28 Sept 1998

Improving Accuracy by Voting Classification Algorithms

Ronny Kohavi

Silicon Graphics, Inc. and Blue Martini LLC

Joint work with Eric Bauer

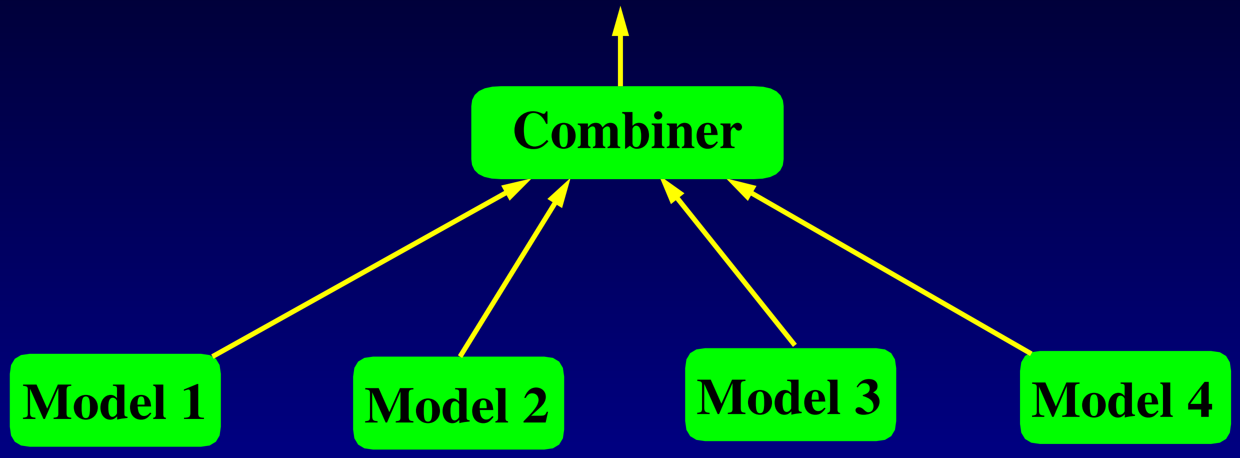
Silicon Graphics, Inc.

Outline

- Introduction to voting methods.
- Experimental design and the Bias–Variance decomposition.
- Bagging: pruning, using prob estimates, wagging, backfitting.
- Boosting: AdaBoost, Arc–X4. Numerical instabilities.
- Open questions.

Introduction to Voting Methods

- Main idea: build multiple models and combine them.



- Variants differ in:
  - How models are built (e.g., change data or change algorithm).
  - How predictions are combined (e.g., uniform vs. non–uniform weighting, multiple levels––stacking).

Key Ingredients

1. Low error rate for models.

2. Diversity, i.e., non–correlated (or anti–correlated) models.

3. Many models.

- It is easy to satisfy #2 and #3 by sacrificing #1: build bad models.

- It is easy to satisfy #1 and #3 by sacrificing #2: build small tweaks to a good model.

Examples of Voting Algorithms

- **Bagging:**
  - Use bootstrap samples (sample with replacement) to create different datasets.
  - Combiner uses uniform weighting.

Modify data

- **Wagging: similar to bagging, but**
  - Reweigh instances instead of sample.

Modify data

- **Randomized splits in trees:**
  - Modify split selection: randomly select (e.g., uniformly) from k best splits.

Modify algo

- **Option trees:**
  - Select top k splits and combine them (at multiple levels of the tree).

Modify algo

Examples of Voting Algorithms (II)

- **Arc–x4:**
  - Increase weight of misclassified instances

Modify data

- **Boosting:**
  - Increase weight of misclassified instances
  - Combine classifiers, giving low error classifiers higher weight.

Modify data

Disadvantage of above Adapting resample and combine algorithms: hard to parallelize. Each classifier is created based on the previous ones.

(Dis)advantages of Voting Methods

Advantages

- Lower error rate.
- Multiple models can give more insight (probably only for uniform combinations).

Disadvantages:

- Loss of comprehensibility:
  - Less structure (except for option trees).
  - Huge models.
- Slower induction.
  May exhaust hardware memory.
- Slower classification time.

Introduction to the Bias–Variance Decomposition

The B+V decomposition is a powerful tool for analyzing induction algorithms.

It holds for finite samples (not in asymptopia).

Given: Target concept, Training set size, Induction algorithm, it provides a decomposition of the error into

- Intrinsic noise (Bayes Optimal)
- Squared bias: how well do hypotheses match the target on average.
- Variance: how much hypotheses vary for different training sets.

The Decomposition

$$E(C) = \sum_x P(x) \left(\text{bias}_x^2 + \text{variance}_x + \sigma_x^2 \right) \tag{1}$$

where

$$\text{bias}_x^2 \equiv \frac{1}{2} \sum_{y \in Y} \left[P(Y_F = y \mid x) - P(Y_H = y \mid x) \right]^2 \tag{2}$$

$$\text{variance}_x \equiv \frac{1}{2} \left(1 - \sum_{y \in Y} P(Y_H = y \mid x)^2 \right) \tag{3}$$

$$\sigma_x^2 \equiv \frac{1}{2} \left(1 - \sum_{y \in Y} P(Y_F = y \mid x)^2 \right) . \tag{4}$$

f and m in the conditioning events are implicit.

Example

Assume that the Boolean label is independent of the attributes (random concept).
The label is 1 with probability (1–p) for p<0.5

Constant classifier: predict 1.

- Bias^2: p^2 (the average guess is off by p).
- Variance: 0 (rock stable guess).

Single rule: predict 1 if A_i=1 (A_i is an attribute that leads to a pure split by chance)

- Bias^2: 0 (on average you predict well).
- Var: p(1–p) (unstable predictions because A_i is a "random" split).

Tree Pruning / Overfitting

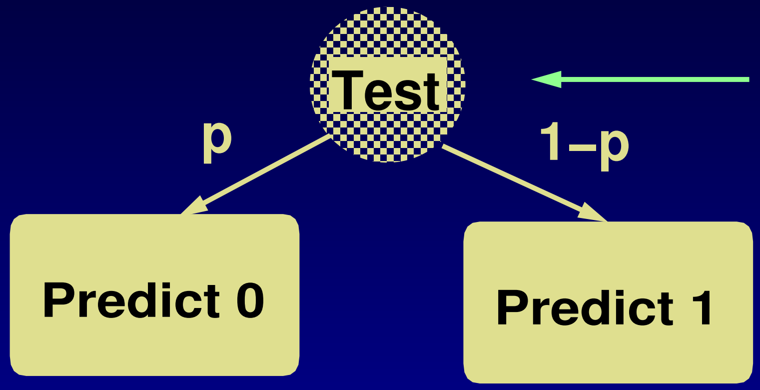
The previous example shows why pruning is useful.

The node is not pure yet we stop and predict majority

$\text{Bias}^2 = p^2$

$\text{Var} = 0$

$\text{Error} = \text{Bias} + \text{Var} = p^2$



We split the node and get two pure nodes with the right probabilities

$\text{Bias}^2 = 0$

$\text{Var} = p(1-p)$

$\text{Error} = \text{Bias} + \text{Var} = p(1-p)$

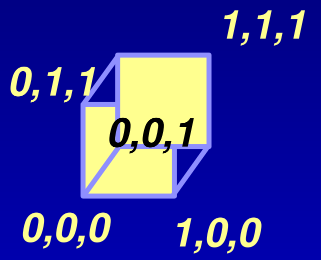
$p^2 < p(1-p)$ if $p < 0.5$, which we assumed.

In this case, it is better not to split. The variance hurts us because we built a structure that is too complex.

Curse of Dimensionality

20 dimensional unit hyper–cube.
100,000 instances uniformly distributed.
What is the expected distance of an instance to its closest neighbor?

- 0.1
- 0.5
- 0.7
- 0.9
- 0.99
- 0.999
- 1.5
- 20.0

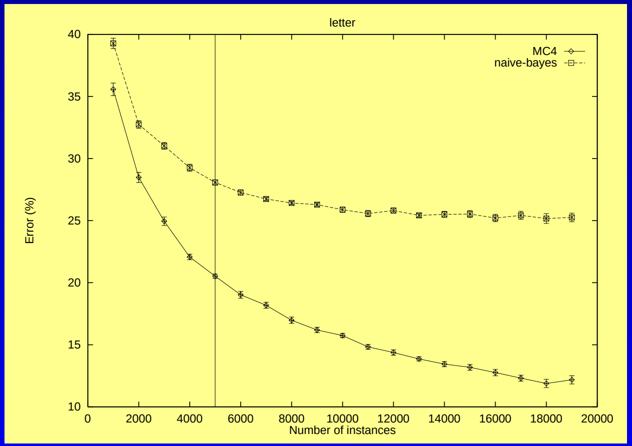


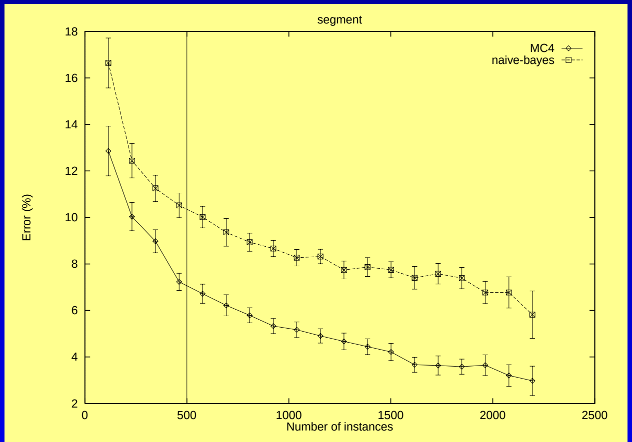
Experimental Design

Details of large experiment by Bauer and Kohavi (to appear in Machine Learning journal).

Desiderata for data sets and sampling sizes:

- Small confidence interval on estimated error. We chose files with >1000 instances.
- There should be room for improvement. Sample sizes chosen based on learning curves so that we know error is not optimal.





Induction Algorithms

- **MC4: similar to C4.5, implemented in MLC++**
  - **No pruning: deactivate pruning.**
  - **Probabilistic estimates: leaves predict distribution (frequency counts).**
  - **(Actual paper has two versions of decision stumps.)**

- **NB: Naive–Bayes with discretized data.**

Bagging

Input: training set S, Inducer $\mathcal{I}$, integer T (number of bootstrap samples).

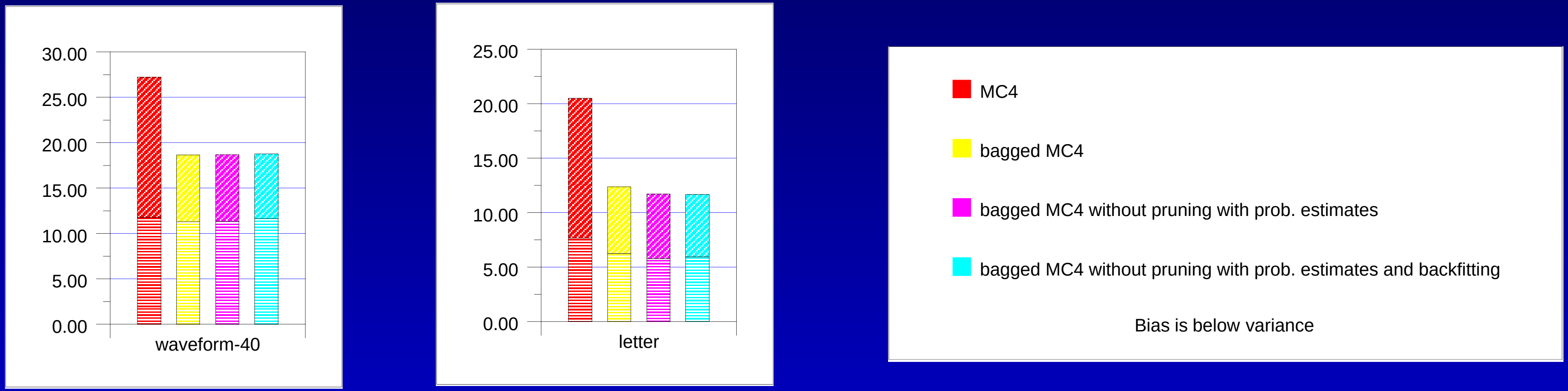
1. for $i = 1$ to T {
2. $\quad S' =$ bootstrap sample from S (i.i.d. sample with replacement).
3. $\quad C_i = \mathcal{I}(S')$
4. }
5. $C^*(x) = \arg\max_{y \in Y} \sum_{i: C_i(x) = y} 1$ (the most often predicted label y)

Output: classifier C^*.

In the experiments, T was set to 25.

Bagging Observations

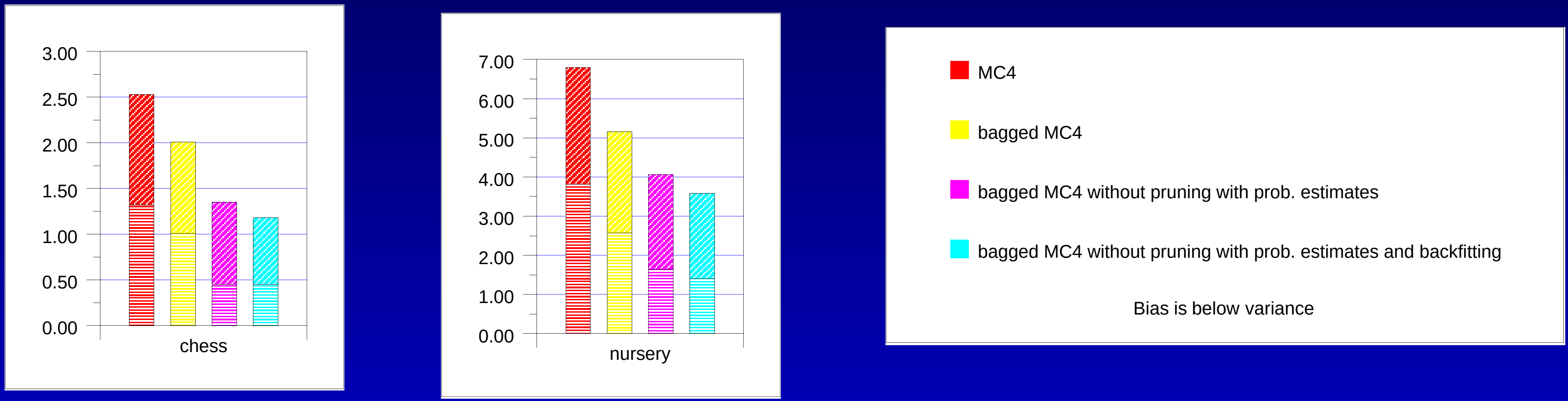
- Bagging was ***uniformly*** better on all 14 datasets!
- Error reduction due to variance reduction. Average relative reduction in err was 29%.



- Trees were larger. Hypothesis: replicated instances seem like strong patterns and pruning is incorrect.

Bagging Observations – Pruning

- If tree pruning is disabled, then
  - Bagged trees are smaller (training set size is effectively smaller–63.2% unique instances).
  - Average bias was reduced by 14% (relative).
  - Average variance grew by 11% (relative).



- "No pruning" did not make an overall difference, but we suspect that with more replicates, it is better not to prune.

Bagging Variants

- Wagging (Weight Aggregation) perturbs the training set weights instead of sampling.

  Results were similar to bagging.

- Backfitting takes the unused data from each bagging replicate (~ 36.8% unique instances) and updates the counts at the leaves.

  Average relative error decreased 3%, which was all due to variance reduction. Variances for *all* files improved!

Boosting

Input: training set S of size m, Inducer $\mathcal{I}$, integer T (number of trials).

1. $S' = S$ with instance weights assigned to be 1.
2. For $i = 1$ to T {
3. $C_i = \mathcal{I}(S')$
4. $\epsilon_i = \frac{1}{m} \sum_{x_j \in S' : C_i(x_j) \neq y_j} \text{weight}(x)$ (weighted error on training set).
5. If $\epsilon_i > 1/2$, set S' to a bootstrap sample from S with weight 1 for every instance and goto step 3 (this step is limited to 25 times after which we exit the loop).
6. $\beta_i = \epsilon_i / (1 - \epsilon_i)$
7. For-each x_j, divide weight(x_j) by $2\epsilon_i$ if $C_i(x_j) \neq y_j$ and $2(1-\epsilon_i)$ otherwise
8. }
9. $C^*(x) = \arg\max_{y \in Y} \sum_{i : C_i(x) = y} \log \frac{1}{\beta_i}$

Output: classifier C^*.

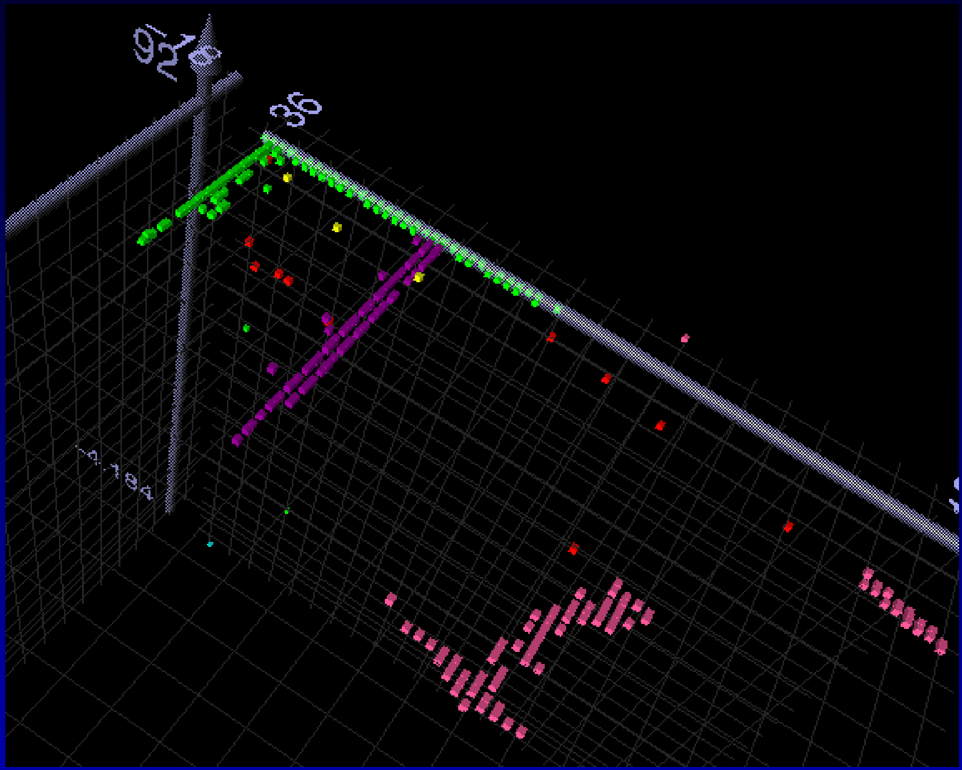
Observations on Boosting

- Incorrect instances are weighted by a factor inversely proportional to the training set error (1/2e).

  A training set error of 0.1% will cause weights to grow by a factor of 500.
  Without careful attention, numerical precision problems occur.

- The total weight of the misclassified instances is half the original training set weight.
  The correctly classified instances get the other half of the total weight.

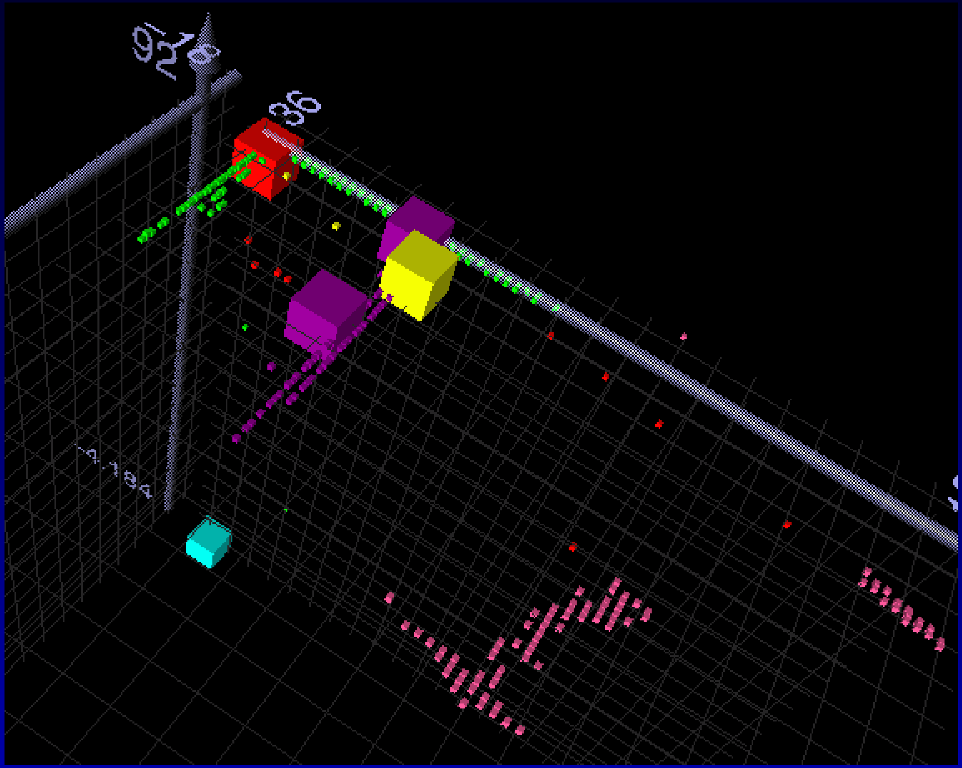
Running Example – Shuttle (I)



Test–set error: 0.38%

Five misclassified examples on training set of size 5,000 (0.1%) causes their weight to be 500.

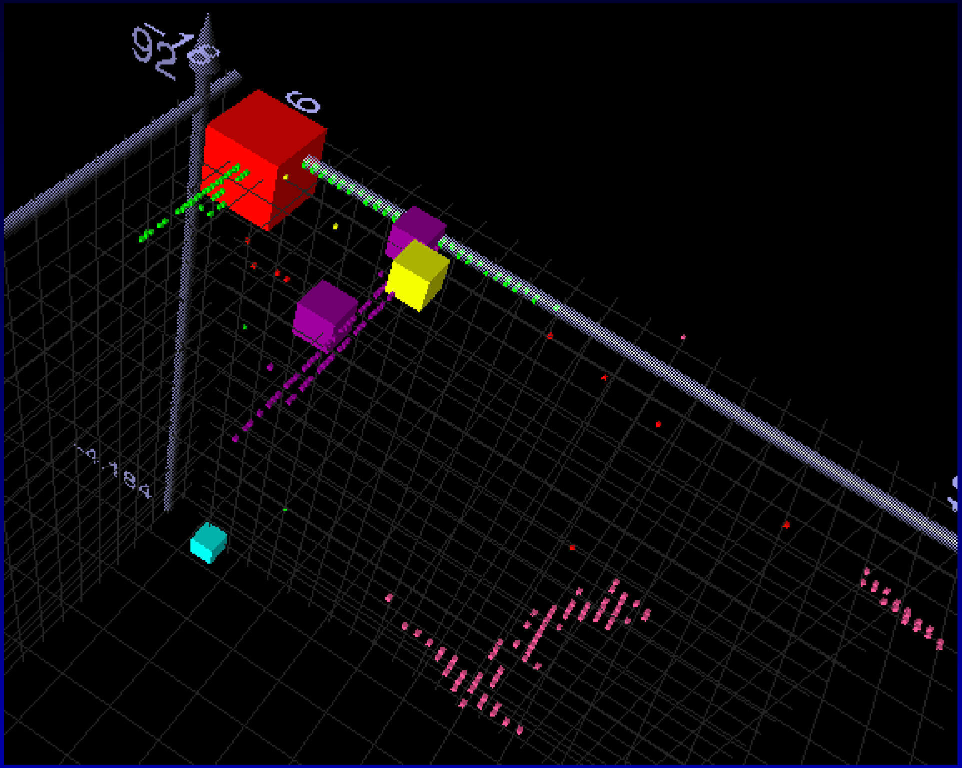
Running Example – Shuttle (II)



Test–set error: 0.19%

One misclassified example (0.01%) that was not previously misclassified is reweighted from 0.5 to 2500.

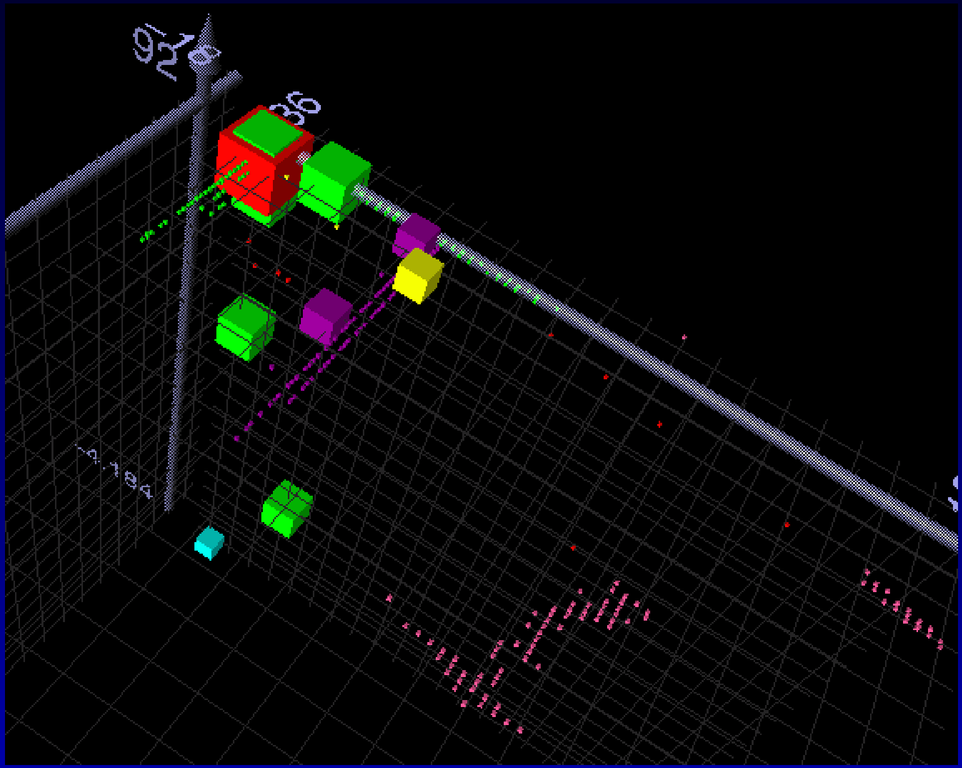
Running Example – Shuttle (III)



Test–set error: 0.21%

Five mistakes again, all on instances previously correctly classified.

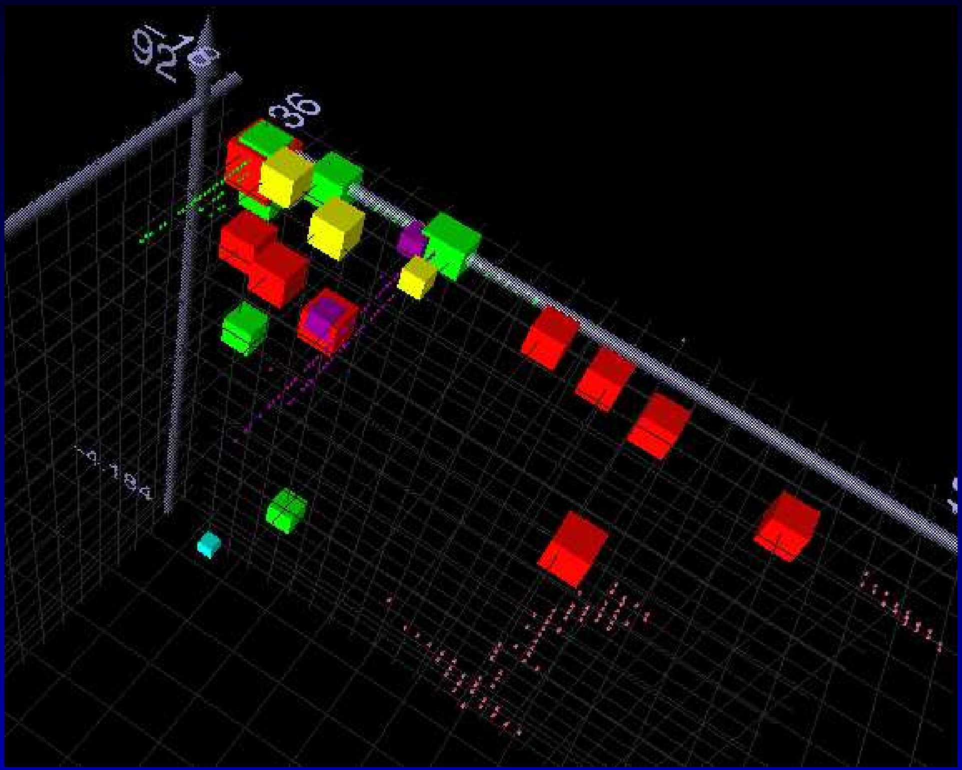
Running Example – Shuttle (IV)



Test–set error: 0.45%

12 mistakes are made.

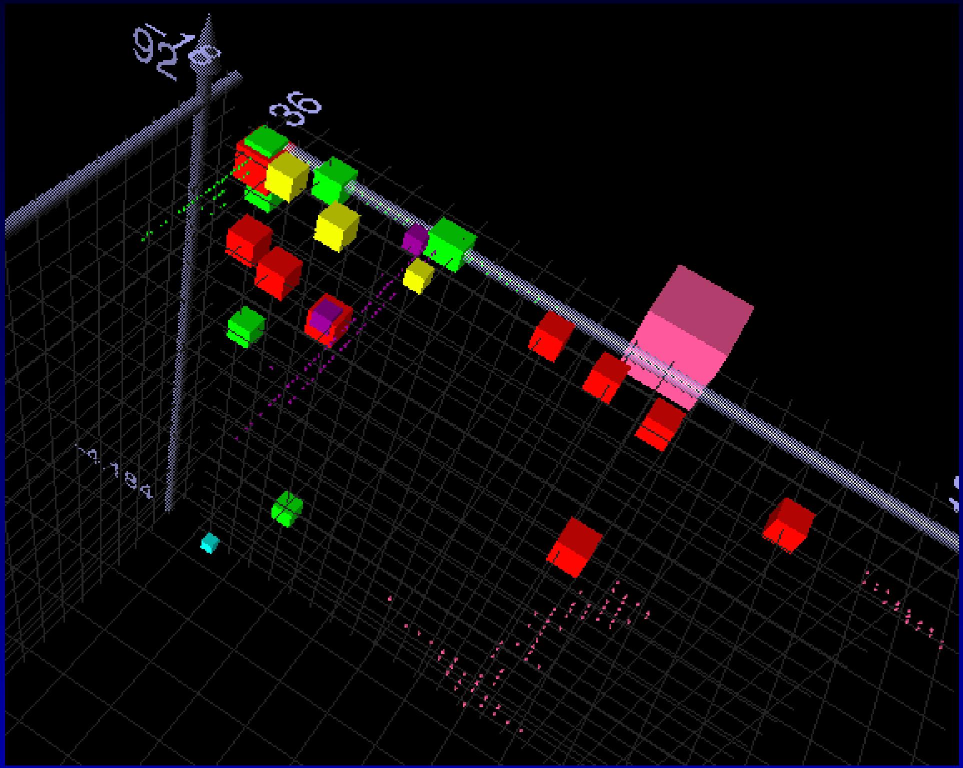
Running Example – Shuttle (V)



If original AdaBoost is used, beta is 0.0000125, which causes weights to go below 10^{-6} prior to normalization. Underflow problems start...

One misclassified example with weight 0.063. Training set error is 0.0012%.

Running Example – Shuttle (VI)



Test–set error: 0.08%

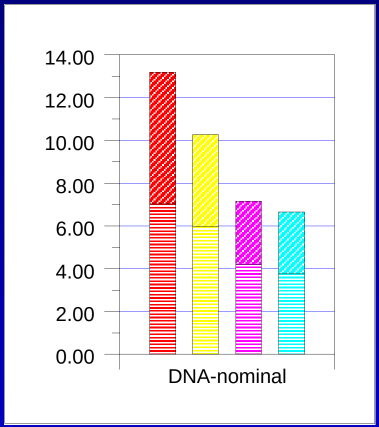
Classifier makes no mistakes.
Note that this is a single classifier, which is significantly better than the original one!

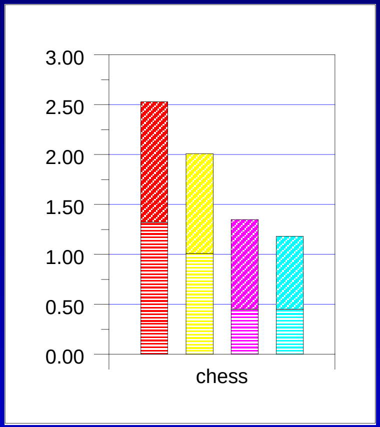
AdaBoost Observations

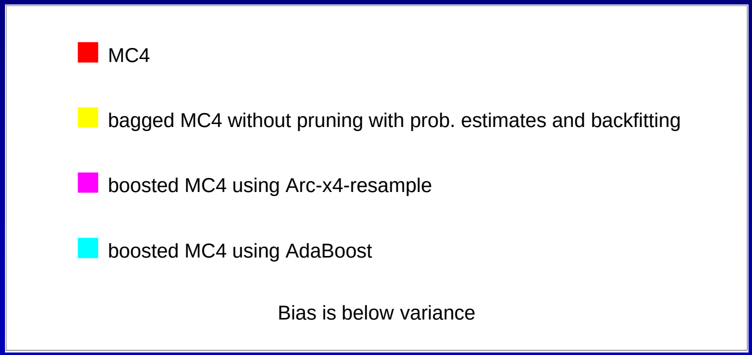
- AdaBoost slightly outperformed Bagging.
- Unlike Bagging, boosting did not uniformly reduce the error.
  Hypothyroid, sick–euthyroid, adult, and LED–24 had higher errors.
- Average tree size was larger for most files. It was especially larger for files on which performance degraded.
- Problems with robustness to noise.

Boosting: Bias + Variance

- Boosting reduced both bias and variance:
  Average bias reduced 32% (relative).
  Average variance reduced 16% (relative).







Open Questions

- Can AdaBoost be made more robust to noise?
- Arc–X4 did not work with reweighting. Why?
- Can we learn a single model that is better (as happened with shuttle)?
- Bagging and Boosting build huge structures. What happened to Occam's razor? Is there a compact representation?
- Bagging worked better without pruning. AdaBoost did not. Why?
- Boosting is sequential. Can parallelism be used?

Summary

- AdaBoost reduced the error by 27% with MC4 and 24% with Naive–Bayes (relative).
  Note however that we knew improvement was possible on our datasets.

- Bagging reduces variance. AdaBoost reduces both bias and variance.

- Bagging benefits from no pruning, probabilistic variants, and backfitting.

- Be careful with numerical instabilities when implementing AdaBoost.

CPU #55 on Flurry

- We used about 4,000 CPU hours.
  Many runs were done on Flurry, a 128 CPU Origin 2000 with 30GB of RAM.
- We spent a lot of time trying to track an assertion failure, where sometimes normalizing an array did not add up to 1.0.

  After many experiments, we found that CPU#55 on Flurry was making arithmetic errors sometimes...
- Today the OS runs a program called paranoia on large machines to track such problems.