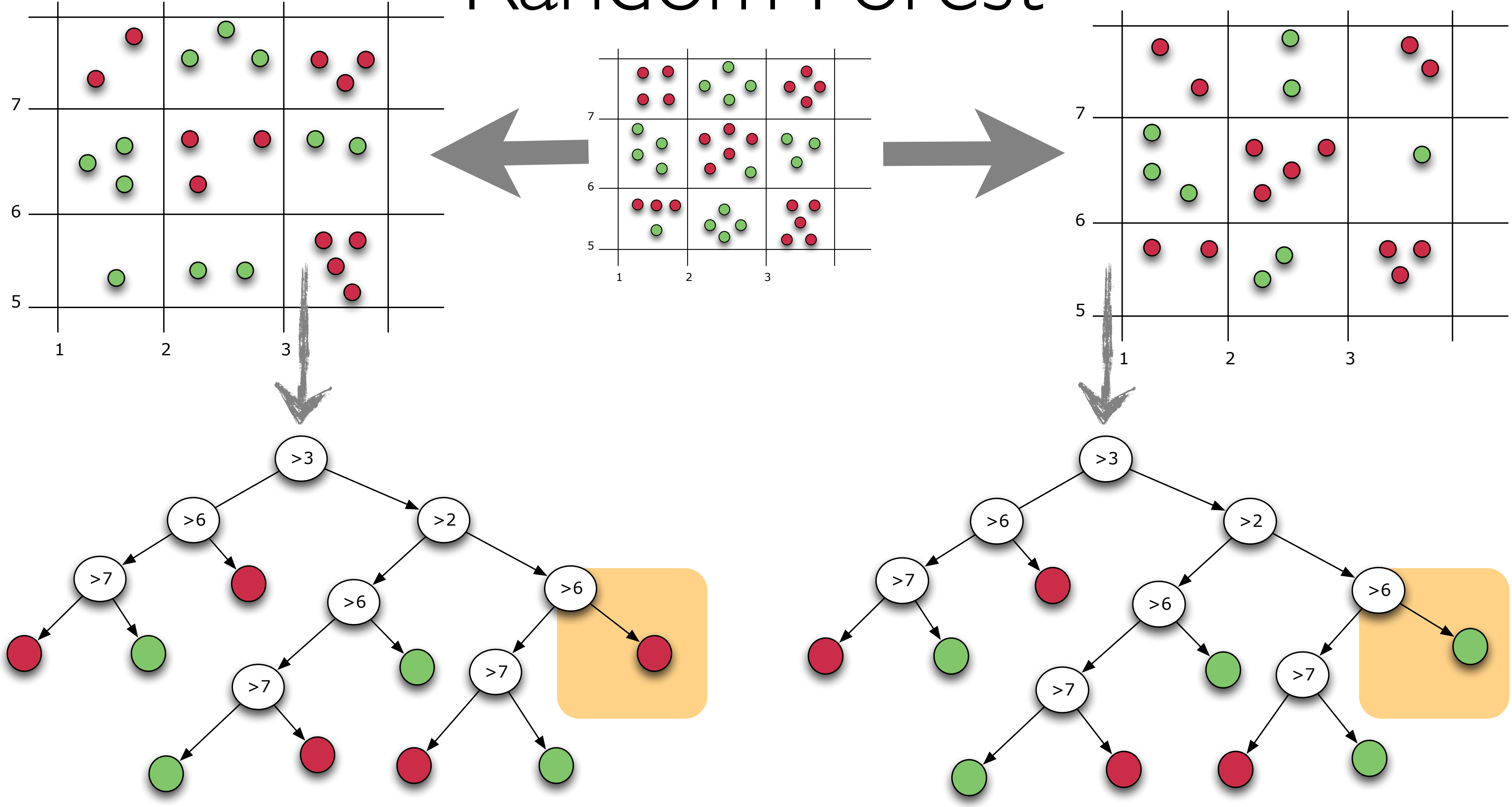
- Classification trees overfit the data

Random Forest



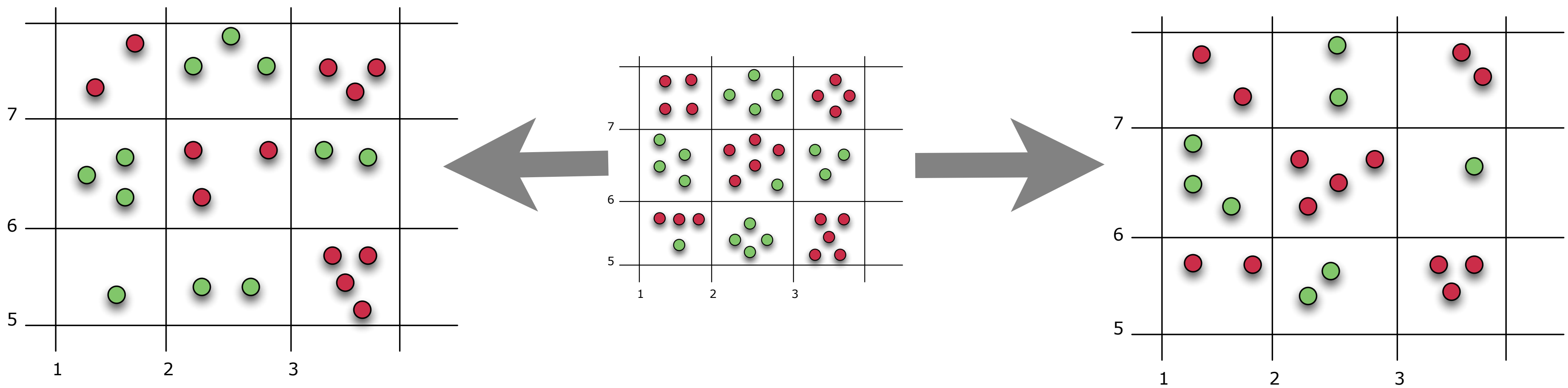
- Avoid overfitting by building many randomized, partial, trees and vote to determine class of new observations

Random Forest



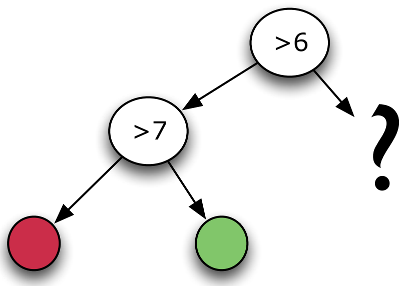
- Each tree sees part of the training sets and captures part of the information it contains

Bagging



- **First rule of RF:**
  each tree see is a different random selection
  (without replacement) of the training set.

Split selection



Gini impurity

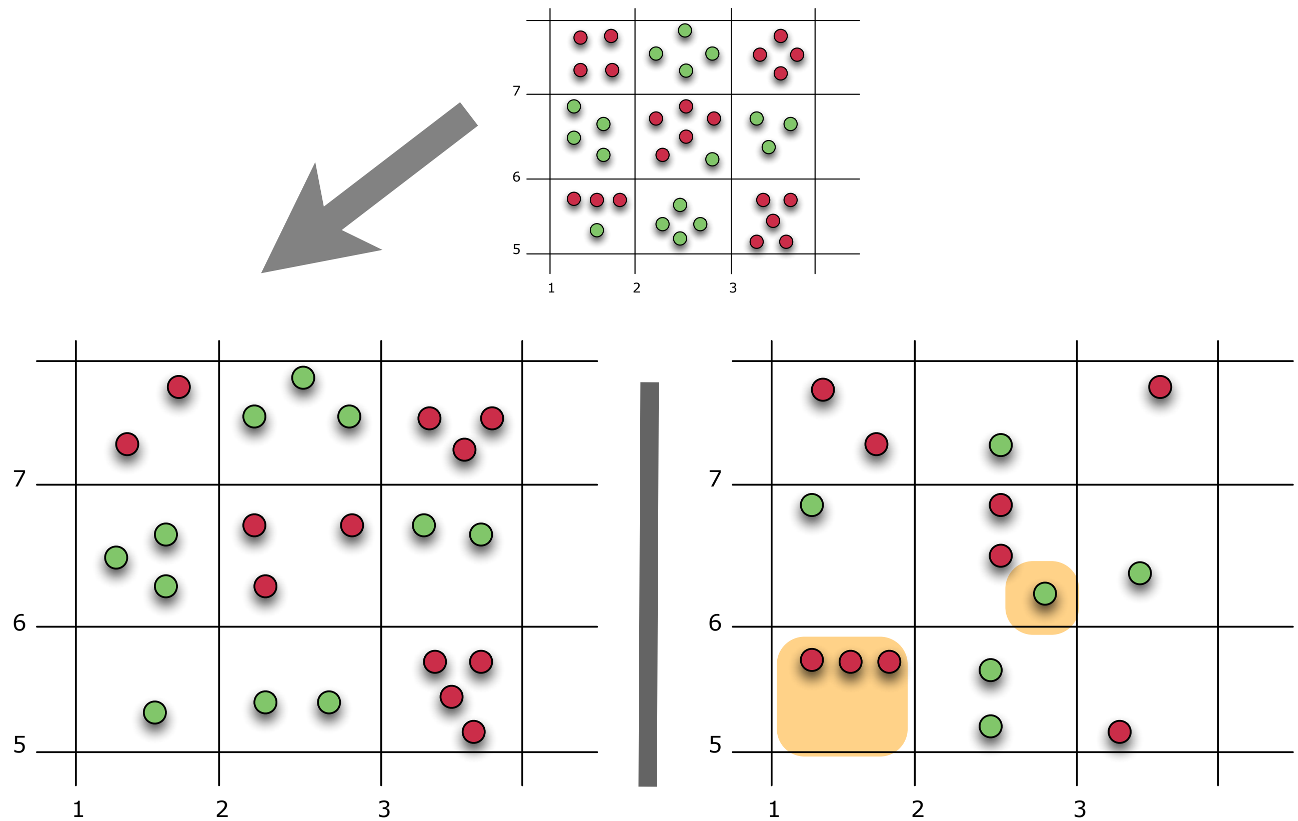
$$I_G(f) = \sum_{i=1}^{m} f_i(1-f_i) = \sum_{i=1}^{m}(f_i - f_i{}^2) = \sum_{i=1}^{m} f_i - \sum_{i=1}^{m} f_i{}^2 = 1 - \sum_{i=1}^{m} f_i{}^2$$

$$I_E(f) = -\sum_{i=1}^{m} f_i \log_2 f_i$$

Information gain

- **Second rule of RF:**
  Splits are selected to maximize gain on a random subset of features. Each split sees a new random subset.

OOBE



- One can use the training data to get an error estimate ("out of bag error" or OOBE)
- Validate each tree on complement of training data

Validation

assigned / actual	Red	Green	
Red	**15**	**5**	33%
Green	**1**	**10**	10%

- Validation can be done using OOBE (which is often convenient as it does not require preprocessing) or with a separate validation data set.

- A Confusion Matrix summarizes the class assignments performed during validation and gives an overview of the classification errors

PART II
Demo

Running RF on Iris

Photo credit: http://foundwalls.com/winter-snow-forest-pine/

Iris RF results

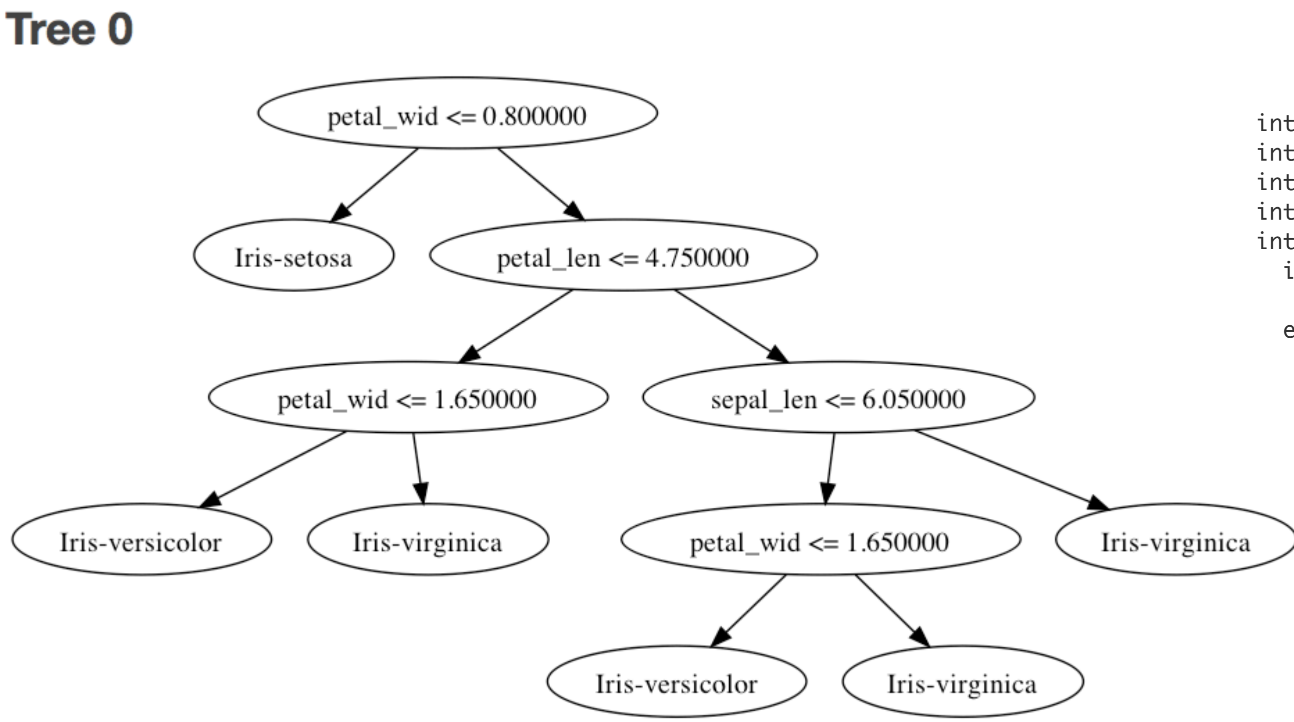
Confusion matrix - OOB error estimate

classification error	4.667 %
used / skipped rows	150 / 0 (0.0 %)

Actual \ Predicted	Iris-setosa	Iris-versicolor	Iris-virginica	Error
Iris-setosa	50	0	0	0.000 = 0 / 50
Iris-versicolor	0	47	3	0.060 = 3 / 50
Iris-virginica	0	4	46	0.080 = 4 / 50
Totals	50	51	49	**0.047 = 7 / 150**

Trees used: 50

Sample tree



- // Column constants

```
int COLSEPALLEN = 0;
int COLSEPALWID = 1;
int COLPETALLEN = 2;
int COLPETALWID = 3;
int classify(float fs[]) {
  if( fs[COLPETALWID] <= 0.800000 )
    return Iris-setosa;
  else
    if( fs[COLPETALLEN] <= 4.750000 )
      if( fs[COLPETALWID] <= 1.650000 )
        return Iris-versicolor;
      else
        return Iris-virginica;
    else
      if( fs[COLSEPALLEN] <= 6.050000 )
        if( fs[COLPETALWID] <= 1.650000 )
          return Iris-versicolor;
        else
          return Iris-virginica;
      else
        return Iris-virginica;
}
```

Comparing accuracy

Dataset	H_2O	R	Weka	wiseRF
Iris	**2.0%**	**2.0%**	**2.0%**	**2.0%**
Vehicle	**21.3%**	**21.3%**	22.0%	22.0%
Stego	**13.6%**	13.9%	14.0%	14.9%
Spam	**4.2%**	**4.2%**	4.4%	5.2%
Credit	6.7%	6.7%	**6.5%**	**6.5%**
Intrusion	21.2%	**19.0%**	19.5%	20.4%
Covtype	**3.6%**	22.9%	–	14.8%

Dataset	Features	Predictor	Instances (train/test)	Imbalanced	Missing observations
Iris	4	3 classes	100/50	NO	0
Vehicle	18	4 classes	564/282	NO	0
Stego	163	3 classes	3,000/4,500	NO	0
Spam	57	2 classes	3,067/1,534	YES	0
Credit	10	2 classes	100,000/50,000	YES	29,731
Intrusion	41	2 classes	125,973/22,544	NO	0
Covtype	54	7 classes	387,342/193,672	YES	0

- We compared several implementations and found that we are OK

PART III
Writing a DRF algorithm in Java with H2O

Design choices,
implementation techniques,
pitfalls.

Photo credit: http://foundwalls.com/winter-snow-forest-pine/

Distributing and Parallelizing RF

- When data does not fit in RAM, what impact does that have for random forest:
  - How do we sample?
  - How do we select splits?
  - How do we estimate OOBE?

Insights

- RF building parallelize extremely well when random data sample fits in memory
- Trees can be built in parallel trivially
- Trees size increases with data volume
- Validation requires trees to be co-located with data

Strategy

- Start with a randomized partition of the data on nodes
- Build trees in parallel on subsets of each node's data
- Exchange trees for validation

Reading and Parsing Data

- H2O does that for us and returns a ValueArray which is row-order distributed table

```
class ValueArray extends Iced implements Cloneable {

        long numRows()
        int numCols()
        long length()

        double datad(long rownum, int colnum) {
```

- Each 4MB chunk of the VA is stored on a (possibly) different node and identified by a unique key

Extracting random subsets

- Each node holds a random set of 4MB chunks of the value array

```
final ValueArray ary = DKV.get( dataKey ).get();

ArrayList<RecursiveAction> dataInhaleJobs = new ArrayList<RecursiveAction>();

for( final Key k : keys ) {
  if (!k.home()) continue;     // skip non-local keys
  final int rows = ary.rpc(ValueArray.getChunkIndex(k));
  dataInhaleJobs.add(new RecursiveAction() {
    @Override protected void compute() {
      for(int j = 0; j < rows; ++j)
        for( int c = 0; c < ncolumns; ++c)
          localData.add ( ary.datad( j , c) );
    }});
}

ForkJoinTask.invokeAll(dataInhaleJobs);
```

Evaluating splits

- Each feature that must be considered for a split requires processing data of the form (feature value, class)

  { (3.4, red), (3.3, green), (2, red), (5, green), (6.1, green) }

- We should sort the values before processing

  { (2, red), (3.3, green), (3.4, red), (5, green), (6.1, green) }

- But since each split is done on different sets of rows, we have to sort features at every split

- Trees can have 100k splits

Evaluating splits

- Instead we discretize the value

  { (2, red), (3.3, green), (3.4, red), (5, green), (6.1, green) }

- becomes

  { (0, red), (1, green), (2, red), (3, green), (4, green) }

- and no sorting is required as we can represent the colors by arrays (of size #cardinality of the feature)
- For efficiency we can bin multiple values together

Evaluating splits

- The implementation of entropy based split is now simple

```
Split ltSplit(int col, Data d, int[] dist, Random rand) {

final int[] distL = new int[d.classes()], distR = dist.clone();
final double upperBoundReduction = upperBoundReduction(d.classes());
double maxReduction = -1;  int bestSplit = -1;
for (int i = 0; i < columnDists[col].length - 1; ++i) {
  for (int j = 0; j < distL.length; ++j) {
    double v = columnDists[col][i][j]; distL[j] += v;  distR[j] -= v;
  }
  int totL = 0, totR = 0;
  for (int e: distL) totL += e;
  for (int e: distR) totR += e;
  double eL = 0, eR = 0;
  for (int e: distL) eL += gain(e,totL);
  for (int e: distR) eR += gain(e,totR);
  double eReduction = upperBoundReduction-( (eL*totL + eR*totR) / (totL+totR) );

  if (eReduction > maxReduction) { bestSplit = i;  maxReduction = eReduction; }
}
return Split.split(col,bestSplit,maxReduction);
```

Parallelizing tree building

- Trees are built in parallel with the Fork/Join framework

```
Statistic left = getStatistic(0,data, seed + LTSSINIT);
Statistic rite = getStatistic(1,data, seed + RTSSINIT);
int c = split.column, s = split.split;
SplitNode nd = new SplitNode(c, s,…);
data.filter(nd,res,left,rite);
FJBuild fj0 = null, fj1 = null;
Split ls = left.split(res[0], depth >= maxdepth);
Split rs = rite.split(res[1], depth >= maxdepth);
if (ls.isLeafNode()) nd.l = new LeafNode(...);
else  fj0 = new FJBuild(ls,res[0],depth+1, seed + LTSINIT);
if (rs.isLeafNode()) nd.r = new LeafNode(...);
else  fj1 = new  FJBuild(rs,res[1],depth+1, seed - RTSINIT);
if (data.rows() > ROWSFORKTRESHOLD)…
fj0.fork();
nd.r = fj1.compute();
nd.l = fj0.join();
```

Challenges

- Found out that Java Random isn't
- Tree size does get to be a challenge
- Need more randomization
- Determinism is needed for debugging

PART III
Playing with DRF

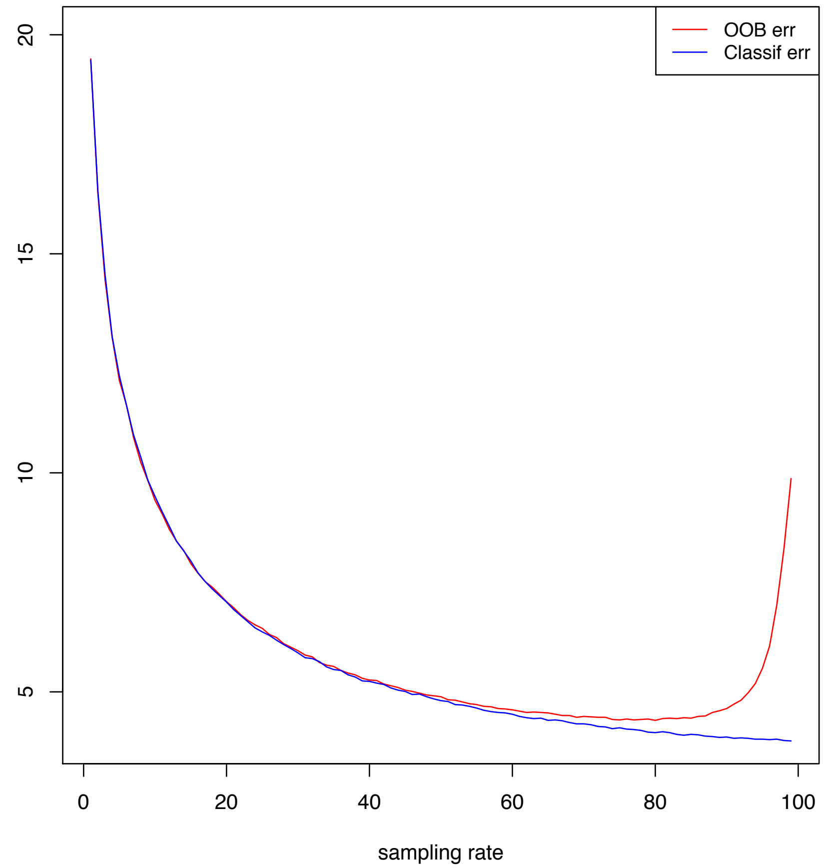
Covtype,
playing with knobs

Photo credit: http://foundwalls.com/winter-snow-forest-pine/

Covtype

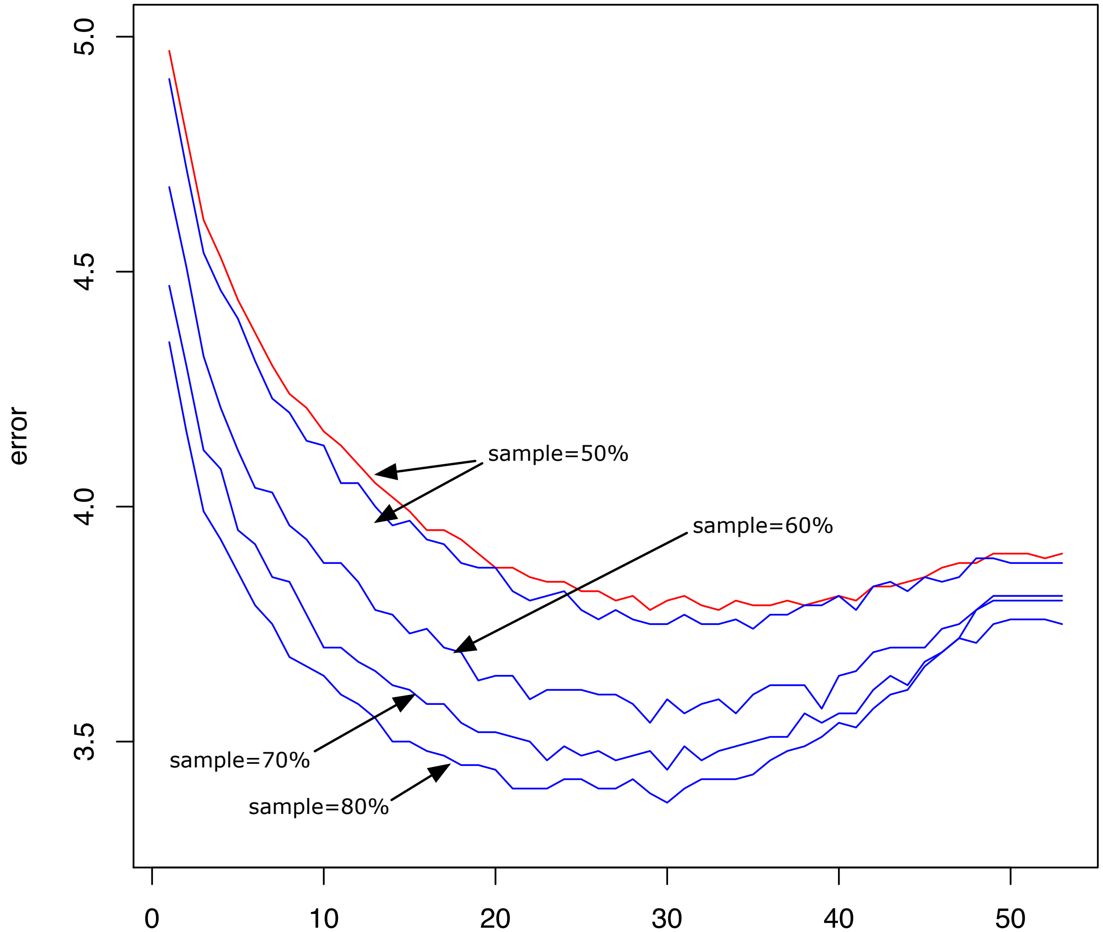
Dataset	Features	Predictor	Instances (train/test)	Imbalanced	Missing observations
Iris	4	3 classes	100/50	NO	0
Vehicle	18	4 classes	564/282	NO	0
Stego	163	3 classes	3,000/4,500	NO	0
Spam	57	2 classes	3,067/1,534	YES	0
Credit	10	2 classes	100,000/50,000	YES	29,731
Intrusion	41	2 classes	125,973/22,544	NO	0
Covtype	54	7 classes	387,342/193,672	YES	0

Varying sampling rate for covtype



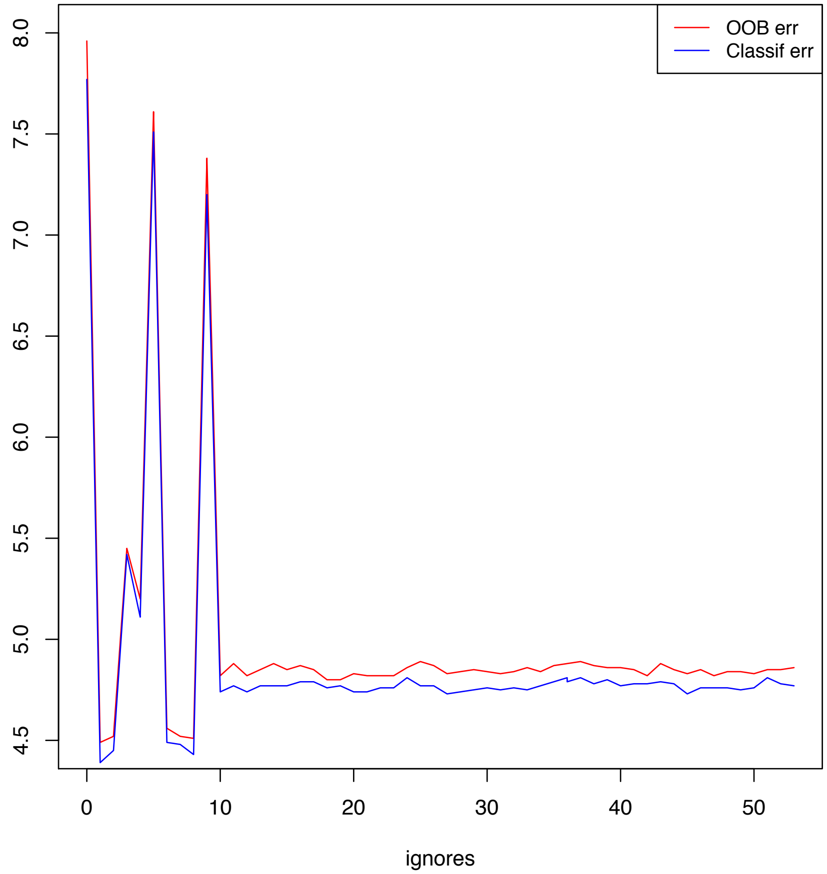
- Changing the proportion of data used for each tree affects error
- The danger is overfitting; and loosing the OOBE

Changing #feature / split for covtype



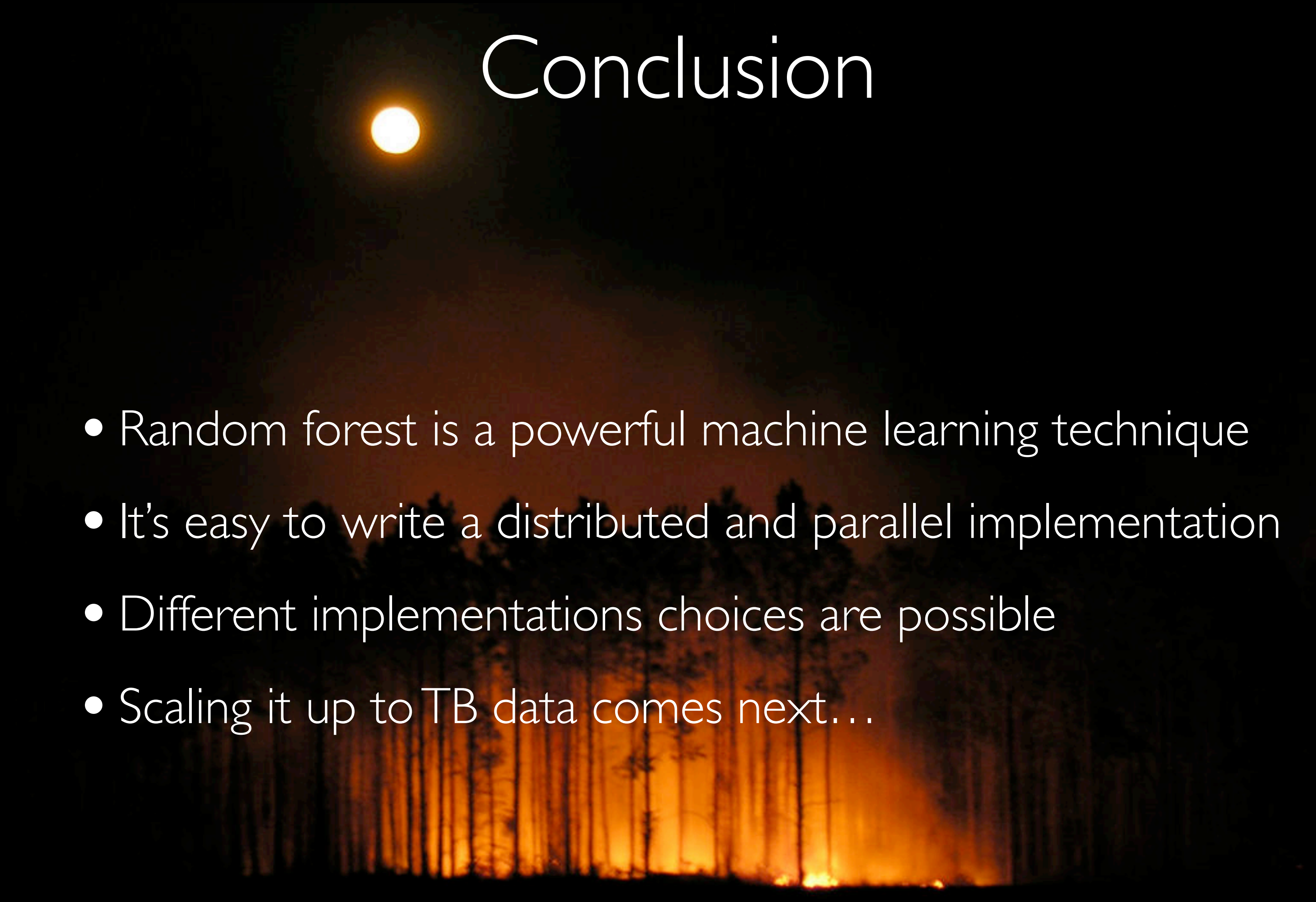
- Increasing the number of features can be beneficial
- Impact is not huge though

Ignoring features for covtype



- Some features are best ignored

Conclusion



- Random forest is a powerful machine learning technique
- It's easy to write a distributed and parallel implementation
- Different implementations choices are possible
- Scaling it up to TB data comes next…

Photo credit www.twitsnaps.com