Data Mining: Ensembles

Mirek Riedewald

Key Learning Goals

- Summarize in three sentences the main idea behind bagging.
- What are the requirements on the individual models for bagging to improve prediction quality?
- Write the pseudo-code for training a bagged ensemble. Try to do it for all three partitioning approaches.
- Given a concrete example, be able to quantify the data transfer for training and prediction for a bagged ensemble.

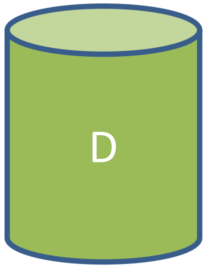
Introduction

- Ensemble methods like bagging are among the best classification and prediction techniques in terms of prediction quality. Their main drawback are high training and prediction cost. This makes them ideal candidates for use in a distributed environment.

Ensemble Methods

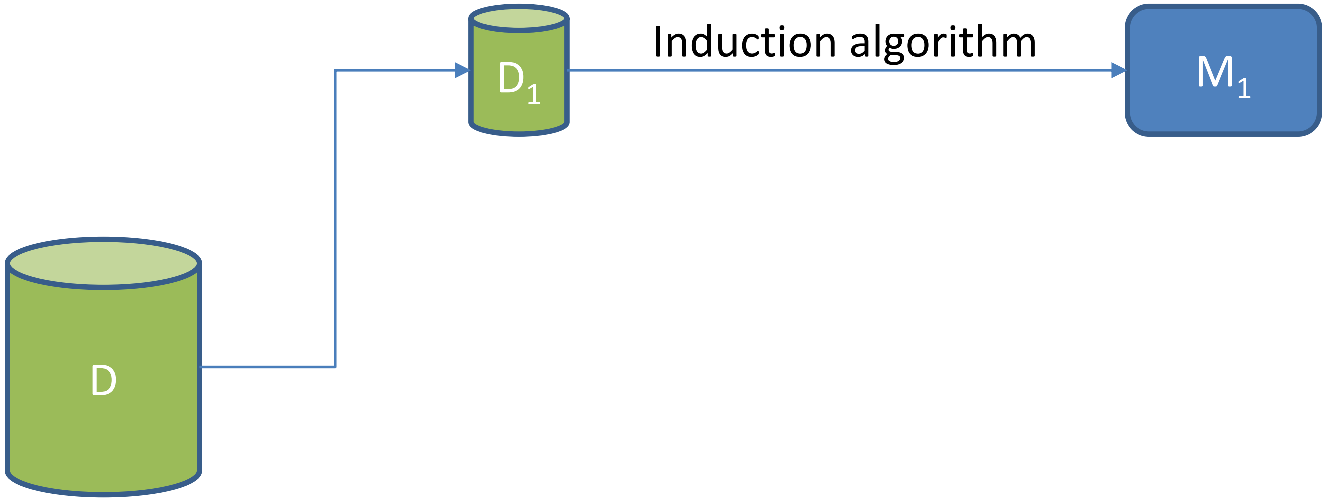
- Ensemble methods for classification and prediction rely on a pool of models to make better predictions than each individual model on its own. Many different types of ensemble methods exist, the most well-known general approaches being bagging and boosting.
- This module focuses on bagging, which is particularly well-suited for distributed computation because each model in the ensemble can be trained independently on a different data set D_i derived from the given labeled training data D.
- To make a prediction for a test record, the individual predictions of all models in the ensemble are aggregated appropriately.
- The next pages illustrate this idea.

Training an Ensemble



Given training data set

Training an Ensemble

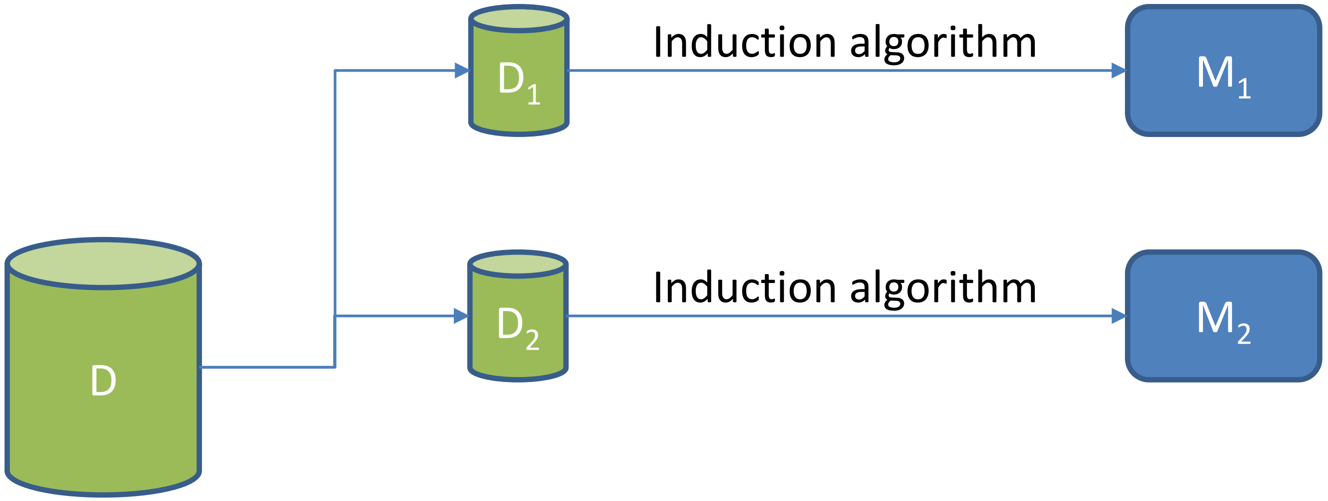


Trained models

Given training data set

Multiple training sets derived from D

Training an Ensemble

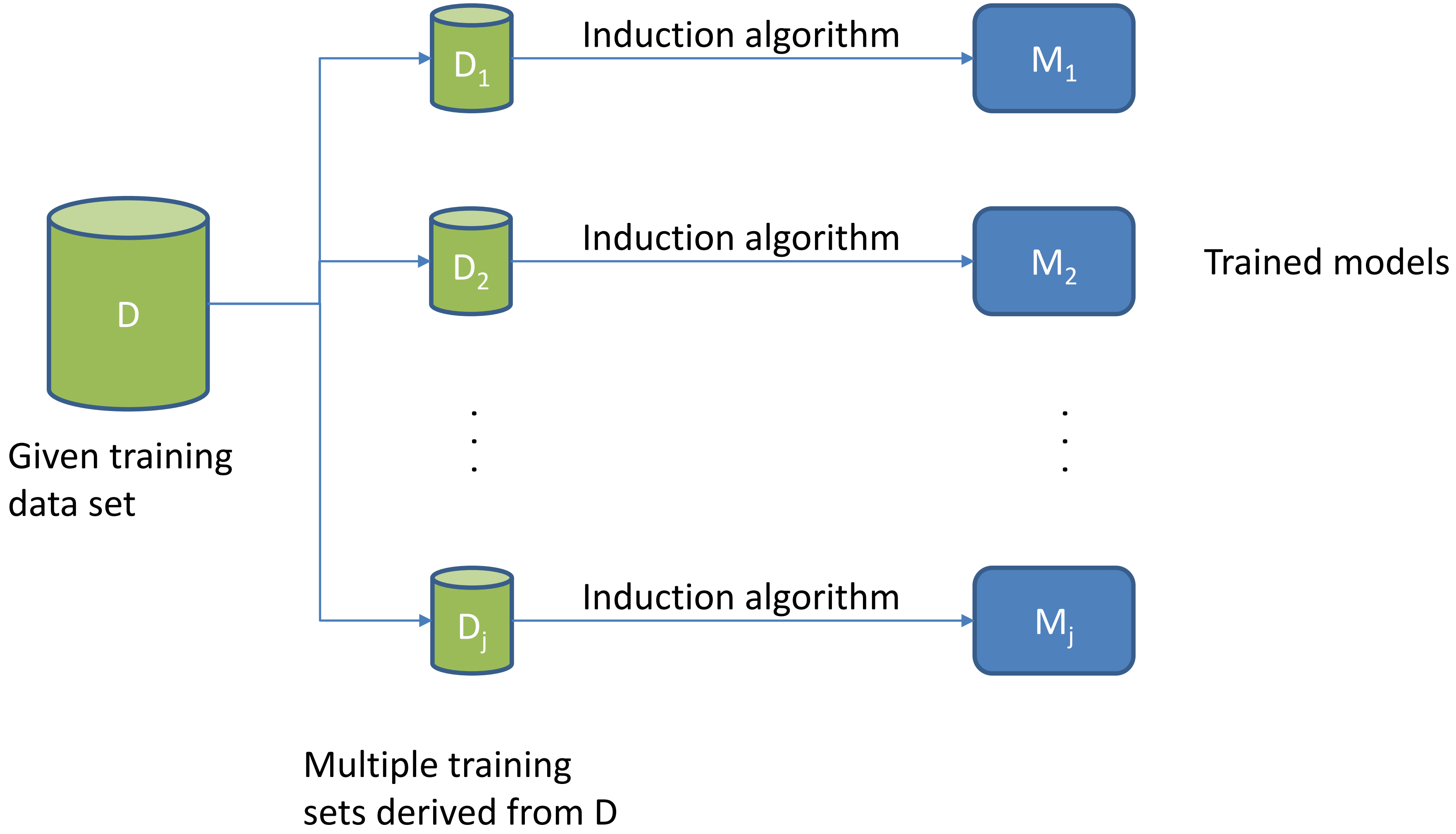


Trained models

Given training data set

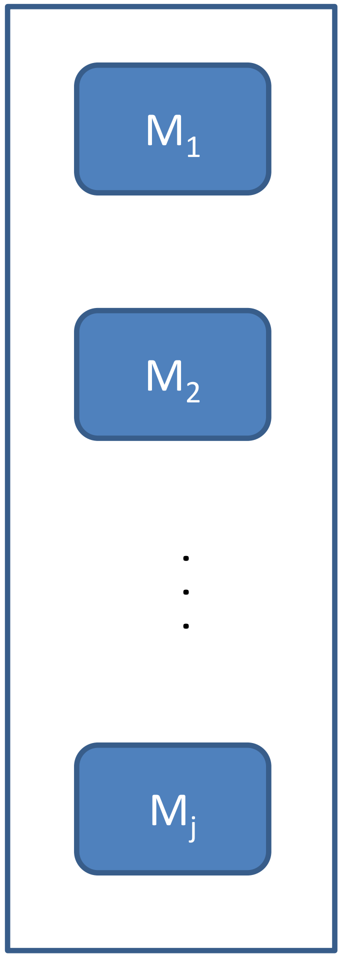
Multiple training sets derived from D

Training an Ensemble

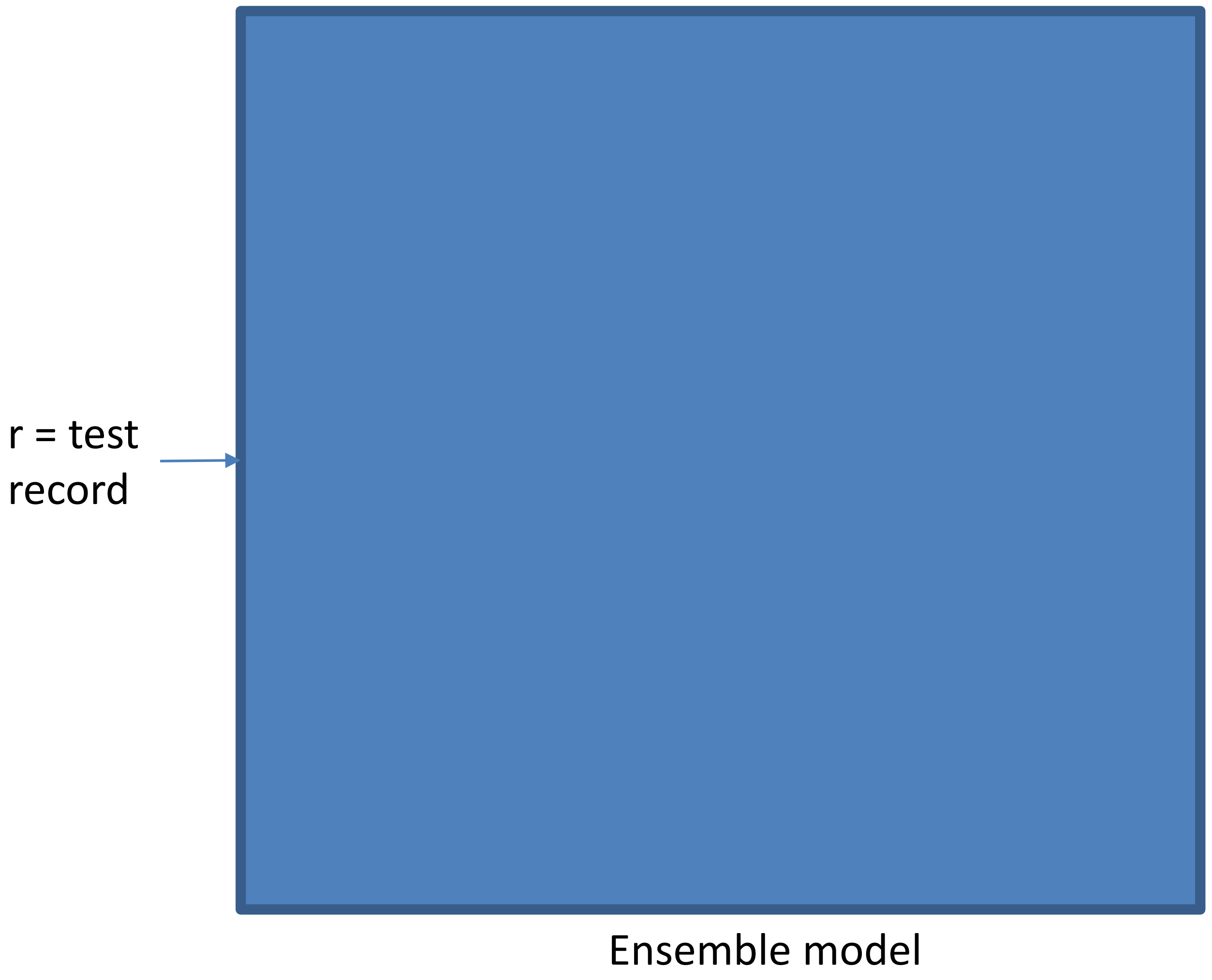


Training an Ensemble

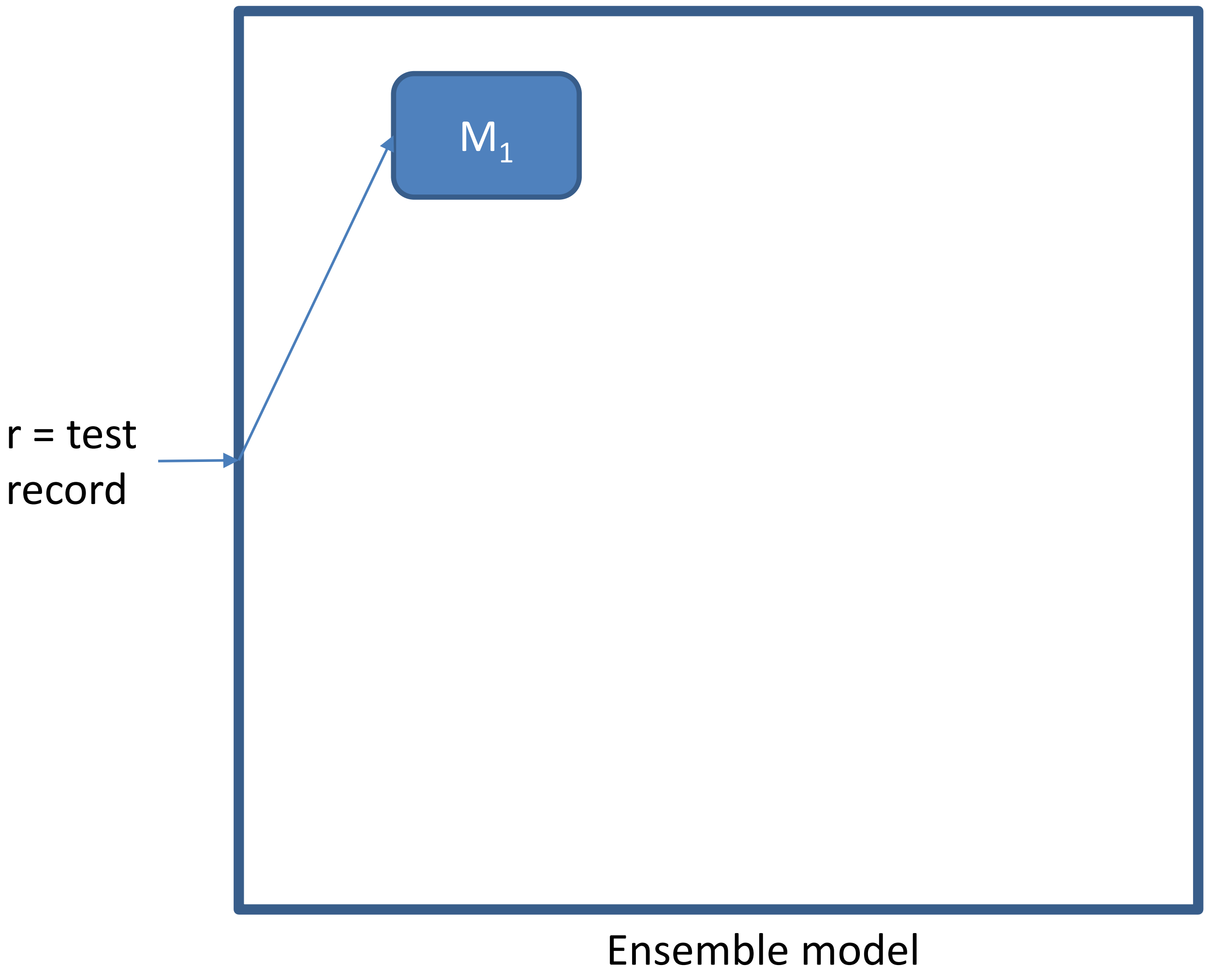
Ensemble model



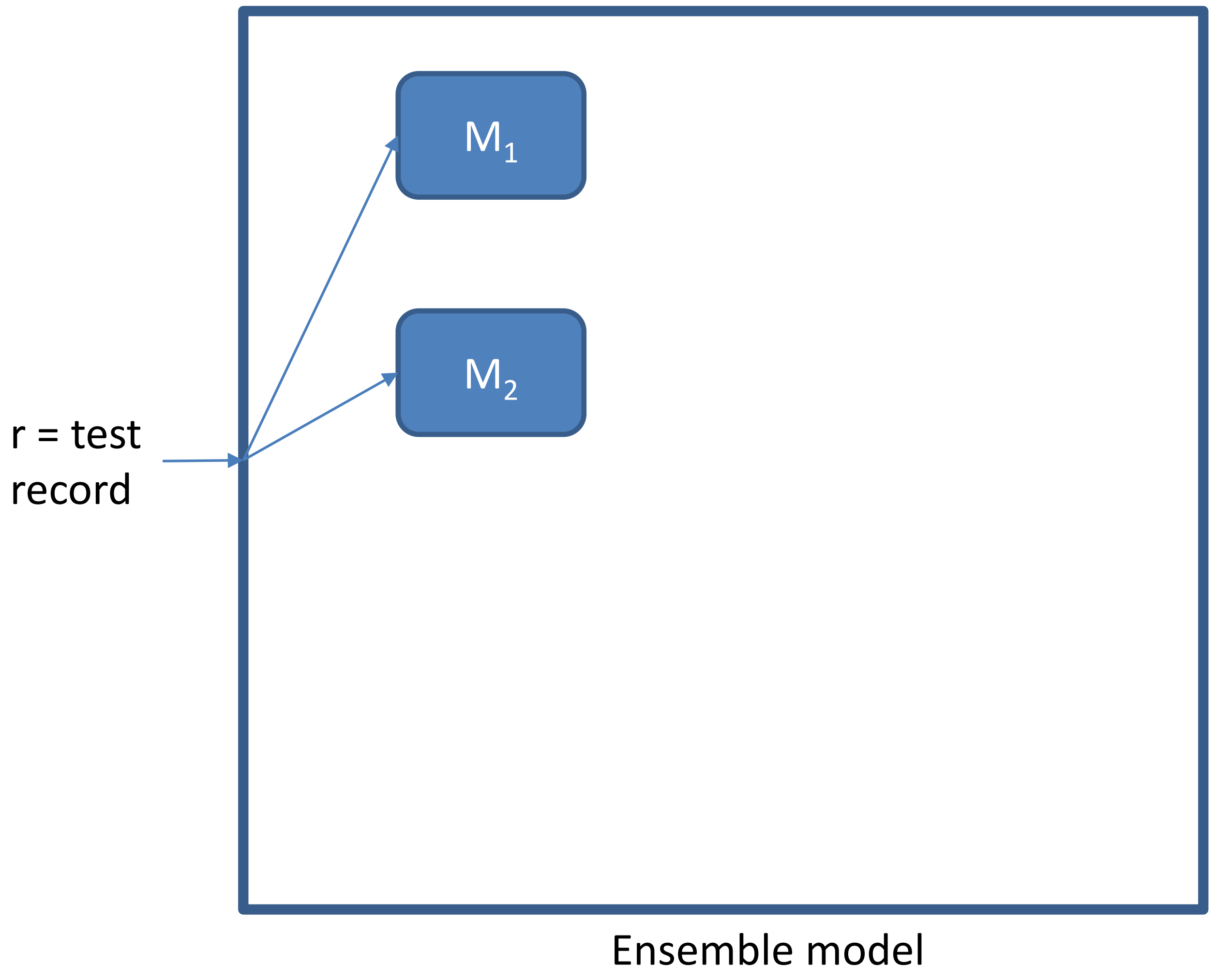
Making Predictions



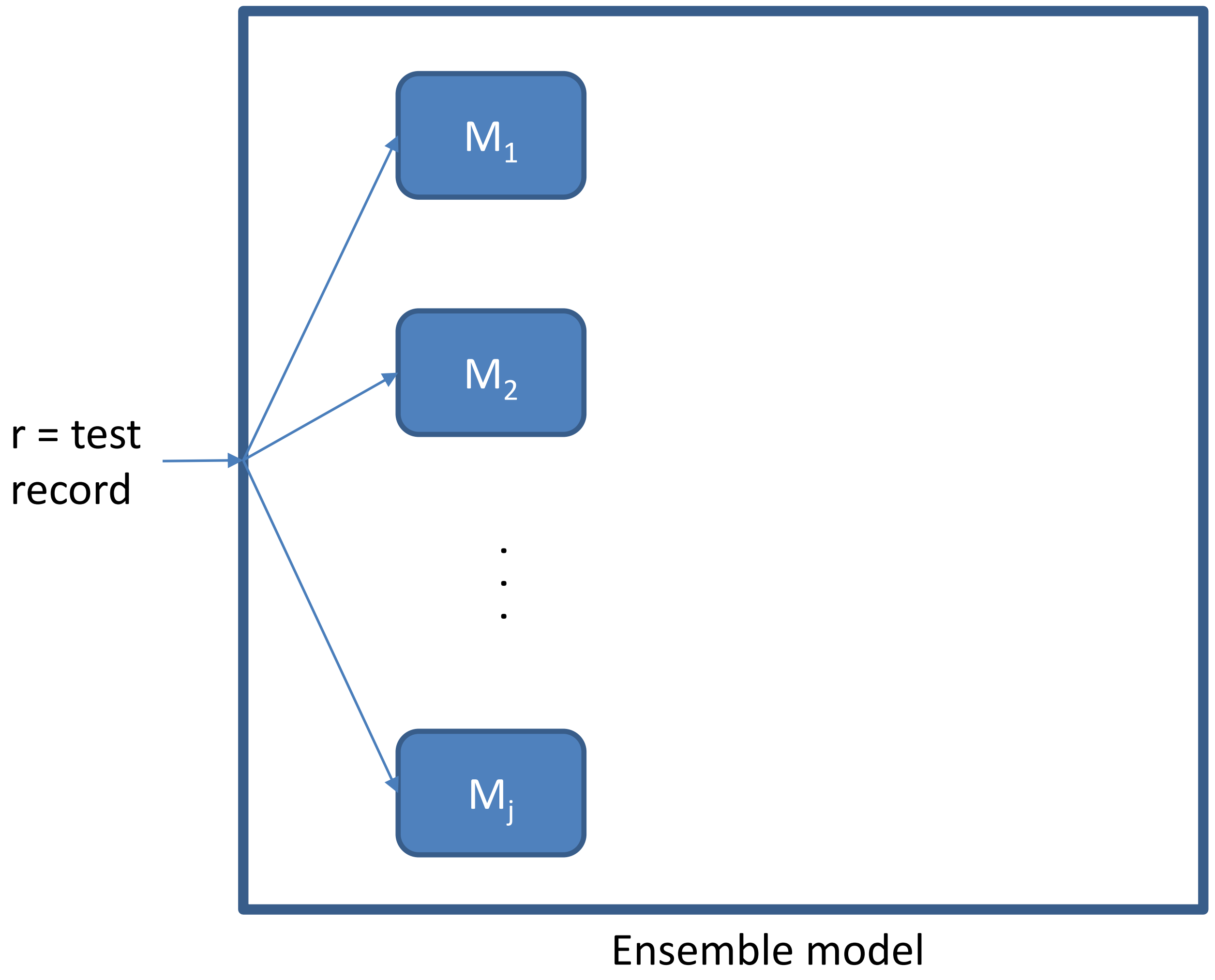
Making Predictions



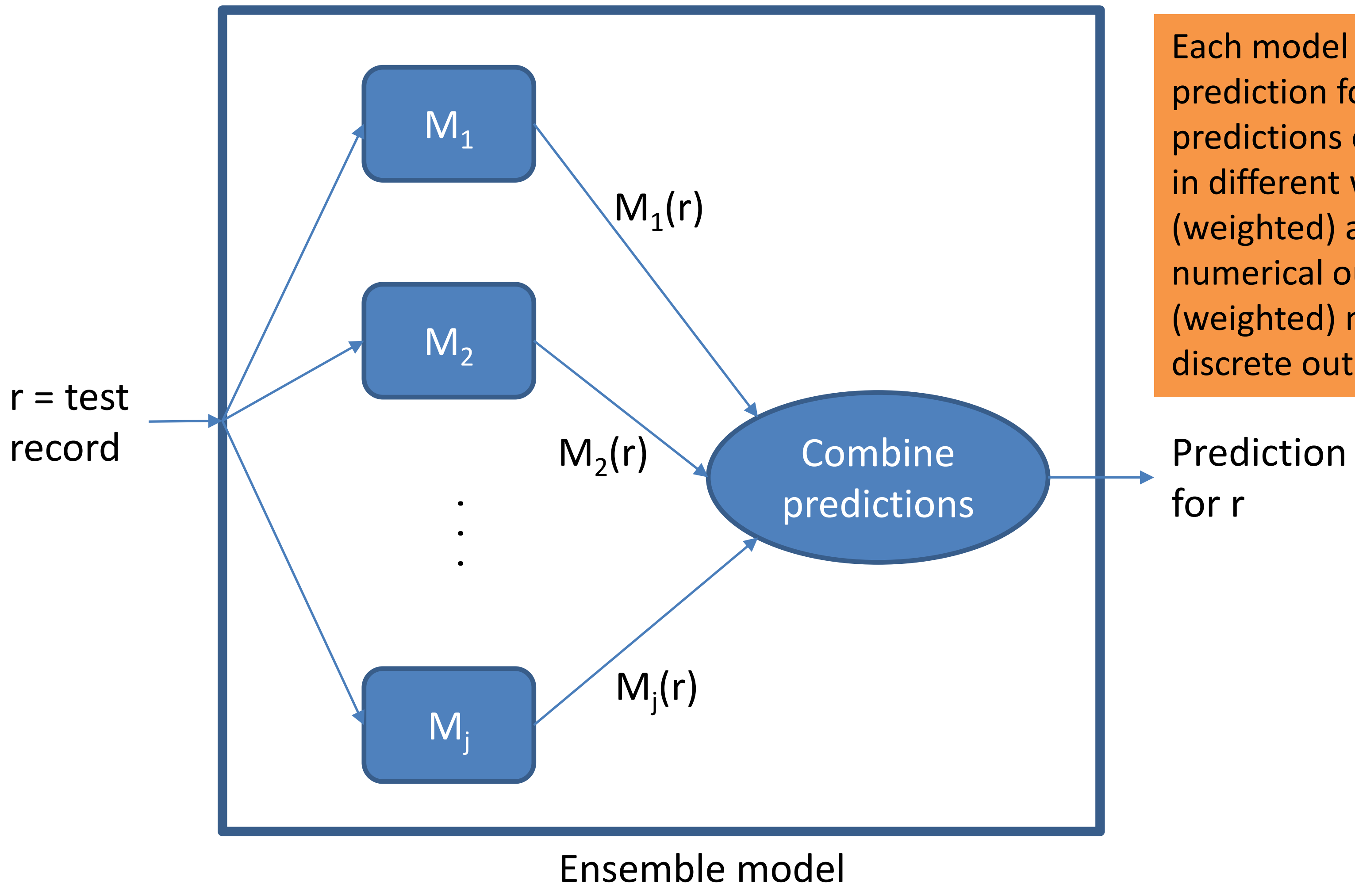
Making Predictions



Making Predictions



Making Predictions



Each model makes a prediction for r. Individual predictions can be combined in different ways, e.g., by (weighted) averaging for numerical output or (weighted) majority vote for discrete output.

Intuition For Ensembles

- Ensemble methods are based on the following intuition: Consider a scenario where you seek advice from your friends about taking CS 6240. Assume each friend individually gives you good advice most of the time. By asking more friends and following the majority recommendation, your probability of getting good advice will increase significantly.
- Let us make this concrete now.

The Power of Friendship

- Assume each friend individually gives helpful advice 60% of the time, i.e., each friend's error rate is 0.4. You ask 11 friends about taking CS 6240. Each of them responds with either Yes or No. You follow the majority advice.
- The error rate of the 11-friend ensemble is the sum of the probability of exactly 6, 7, 8, 9, 10, or 11 friends giving *unhelpful* advice. If each friend's recommendation is independent of the others, then this probability can be expressed as

$$\sum_{i=6}^{11} \binom{11}{i} 0.4^{i}(1-0.4)^{11-i} = 0.25$$

- Stated differently, the ensemble of 11 independent friends gives helpful advice 75% of the time, which is significantly better than the 60% of each individual friend. Asking more such friends will result in even better advice.

When Does Model Averaging Work?

- The example illustrates an ensemble approach called bagging, short for bootstrap aggregation. For it to improve prediction quality, two conditions must hold:
  - **Independence**: The individual models (friends in the example) make their decisions independently. If one friend influences the others, then the combined advice will be as good or bad as that individual.
  - **Better than 50-50**: Each model (friend) individually has an error rate below 50%, otherwise the error rate actually increases with the number of friends.

Model Averaging and Bias-Variance Tradeoff

- The example of combining advice from multiple friends provides anecdotal evidence why ensembles improve prediction quality. This property can also be shown more generally.
- The key to understanding why model averaging improves predictions lies in the bias-variance tradeoff. Intuitively, a model's total prediction error consists of two components affected by the choice of model: bias and variance. For *individual* models, lowering one tends to increase the other. Through model parameters, e.g., the height of a decision tree, the tradeoff between bias and variance can be adjusted.
- Bagging can overcome this tradeoff as follows:
    1. Train individual models that overfit, i.e., have low bias, but potentially high variance.
    2. Reduce the variance by averaging many of these models.
- Bagging can be applied to any type of classification and prediction model. An ensemble may even contain models of different types, e.g., trees, SVMs, ANNs, and Bayesian networks.

Intuition for Model Error

- A prediction model is a function $f: X \to Y$ that returns a prediction $f(x)$ for input x. When the model is trained using training data d, then we write $f(x; d)$ to highlight the dependence on d.
- We train f to represent an unknown true distribution that we can think of as a set of pairs (x, y), where x is a vector of 1 or more input-attribute values and y is the output value. (The training data is generally assumed to be a random sample from this distribution.) To measure model quality for prediction/regression, we use squared error, defined as the squared difference $(f(x) - y)^2$ between predicted and true output for a given input x.
- Example: To predict income from age, X is age and Y is income. (When predicting income from age and highestDegree, X is a 2-dimensional vector (age, highestDegree), and so on.)
  - Assume the model predicts $f(\mathrm{age} = 30) = 80$, while the true distribution has incomes 50, 70, 70, and 90 associated with age 30.
  - Total squared error for age 30 then is $(50 - 80)^2 + (70 - 80)^2 + (70 - 80)^2 + (90 - 80)^2 = 1200$.
  - A better prediction would be the average income for age 30, i.e., 70: $(50 - 70)^2 + (70 - 70)^2 + (70 - 70)^2 + (90 - 70)^2 = 800$.

Intuition for Model Error (cont.)

- The total model error is the sum of squared errors over the entire distribution of (X, Y) combinations. For discrete input this is $\sum_{(x,y)\in(X,Y)}(y - f(x; d))^2$. (It would be the corresponding integral for continuous cases.)
- How can we take the impact of the training data into account? Let D be the set of possible training datasets from which the actual training dataset d is randomly picked. Then the error over all possible training sets is

$$\sum_{d\in D} \sum_{(x,y)\in(X,Y)} (y - f(x; d))^2$$

  - It may seem more intuitive to consider the average error per training set, i.e., divide the above formula by $|D|$. However, since all models are evaluated against the same D, $|D|$ is a constant that is independent of model choice. It therefore does not impact the minimization over possible models f.

Mathematical Derivation of the Bias-Variance Tradeoff

- The bias-variance tradeoff can be derived using statistical decision theory. Consider 3 random variables X, Y, and D, where
  - X takes on any m-tuple of real numbers to represent a vector of m input-attribute values,
  - Y takes on any real number to represent the output-attribute value, and
  - D takes on any set of $(m+1)$-tuples of real numbers to represent a training dataset.
- The training data is drawn from the true joint distribution of (X, Y) pairs. We want to learn this unknown distribution by constructing from the training data a function $f(X)$, the prediction model, that returns good approximations of the true Y for a given input X. To make the dependence of the model on the training data explicit, we write $f(X; D)$.
- The best model is the one with lowest mean squared error over X, Y, and D. How can we mathematically express this error as a function of model f?
  - Assume model f was trained on data $D = d$. Its prediction error for input $X = x$ is defined as $E_Y[(Y - f(X; D))^2 \mid X = x, D = d]$, i.e., the expected squared difference between model prediction $f(\mathrm{x}; \mathrm{d})$ and the true output values Y associated with input x.
  - When considering all inputs, the expected error is the expectation over X: $E_X\big[E_Y[(Y - f(X; D))^2 \mid X, D = d]\big]$.
  - When considering all training datasets, the expected error is the expectation over D: $E_D\Big[E_X\big[E_Y[(Y - f(X; D))^2 \mid X, D]\big]\Big]$.
- The last formula can be rewritten as $E_X E_D E_Y[(Y - f(X; D))^2 \mid X, D]$.

Why Optimize for this Expectation?

- Here is the formula again: $E_X E_D E_Y[(Y - f(X;D))^2 \mid X, D]$
- E_X: We do not want a model that makes good predictions only for some specific inputs, while producing large errors for others.
- E_D: Similarly, our induction method should create an accurate model not only for certain “lucky” training datasets, while creating a poor model for others.
- E_Y: Similarly, we need to consider all possible outputs.
- Optimizing for the expectation over all combinations ensures that, no matter what training data is presented to us, we will strive for a model that works well for the entire distribution of input-output combinations.
- We discuss next how to transform the formula to a format that clearly shows bias and variance.

First consider inner term $\boldsymbol{E_Y\left[(Y - f(X; D))^2 \mid X, D\right]}$.
Using $Y - f(X; D) = (\mathrm{Y} - \mathrm{E}[\mathrm{Y} \mid \mathrm{X}]) + (\mathrm{E}[\mathrm{Y} \mid \mathrm{X}] - \mathrm{f}(\mathrm{X}; \mathrm{D}))$, we derive

$$
\begin{aligned}
& E_Y[(Y - f(X; D))^2 \mid X, D] \\
& = E_Y[(\mathrm{Y} - \mathrm{E}[\mathrm{Y} \mid \mathrm{X}])^2 \mid X, D] + 2E_Y\left[(Y - E[Y \mid X])\left(E[Y \mid X] - f(X; D)\right) \mid X, D\right] \\
& + E_Y[(\mathrm{E}[\mathrm{Y} \mid \mathrm{X}] - \mathrm{f}(\mathrm{X}; \mathrm{D}))^2 \mid X, D]
\end{aligned}
$$

The second term $E_Y\left[(\mathrm{Y} - \mathrm{E}[\mathrm{Y} \mid \mathrm{X}])\left(\mathrm{E}[\mathrm{Y} \mid \mathrm{X}] - \mathrm{f}(\mathrm{X}; \mathrm{D})\right) \mid X, D\right]$ is equal to

$$
\begin{aligned}
& E_Y[YE[Y \mid X] - Yf(X; D) - E^2[Y \mid X] + E[Y \mid X]f(X; D) \mid X, D] \\
& = E_Y[YE[Y \mid X] \mid X, D] - E_Y[Yf(X; D) \mid X, D] \\
& - E_Y[E^2[Y \mid X] \mid X, D] + E_Y\, [E[Y \mid X]f(X; D) \mid X, D]
\end{aligned}
$$

Note that E[Y | X] does not change with Y. And neither Y nor E[Y | X] depend on D, hence

$$E_Y[YE[Y \mid X] \mid X, D] = E[Y \mid X]E_Y[Y \mid X] = E^2[Y \mid X]$$

Similarly, since f(X; D) does not depend on Y:

$$E_Y[Yf(X; D) \mid X, D] = f(X; D)E[Y \mid X]$$

and

$$E_Y[E^2[Y \mid X] \mid X, D] = E^2[Y \mid X]$$

and

$$E_Y[E[Y \mid X]f(X; D) \mid X, D] = f(X; D)E[Y \mid X]$$

Hence the second term cancels out to zero.

Reminder: for the inner term of the squared error of the learned model f we derived

$$
\begin{aligned}
&E_Y[(Y - f(X;D))^2 \mid X, D] \\
&= E_Y[(\mathrm{Y} - \mathrm{E}[\mathrm{Y} \mid \mathrm{X}])^2 \mid X, D] + 2E_Y\big[(Y - E[Y \mid X])\big(E[Y \mid X] - f(X;D)\big) \mid X, D\big] \\
&+ E_Y[(\mathrm{E}[\mathrm{Y} \mid \mathrm{X}] - \mathrm{f}(\mathrm{X};\mathrm{D}))^2 \mid X, D]
\end{aligned}
$$

and showed $E_Y\big[(Y - E[Y \mid X])\big(E[Y \mid X] - f(X;D)\big) \mid X, D\big] = 0$.

Now we turn our attention to the third term $E_Y[(\mathrm{E}[\mathrm{Y} \mid \mathrm{X}] - \mathrm{f}(\mathrm{X};\mathrm{D}))^2 \mid X, D]$. Since both E[Y | X] and f(X; D) do not depend on Y, we obtain

$$E_Y[(\mathrm{E}[\mathrm{Y} \mid \mathrm{X}] - \mathrm{f}(\mathrm{X};\mathrm{D}))^2 \mid X, D] = (\mathrm{E}[\mathrm{Y} \mid \mathrm{X}] - \mathrm{f}(\mathrm{X};\mathrm{D}))^2$$

Putting it all together, we derive

$$E_Y[(Y - f(X;D))^2 \mid X, D] = E_Y[(\mathrm{Y} - \mathrm{E}[\mathrm{Y} \mid \mathrm{X}])^2 \mid X, D] + (\mathrm{E}[\mathrm{Y} \mid \mathrm{X}] - \mathrm{f}(\mathrm{X};\mathrm{D}))^2$$

This means that so far we have shown for squared error of the model:

$$
\begin{aligned}
&E_X E_D E_Y[(Y - f(X;D))^2 \mid X, D] \\
&= E_X E_D[E_Y[(\mathrm{Y} - \mathrm{E}[\mathrm{Y} \mid \mathrm{X}])^2 \mid X, D] + (\mathrm{E}[\mathrm{Y} \mid \mathrm{X}] - \mathrm{f}(\mathrm{X};\mathrm{D}))^2] \\
&= E_X E_D\big[E_Y[(\mathrm{Y} - \mathrm{E}[\mathrm{Y} \mid \mathrm{X}])^2 \mid X, D]\big] + E_X E_D[(\mathrm{E}[\mathrm{Y} \mid \mathrm{X}] - \mathrm{f}(\mathrm{X};\mathrm{D}))^2]
\end{aligned}
$$

Since the first term of the error formula does not depend on D, this simplifies to

$$
\begin{aligned}
&E_X E_D E_Y[(Y - f(X;D))^2 \mid X, D] \\
&= E_X\big[E_Y[(\mathrm{Y} - \mathrm{E}[\mathrm{Y} \mid \mathrm{X}])^2 \mid X]\big] + E_X E_D[(\mathrm{E}[\mathrm{Y} \mid \mathrm{X}] - \mathrm{f}(\mathrm{X};\mathrm{D}))^2]
\end{aligned}
$$

Now consider the inner part $E_D[(\mathrm{E}[\mathrm{Y} \mid \mathrm{X}] - \mathrm{f}(\mathrm{X}; \mathrm{D}))^2]$ of the second term. Analogous to the derivation for $E_Y[(Y - f(X; D))^2 \mid X, D]$, we use

$$\mathrm{f}(\mathrm{X}; \mathrm{D}) - \mathrm{E}[\mathrm{Y} \mid \mathrm{X}] = (\mathrm{f}(\mathrm{X}; \mathrm{D}) - E_D[f(X; D)]) + (E_D[f(X; D)] - \mathrm{E}[\mathrm{Y} \mid \mathrm{X}])$$

to show that

$$E_D[(\mathrm{E}[\mathrm{Y} \mid \mathrm{X}] - \mathrm{f}(\mathrm{X}; \mathrm{D}))^2] = E_D[(\mathrm{f}(\mathrm{X}; \mathrm{D}) - E_D[f(X; D)])^2] + (E_D[f(X; D)] - E[Y \mid X])^2$$

Plugging this result into the squared error formula, we obtain

$$\begin{aligned} &E_X E_D E_Y[(Y - f(X; D))^2 \mid X, D] \\ &= E_X[E_Y[(\mathrm{Y} - \mathrm{E}[\mathrm{Y} \mid \mathrm{X}])^2 \mid X] + E_D[(\mathrm{f}(\mathrm{X}; \mathrm{D}) - E_D[f(X; D)])^2] \\ &+ (E_D[f(X; D)] - E[Y \mid X])^2] \\ &\qquad = E_X\big[E_Y[(\mathrm{Y} - \mathrm{E}[\mathrm{Y} \mid \mathrm{X}])^2 \mid X]\big] \\ &\qquad + E_X\big[E_D[(\mathrm{f}(\mathrm{X}; \mathrm{D}) - E_D[f(X; D)])^2]\big] \\ &\qquad + E_X[(E_D[f(X; D)] - E[Y \mid X])^2] \end{aligned}$$

The three terms in the expectation formula are called **irreducible error**, **variance**, and **bias**, respectively. We discuss each in more detail on the next pages.

Irreducible Error

- Note that $E_X\left[E_Y[(\mathrm{Y} - \mathrm{E}[\mathrm{Y} \mid \mathrm{X}])^2 \mid X]\right]$ does not depend on model f or data sample D. This means that no matter what model we choose, this component of the prediction error will remain the same.
- The term therefore measures the inherent noisiness of the data. In particular, if some input $X = x$ is associated with different values of Y, then there cannot be a single "correct" Y for x: no matter the prediction for that x, any model will make an error.
  - Consider a model for predicting income based on GPA; i.e., X is GPA and Y is income. Assume the following about the (unknown) true joint distribution of GPA and income: For GPA=3.8, income is always 90K, but for GPA=3.4, income is 40K or 50K, each with probability 0.5.
  - Then E[income | GPA=3.8] = 90K and hence E_{income}[(income - E[income | GPA=3.8])^2 | GPA=3.8] = E_{income}[(income – 90K)^2 | GPA=3.8]. Since for GPA=3.8 the probability of income=90K is 1.0, this expectation is 1.0(90K-90K)^2 = 0.
  - Now consider GPA 3.4. Here E[income | GPA=3.4] = 0.5(40K+50K) = 45K and hence E_{income}[(income - E[income | GPA=3.4])^2 | GPA=3.4] = E_{income}[(income – 45K)^2 | GPA=3.4]. Since for GPA=3.4 the income is 40K or 50K, each with probability 0.5, this expectation is 0.5(40K-45K)^2+0.5(50K-45K)^2 = (5K)^2.

Model Variance

- Model variance $E_X\left[E_D[(\mathrm{f}(\mathrm{X};\mathrm{D}) - E_D[f(X;D)])^2]\right]$ should not be confused with the notion of variance of a random variable. It measures how much the predictions of an individual model differ from the average prediction of all models trained on the different possible training sets.
  - Note that $f(X = x; D = d)$ is the Y-value returned for input x by a model f that was trained on training data d. Similarly, $E_D[f(X = x; D)]$ represents the expected prediction for x, taken over all models trained on the possible training sets.
- Variance is zero if and only if $\mathrm{f}(\mathrm{X};\mathrm{D}) = E_D[f(X;D)]$ for all inputs x, i.e., the individual model is identical to the model average over the different training sets.
  - Consider model $\mathrm{f}(\mathrm{X};\mathrm{D}) = 1$, which always returns 1. For each input, $\mathrm{f}(\mathrm{X};\mathrm{D}) = E_D[f(X;D)] = 1$.
- Variance is high if and only if $\mathrm{f}(\mathrm{X};\mathrm{D})$ is very different from the model average, i.e., when changing the training data results in large model changes.
  - Consider a distribution where a random 50% of inputs have output 2, the others -2. A 1-nearest-neighbor model predicts for input x the output y' where (x', y') is the training record with the minimal distance from x' to x (ties broken arbitrarily). By construction, there is a 50-50 chance that y' is -2 or 2, therefore $E_D[f(X;D)] = 0.5 \cdot (-2) + 0.5 \cdot 2 = 0$ and variance simplifies to $E_X\left[E_D\left[\left(\mathrm{f}(\mathrm{X};\mathrm{D})\right)^2\right]\right]$. By construction, $\left(\mathrm{f}(\mathrm{X};\mathrm{D})\right)^2 = 4$ and hence $E_D\left[\left(\mathrm{f}(\mathrm{X};\mathrm{D})\right)^2\right] = 4$.
  - We can reduce variance by averaging predictions over the $k > 1$ nearest neighbors in the training data. For $k = 2$, with probability 0.25 both nearest neighbors have output -2, with probability 0.5 one is -2 and the other 2, and with probability 0.25 both are 2. This implies $E_D[f(X;D)] = 0.25 \cdot (-2) + 0.5 \cdot 0 + 0.25 \cdot 2 = 0$ and $E_D\left[\left(\mathrm{f}(\mathrm{X};\mathrm{D})\right)^2\right] = 0.25 \cdot (-2)^2 + 0.5 \cdot 0 + 0.25 \cdot 2^2 = 2$, halving the variance compared to the 1-nearest-neighbor model.

- In summary, model variance is high if model predictions closely track individual Y-values found in a training set. By "smoothing" over many "nearby" samples, model variance can be reduced.

Model Bias

- Model bias $E_X[(E_D[f(X;D)] - E[Y \mid X])^2]$ should not be confused with the notion of bias for estimators. It measures how well f can represent the (X, Y) combinations of the true data distribution.
  - $E[Y \mid X]$ does not depend on the model, but only on the data distribution—it is the expected value of the output for a given input.
  - $E_D[f(X;D)]$ describes the prediction for some input X, averaged over the models trained for the different training sets.
- Bias is zero, if and only if $E_D[f(X;D)] = E[Y \mid X]$ for all X. For this to hold, model f should be flexible enough to represent the relationship between X and Y.
  - Consider true distribution $Y = X$ and assume we train a model that fits a line to the training data. Clearly, this line is $f(X;D) = X$, i.e., it matches the true distribution. Since by construction $E_D[f(X;D)] = X = E[Y \mid X]$, bias for this model is zero.
- Bias is high when the model is not flexible enough to represent the data distribution.
  - Consider the same data, but now a constant model $f(X;D) = c$ for some constant c. Since $E[Y \mid X] = X$ we obtain $(E_D[f(X;D)] - E[Y \mid X])^2 = (X - c)^2$, i.e., quadratically increasing bias with increasing difference between X and c.

Overfitting

- The bias-variance tradeoff is closely related to the problem of overfitting. A model overfits when it represents the training sample too closely, and hence does not generalize well to the (unknown) true distribution the sample was drawn from.
- In other words, a model that overfits has excessively high variance and would improve by lowering that variance.
- In practice, overfitting can be detected by comparing prediction error on the training data to the error on a withheld test dataset that was not used for training.
  - If training error is “significantly” lower than test error, then the model likely overfits: it performs well on the data it was trained on but not other data drawn from the same distribution.

Where is the Tradeoff?

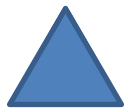
- As discussed above, the simple constant model has zero variance, but potentially high bias because most real-world functions are not flat. On the other hand, a very flexible model like 1-nearest neighbor follows the idiosyncrasies of the given training sample too closely, achieving very low bias at the cost of high variance.
- This showcases a typical behavior of machine learning models: Increasing the "flexibility" of a model allows it to capture more complex real relationships (lower bias), but also makes it more sensitive to changes in the training sample (higher variance). The latter implies that the model is more likely to pick up spurious relationships that hold for the given training sample but not the true distribution.

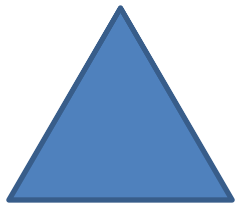
Is There an Optimal Model?

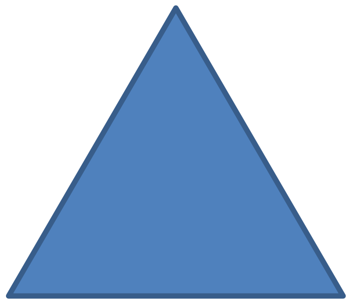
- Somewhat surprisingly, there is! For prediction, the optimal model is $\mathrm{f(X; D)} = E[Y \mid X]$.
  - For variance $E_D[(\mathrm{f(X; D)} - E_D[f(X; D)])^2]$, this results in $E_D[(E[Y \mid X] - E_D[E[Y \mid X]])^2] = E_D[(E[Y \mid X] - E[Y \mid X])^2] = 0$.
  - For bias $(E_D[f(X; D)] - E[Y \mid X])^2$, this results in $(E_D[E[Y \mid X]] - E[Y \mid X])^2 = (E[Y \mid X] - E[Y \mid X])^2 = 0$ as well.
- Unfortunately, learning this model accurately would take a practically infinite amount of training data.
  - Notice that for each input, we need to estimate the expected output. To do so reliably, we need multiple training records for every possible input value. In practice, most inputs are not present in the training data at all; others occur just once.

Bias and Variance for Decision Trees

- Consider a binary decision stump, i.e., a tree with a single binary split node. It partitions the training data into two (large) subsets. The average of Y over a large sample is very stable (replacing a few sample points will not affect that average). This implies very low variance. On the other hand, the stump represents a simple 2-step function that cannot model complex interactions between attributes and hence has high bias.
- The other extreme is a deep tree where each training record appears in a different leaf. Even a small modification of the training data directly affects the predictions in the corresponding leaves, hence this tree has high variance. Bias is low because the large tree can model complex decision boundaries, perfectly separating the different classes from each other.
- In general, a larger number of split nodes results in higher variance and lower bias.

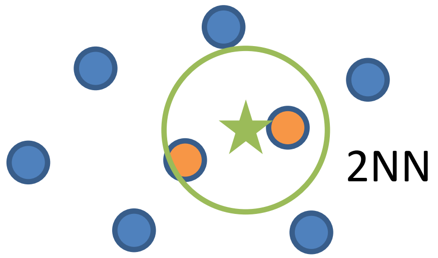

Low variance, high bias


Best tree: good balance of bias and variance

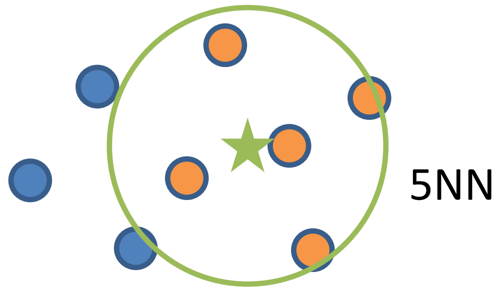

High variance, low bias

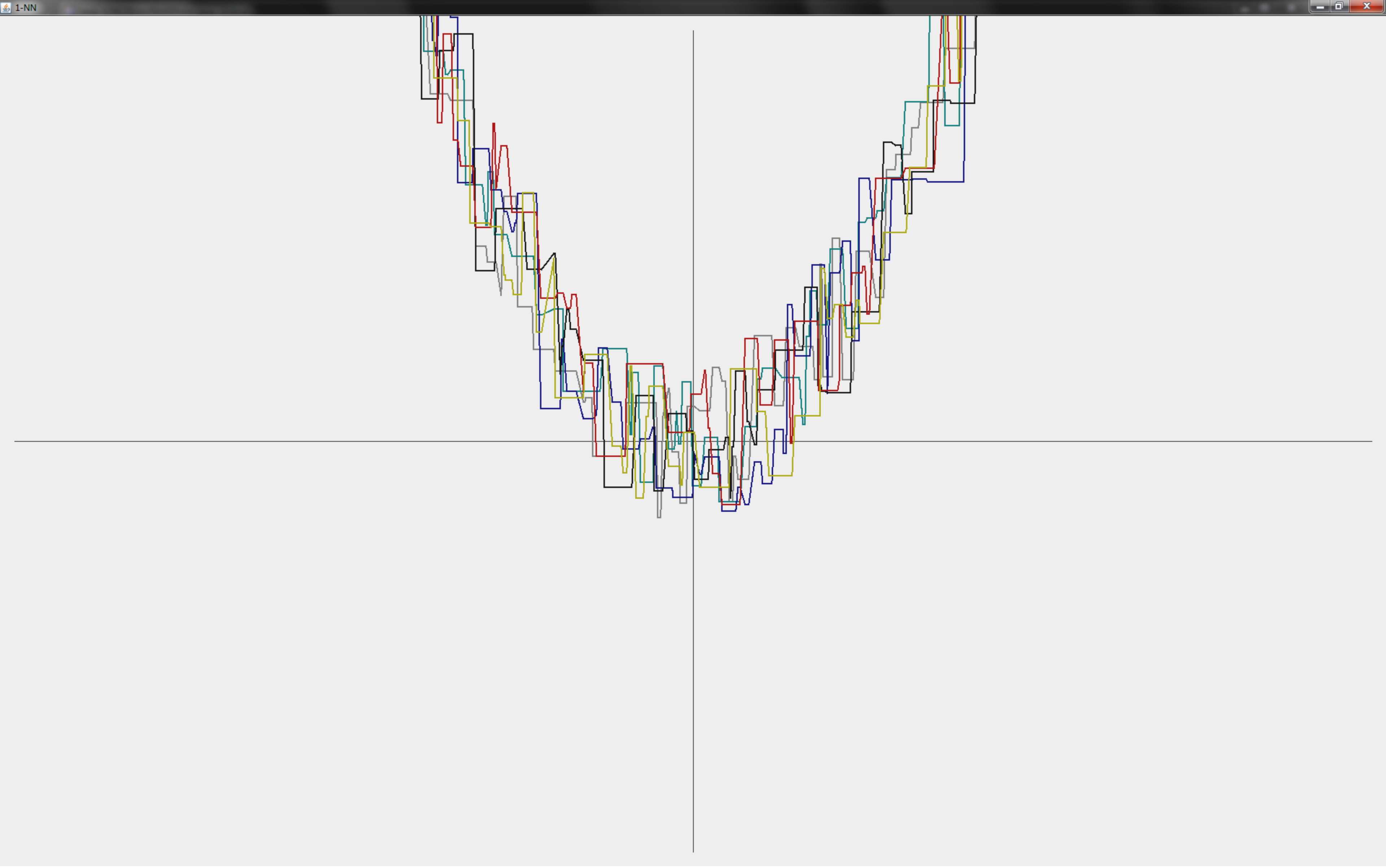
Bias-Variance Tradeoff in Action

- The following experiment for the K-nearest neighbor (KNN) prediction technique shows the bias-variance tradeoff in practice. KNN predicts output Y for a given input X=x by returning the average Y value over the K data points *closest* to x in the training data.
  - Larger K averages over a larger neighborhood. This decreases variance (as variations in training data are averaged out) but increases bias (as local trends are smoothed over).
- Consider quadratic function $\mathrm{f(X)} = X^2$. Given a training set that approximately reflects this function, the goal is to train a KNN model that learns the function as accurately as possible.
- Each training set consists of 50 pairs (x,y) generated as follows: Choose values for x uniformly at random from range $-2 \leq x \leq 2$. For each x, generate the corresponding y as $y = x^2 + \varepsilon$, where ε is the noise, selected uniformly at random from range [-0.5, 0.5].
- Bias and variance are explored experimentally for KNN with K=1, K=20, and K=100.

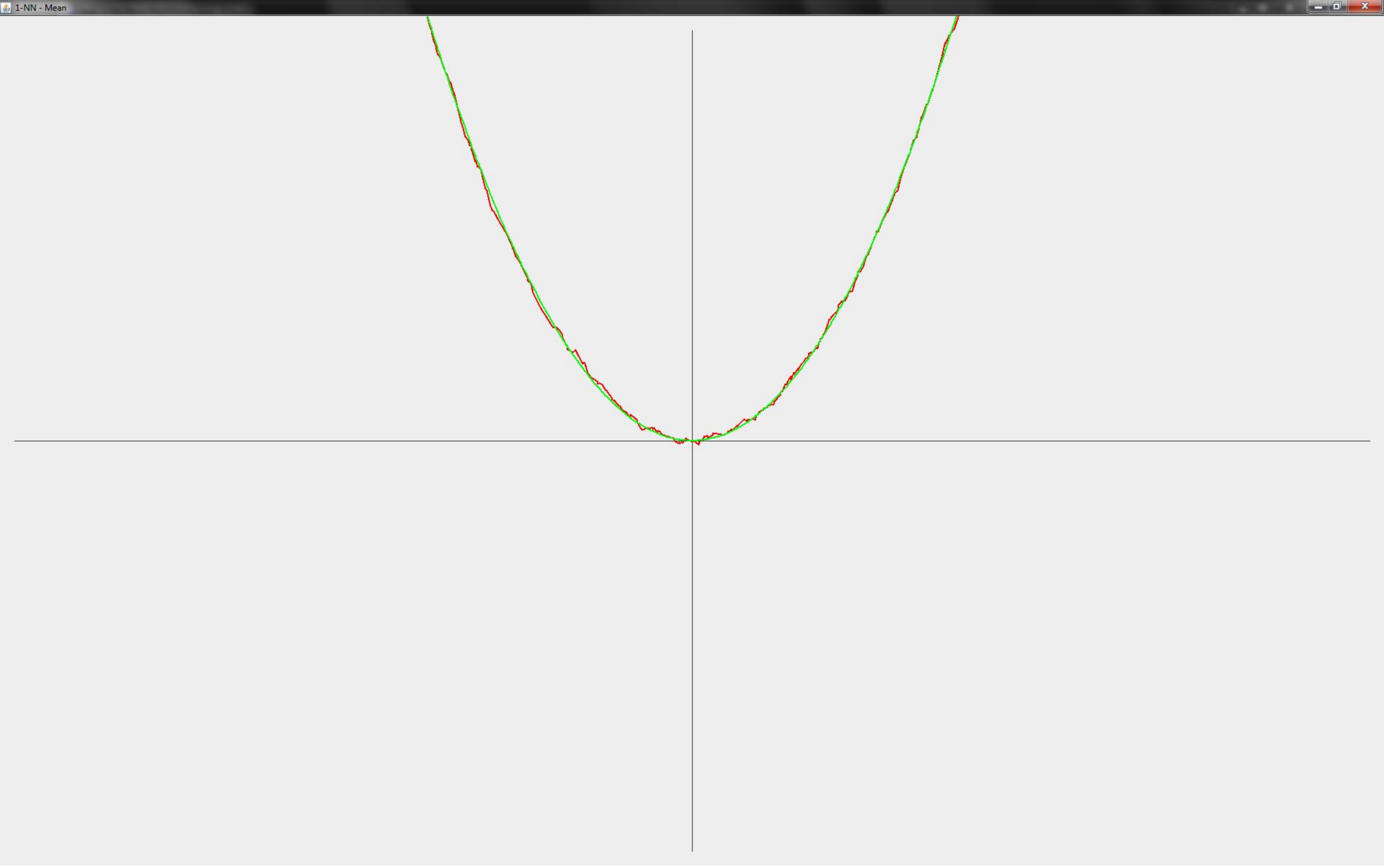


★ Test point
● Training point
● One of the K closest training points

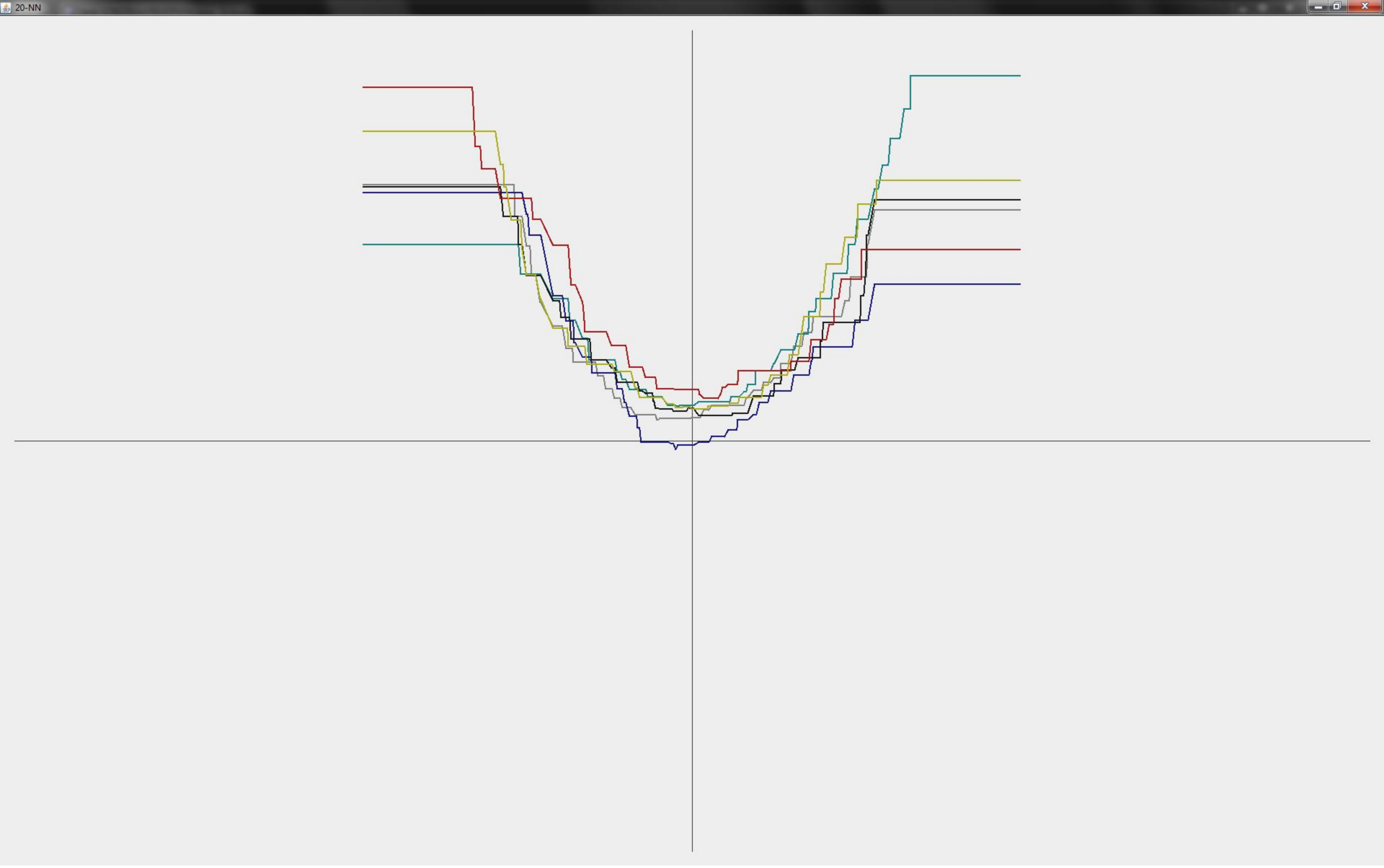




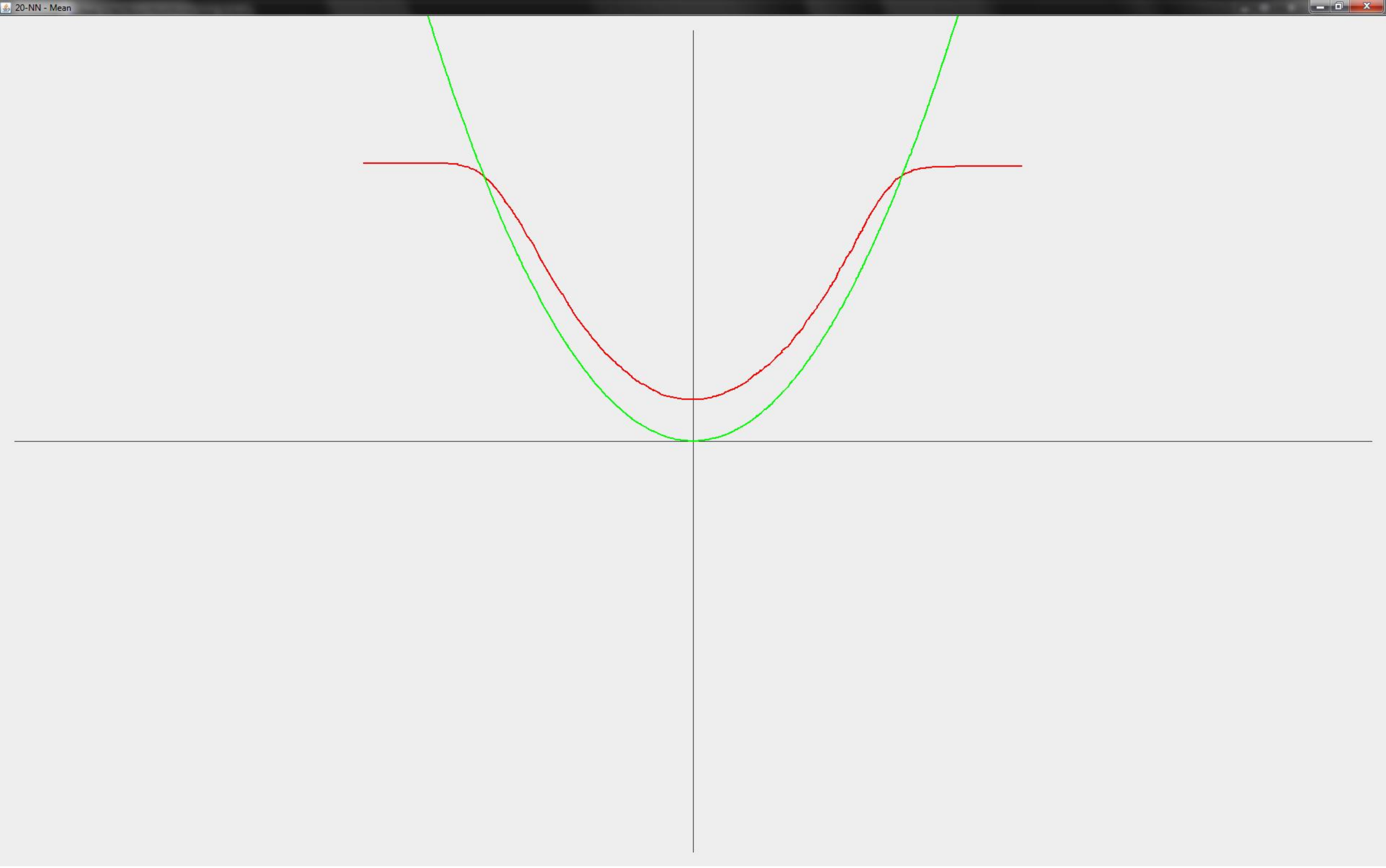
Predictions made by five different KNN models, each trained on a different data sample, for K=1. Notice that the models reflect the noise in the training data and hence differ significantly from each other. This reflects their high variance caused by considering only the single nearest neighbor.



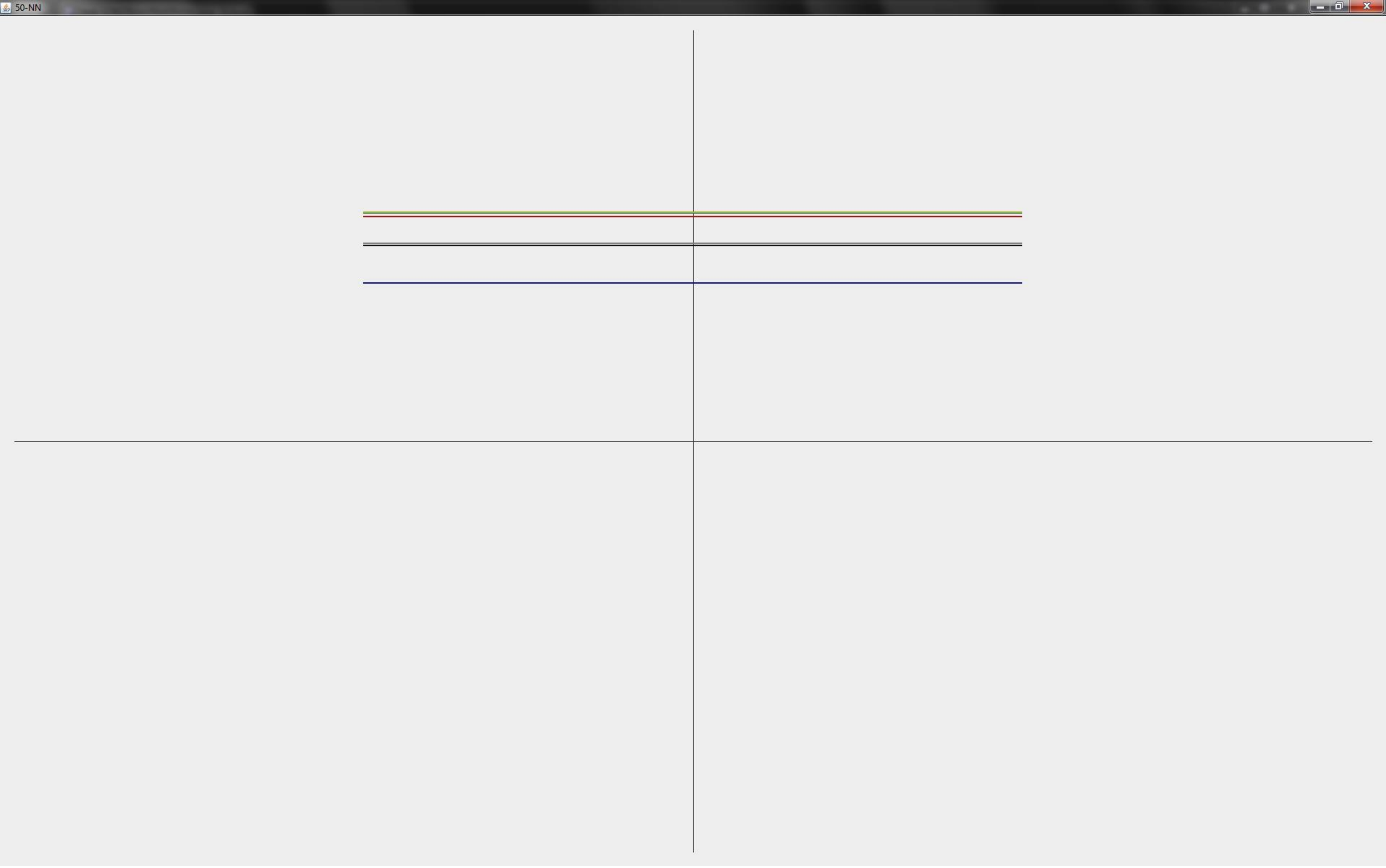
Average prediction over 200 different KNN models , each trained on a different data sample, for K=1 (red line) compared to the correct function $Y=X^2$ (green line). This plot shows the bias of the 1NN model. As expected, since 1NN makes predictions based on very small local neighborhoods, it can closely model any training data, resulting in very low bias.



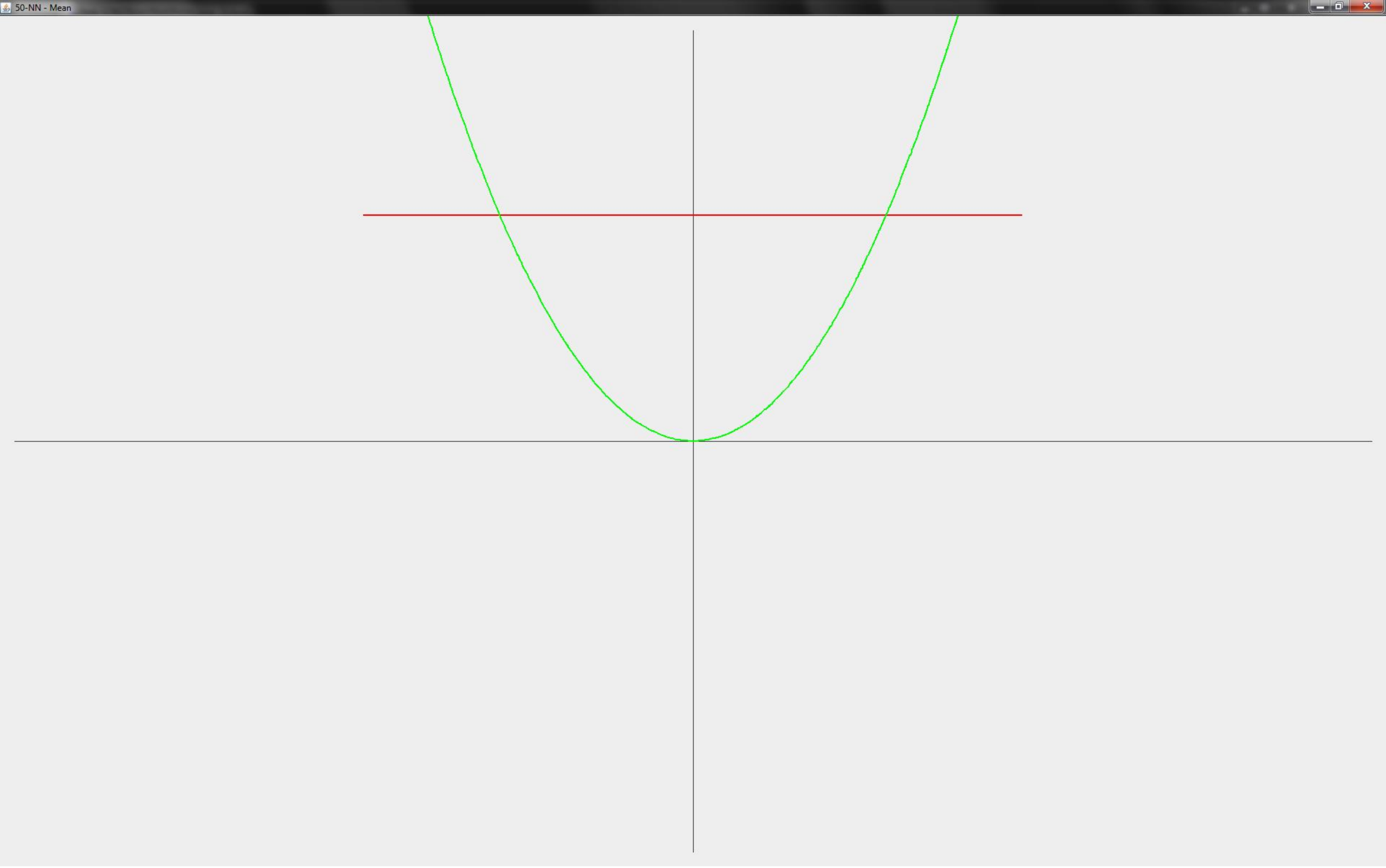
Predictions made by five different KNN models , each trained on a different data sample, for K=20. It is clearly visible that the individual models are less “noisy” than for 1NN, because 20NN averages over larger neighborhoods. (The horizontal lines at the left and right are caused by boundary effects as points near the extremes have most of their neighbors on one side, instead of evenly distributed on their left and right.)



Average prediction over 200 different KNN models , each trained on a different data sample, for K=20 (red line) compared to the correct function Y=X^2 (green line). This plot shows the bias of the 20NN model. Not surprisingly, by averaging over larger neighborhoods than 1NN, 20NN cannot capture local behavior, in particular at the center and the extremes of the range.



Predictions made by five different KNN models , each trained on a different data sample, for K=50. Since there are only 50 training records, each prediction is the average over all those 50 points, resulting in a constant function. The different functions are more similar to each other than for 20NN and 1NN, showing the lower variance due to the averaging over larger neighborhoods.



Average prediction over 200 different KNN models , each trained on a different data sample, for K=50 (red line) compared to the correct function $Y=X^2$ (green line). This plot shows the high bias of the 50NN model, which has little in common with the actual quadratic function. Not surprisingly, by averaging over the entire domain, 50NN cannot capture local behavior for different X values at all.

Let us return to bagging and take a closer look at how it works in practice.

Bagging Reminder

- Bagging stands for bootstrap aggregation.
- Given a training data set D, a bagged ensemble is trained as follows:
  - Create e independent bootstrap samples of D.
  - Train e individual models, each separately on a different sample.
- The bagged model computes the output for a given input X=x as follows:
  - Compute $M_i(x)$ for each of the e models $M_1,\ldots, M_j$.
  - Return the average of these individual predictions.

What is a Bootstrap Sample?

- Consider a training dataset with n records.
- Each bootstrap sample D_i also contains n records. These records are sampled from D uniformly at random, using sampling with replacement.
- This implies that some records from D might be sampled more than once, while others are not sampled at all.
  - Each training record has probability $1 - (1 - 1/n)^n$ of being selected at least once in a sample of size n. For large n, this expression converges to $1 - 1/e = 0.63$.

1	2	3	4	5

Training data D

2	4	1	2	2

5	1	3	1	3

3	4	4	5	2

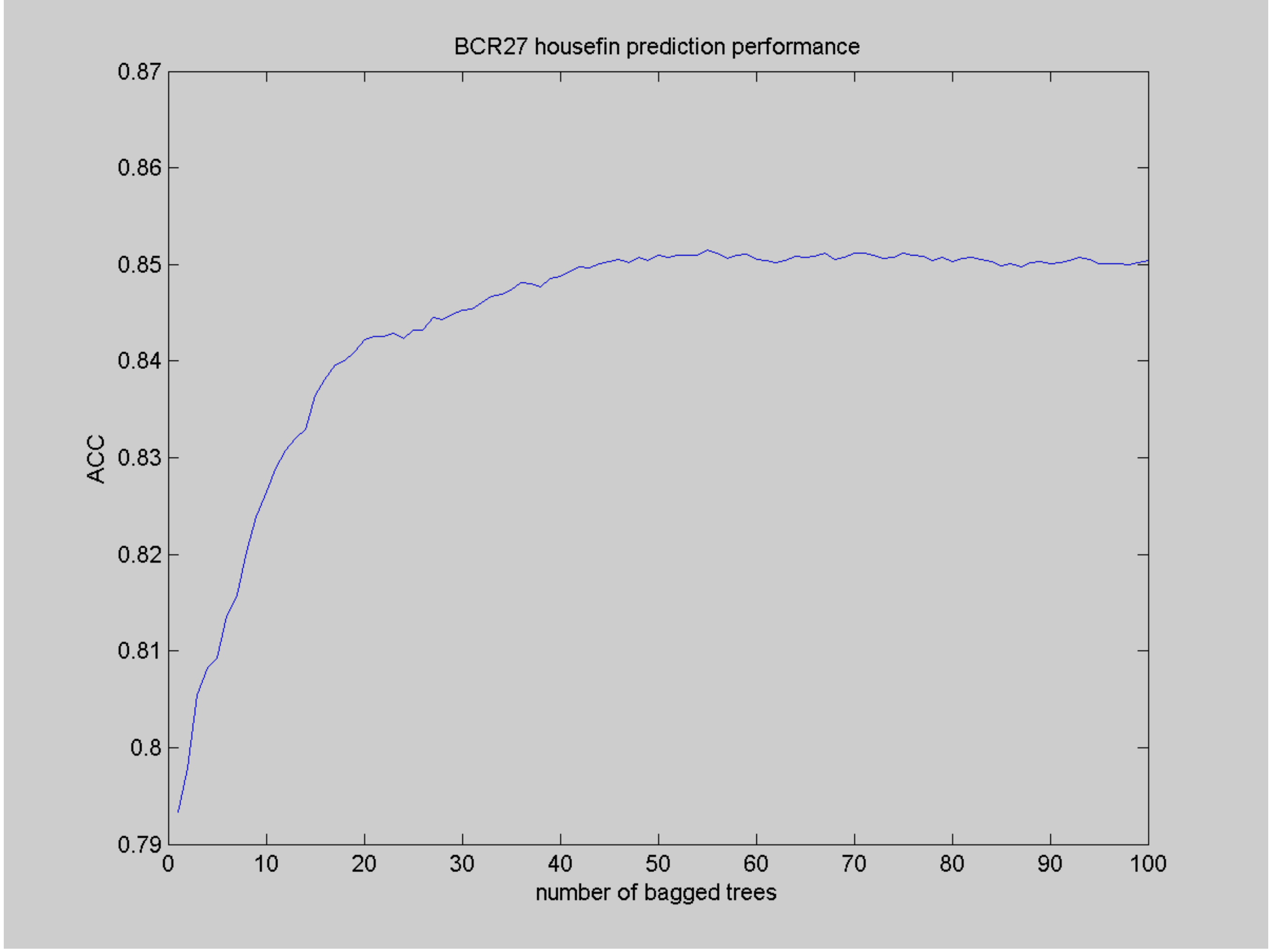
Typical bootstrap samples of D

Bagging Challenges

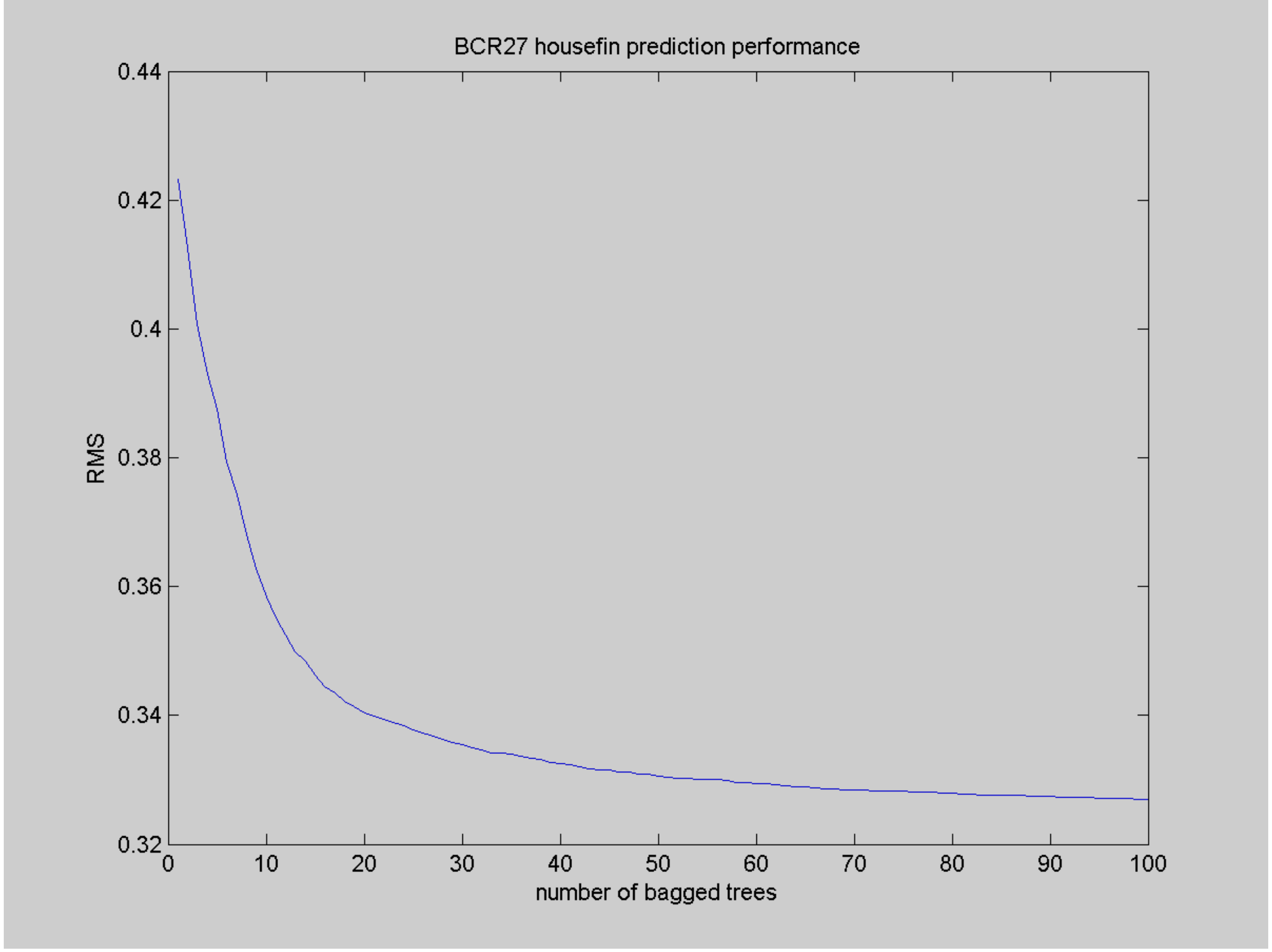
- The individual models in a bagged ensemble should be independent of each other. Only then can variance be effectively reduced through model averaging.
- Independence can be achieved through training on independent data samples, but in practice we usually settle for less because (1) only a limited amount of labeled data is available and (2) for each model, the training set needs to be representative of the overall data distribution.
  - Small training data jeopardizes prediction quality, which is dangerous for ensemble models. Recall that each model must be more than 50% accurate for the ensemble to improve over an individual model.
- Bootstrap sampling represents a practical solution to obtain many training sets that are reasonably independent and large. In contrast, simply partitioning D into j subsets would create more independence, but the n/j records per partition may not sufficiently represent the data distribution.
- Independence can be increased by diversifying models. For example, Random Forest improves tree diversity compared to plain bagged trees by limiting the choice of split attributes to a random subset of the available attributes, each subset independently chosen for each node. Or one can include different model types in the same ensemble, e.g., trees, SVMs, and regression models.

Typical Bagging Results

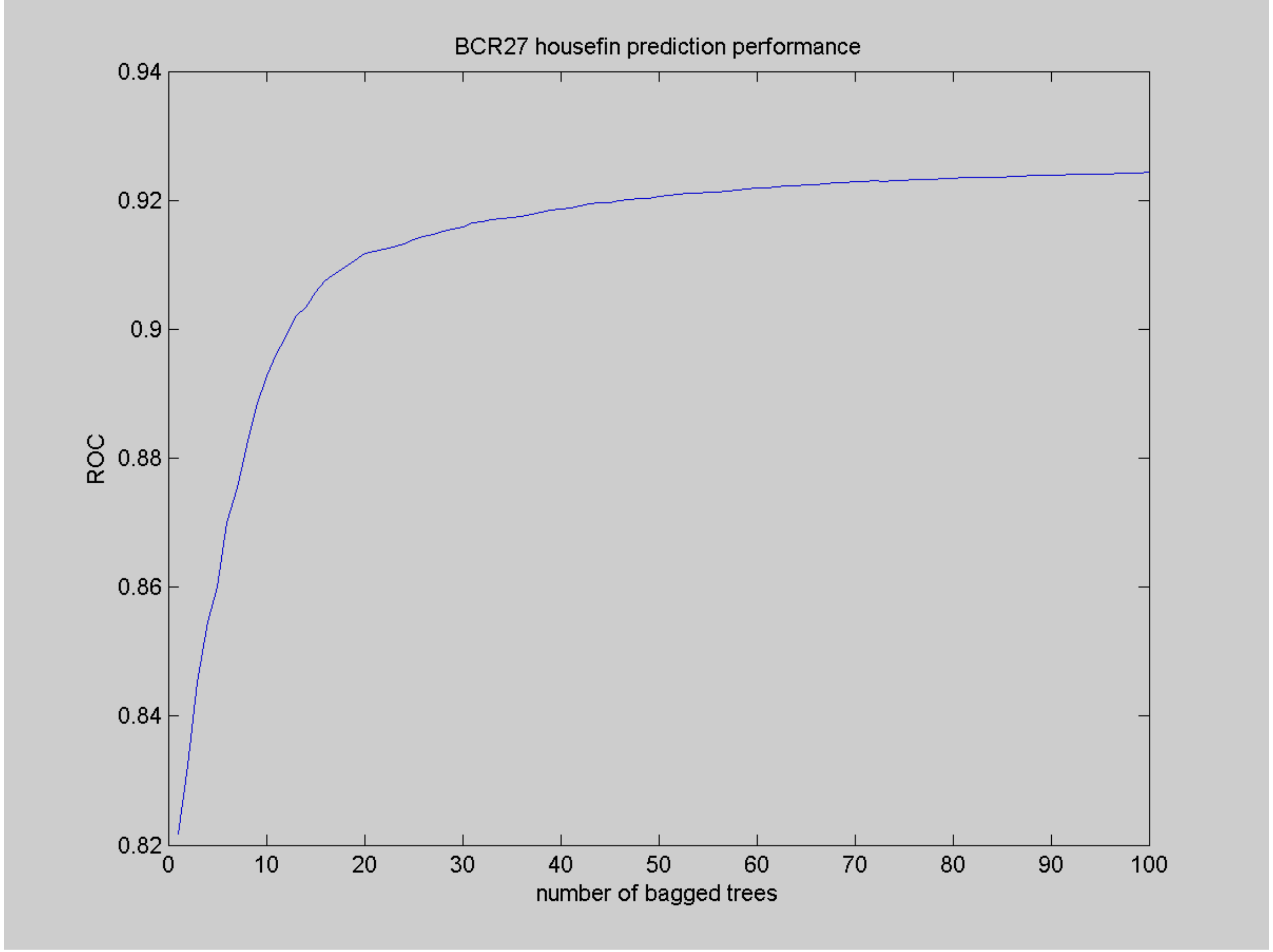
- For bagging to improve significantly over individual models, the ensemble might need dozens or even hundreds of individual models. Since bagging reduces variance without affecting bias, each individual model should overfit, having low bias even at the cost of high variance.
  - For decision trees, choose trees with more split nodes than for the best individual model. For KNN, choose a smaller K.
- Due to overfitting of individual models, small bagging ensembles tend to have mediocre prediction quality. As more models are added, variance is “averaged away” and prediction quality typically improves until it hits a ceiling. If it does not, then either individual models overfit too much to a degree where they are less than 50% accurate; or they are not sufficiently independent of each other.
- The next pages show real experimental results that illustrate the typical behavior of bagged ensembles.



Typical bagging behavior on a real-world problem with bird-observation data. The graph shows how ensemble accuracy (higher is better) improves as more tree models are added. At about 50 trees the ensemble hits a ceiling.



Typical bagging behavior on a real-world problem with bird-observation data. The graph shows how the root mean squared error (lower is better) of the ensemble improves as more tree models are added. Even at 100 trees, ensemble error is still improving, suggesting that more trees should be added.



Typical bagging behavior on a real-world problem with bird-observation data. The graph shows how the area under the ROC curve (higher is better) of the ensemble improves as more tree models are added. Even at 100 trees, ensemble ROC area is still improving, suggesting that more trees should be added.

Bagging in MapReduce

- It is easy to parallelize bagging. For model training, each bootstrap sample and corresponding individual model can be created independently. Similarly, for predictions each model can be evaluated independently, followed by a simple computation of the average across models.
- Existing libraries for local (in-memory) model training can be leveraged by having each individual model trained completely inside a task.

Parallel Training

- Assume a machine-learning library is available for in-memory training on a single machine. Each of the j models in the ensemble can be trained in a different task. This task only needs a bootstrap sample and model parameters to control the training process.
  - Model parameters can be passed to all Reducers using the file cache. Each line in the file states the model identifier and corresponding parameters.
- Mappers create j copies of each data record and send them to j different Reduce calls. Each Reduce call creates its own bootstrap sample and then trains the model.
  - If training data exceeds memory size, Map can randomly sample to reduce data size, setting p < 1.0. (This also improves model independence.)

```
map( training record r )
 for i = 1 to j do
  emit(i, r) with probability p
```

<u>Model-parameter file:</u>
ID, list of parameters
1, parameters for model 1
2, parameters for model 2
3, parameters for model 3
…

```
class Reducer {

 setup () { array params = load from file cache}

 reduce(i, [r1, r2,…])
  R = load [r1, r2,…] into memory
  B = MLlibrary.createBootstrapSample( R )
  M = MLlibrary.trainModel( B, params[i] )
  emit( i, M )          // Or write to HDFS/S3 file
 }
}
```

Alternative Parallel Training

- The previous MapReduce program transfers p·j input copies from Mappers to Reducers. The Map-only program below avoids this transfer.
  - The entire training set is copied to all worker machines using the file cache.
  - Mappers locally create bootstrap samples and train a model for each sample.
- This program transfers as many input copies from DFS as there are machines executing Map tasks. But how does each Mapper know how many models to create and which parameters to use for them?
  - We again use the model parameter file, but this time make it the input of the Map-only job, letting each Map call train the corresponding model.
  - Since the input file is small and each line creates a large amount of work for sampling and model training, the default setting of 1 Map task per file split would be too coarse-grained, possibly resulting in a single Map task for the job. The NLineInputFormat class can be used to create smaller input splits.

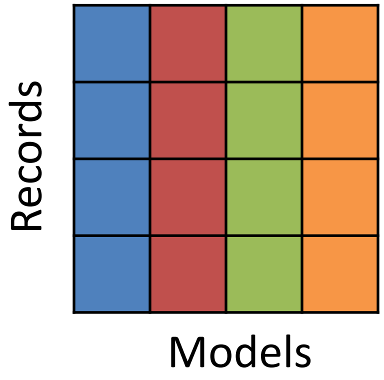
```
map( model number i, model_parameters ) {
 read the training data from file cache, creating a sample S that fits in memory, using sampling rate p

 B = MLlibrary.bootstrap(S)
 M = MLlibrary.trainModel( B, model_parameters )
 emit(i, M)                // Or write to HDFS/S3 file
}
```

Making Predictions in Parallel

- Each individual model in the ensemble needs to compute its predictions for each test record. Abstractly, this corresponds to the Cartesian product between the set of models and the set of test records.
  - This is followed by a simple aggregation, computing the average prediction (or majority vote) for each test record.
- What is the best way to partition this computation over multiple tasks? We will discuss three options and compare their properties.

Prediction: Vertical Stripes

- To implement vertical partitioning in MapReduce, the test-record file is copied to all Mapper machines using the file cache. The file containing the models is the input of the job.
- Map reads a model and computes its prediction for every test record. Reduce computes the average prediction per test record.
- Data transfer (without combining and excluding final output):
  - HDFS to Map: #mapperMachines * |test data file| + 1 * |model file|
  - Map to Reduce: #models * |test data file|
    - Combining is effective if a Mapper receives multiple models.

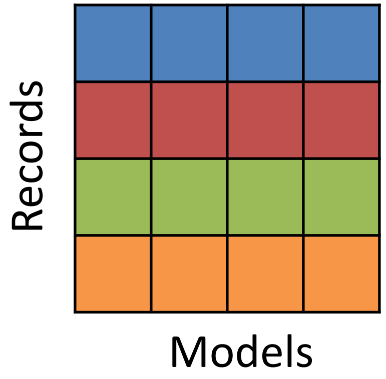


```
Class Mapper {
  T = read all test records from file cache

  map( model M ) {
    for each t in T do
      emit( t, M(t) )
  }
}
```

```
reduce( t, [$M_1$(t), $M_2$(t),…] ) {
  for each M(t) in input list
    update running sum and count

  emit( t, sum/count)
}
```

Prediction: Horizontal Stripes

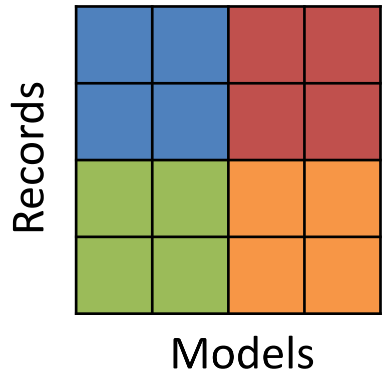
- To implement horizontal partitioning in MapReduce, the model file is copied to all Mapper machines using the file cache. The file containing the test records is the input of the job.
- Since each Map task has the entire bagged model, it can compute the average prediction locally, eliminating the need for a Reduce phase. Map reads a test record and computes its prediction for every model, keeping track of running sum and count to emit the average in the end.
- Data transfer (excluding final output):
  - HDFS to Map: 1 * |test data file| + #mapperMachines * |model file|



```
Class Mapper {
 Models = read all models from file cache

 map( test record t ) {
  for each M in Models do
   compute M(t) and update running sum and count

  emit( t, sum/count )
 }
}
```

Prediction: Blocks

- To implement partitioning into A-by-B blocks in MapReduce, both test data and model file must be appropriately partitioned and duplicated. The algorithm is identical to 1-Bucket-Random, which we discuss in another module. Note that a post-processing job is needed to aggregate predictions across the different blocks (not shown below).
- Data transfer (without in-Reducer combining and excluding final output):
  - HDFS to Map: 1 * |test data file| + 1 * |model file|
  - Map to Reduce: B * |test data file| + A * |model file|
  - Reduce to HDFS: B * 2 * #testRecords
    - Each test record is assigned to a "row" of B blocks, producing a (sum, count) pair per block. In-Reducer combining could lower this when multiple blocks are processed in the same Reducer.
  - Post-processing: read from HDFS B * 2 * #testRecords records with prediction sum and count for each test record



MapReduce Program for Block Partitioning

```
map(…, object x) {
 if (x is a test record) {
  // Select a random integer from range [0,…, A-1]
  row = random( 0, A-1 )

  // Emit the record for all regions in the selected “row”.
  for key = (row * B) to (row * B + B – 1)
   emit( key, (x, “S”) )
 }
 else {    // x is a model
  // Select a random integer from range [0,…, B-1]
  col = random( 0, B-1 )

  // Emit the model for all regions in the selected
  // “column”. This requires skipping B region numbers
  // forward from start region key equal to col.
  for key = col to ((A-1)*B + col) step B
   emit( key, (x, “T”) )
 }
}
```

```
reduce( regionID, [(x1, flag1), (x2,
flag2),…]) {
 initialize S_list and T_list

 // Separate the input list by the data set
 // the tuples came from
 for all (x, flag) in input list do
  if (flag = “S”)
   S_list.add( x )
  else
   T_list.add( x )

 for each test record t in S_list {
  for each model M in T_list
   compute M(t) and update running
                       sum and count
  emit( t, sum/count )
 }
}
```

Ensembles in Spark

- The Spark MLlib machine-learning library as of August 2021 offered two types of ensembles: Random Forest (variation of bagging) and Gradient-Boosted Trees (GBT). More will likely be added over time.
- Challenge question: Find out how distributed training and prediction is implemented for these two tree-based methods.

Random Forest in MLlib With DataSet (from Spark 2.3.2 Documentation)

```
import org.apache.spark.ml.Pipeline
import org.apache.spark.ml.classification.{RandomForestClassificationModel, RandomForestClassifier}
import org.apache.spark.ml.evaluation.MulticlassClassificationEvaluator
import org.apache.spark.ml.feature.{IndexToString, StringIndexer, VectorIndexer}

// Load and parse the data file, converting it to a DataFrame.
val data = spark.read.format("libsvm").load("data/mllib/sample_libsvm_data.txt")

// Index labels, adding metadata to the label column. Fit on whole dataset to include all labels in index.
val labelIndexer = new StringIndexer().setInputCol("label").setOutputCol("indexedLabel").fit(data)
// Automatically identify categorical features and index them. Set maxCategories so features with > 4 distinct values are treated as continuous.
val featureIndexer = new VectorIndexer().setInputCol("features").setOutputCol("indexedFeatures").setMaxCategories(4).fit(data)

// Split the data into training and test sets (30% held out for testing).
val Array(trainingData, testData) = data.randomSplit(Array(0.7, 0.3))

// Train a RandomForest model.
val rf = new RandomForestClassifier().setLabelCol("indexedLabel").setFeaturesCol("indexedFeatures").setNumTrees(10)

// Convert indexed labels back to original labels.
val labelConverter = new IndexToString().setInputCol("prediction").setOutputCol("predictedLabel").setLabels(labelIndexer.labels)

// Chain indexers and forest in a Pipeline.
val pipeline = new Pipeline().setStages(Array(labelIndexer, featureIndexer, rf, labelConverter))

// Train model. This also runs the indexers.
val model = pipeline.fit(trainingData)
```

Random Forest in MLlib With DataSet (from Spark 2.3.2 Documentation, cont.)

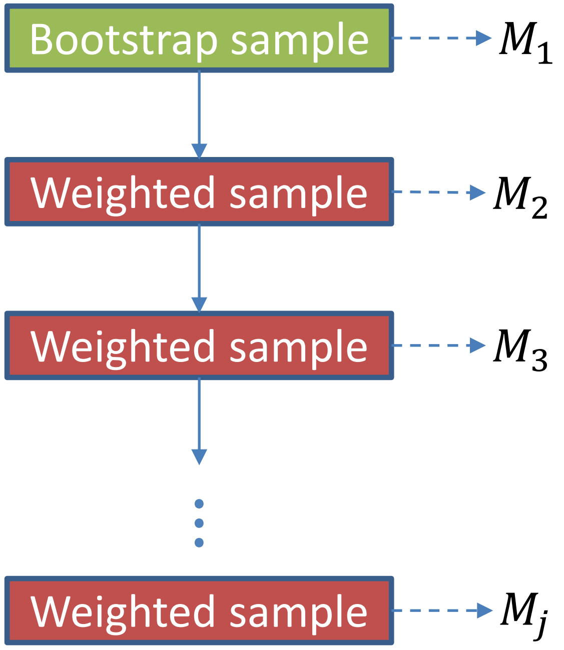
```
// Make predictions.
val predictions = model.transform(testData)

// Select example rows to display.
predictions.select("predictedLabel", "label", "features").show(5)

// Select (prediction, true label) and compute test error.
val evaluator = new MulticlassClassificationEvaluator()
  .setLabelCol("indexedLabel")
  .setPredictionCol("prediction")
  .setMetricName("accuracy")
val accuracy = evaluator.evaluate(predictions)
println(s"Test Error = ${(1.0 - accuracy)}")

val rfModel = model.stages(2).asInstanceOf[RandomForestClassificationModel]
println(s"Learned classification forest model:\n ${rfModel.toDebugString}")
```

Another popular ensemble method is boosting. How does it differ from bagging and how can it be parallelized?

Boosting Overview

- There are different boosting variants. We discuss the classic AdaBoost (Adaptive Boosting) approach.
- Like bagging, it creates an ensemble of M models, but each individual model has a weight according to its prediction accuracy: higher accuracy means higher weight. The ensemble prediction is the weighted sum of the individual model predictions.
  - Intuitively, instead of treating all models equally, AdaBoost “listens” more to the more accurate models.
- The other difference to bagging is that AdaBoost trains models iteratively, one after the other. And while it uses sampling with replacement to create the training data for an individual model, training records have weights that control how likely they are to be sampled.
  - Initially all training records have the same weight. After each iteration, AdaBoost increases the weights of records that were misclassified by the individual model trained in the last iteration, decreasing the weight of those that were correctly classified.
  - This way more and more copies of a misclassified training record will appear in the training sample, forcing the model to “pay more attention to it to get it right.”

Mathematical View



Here we consider a classification model that can distinguish between any number of classes, not just two. Given some input x, each individual model M_i votes its weight α_i for some class. The ensemble tallies the total per class and returns the class that received the highest score. (This is weighted majority voting.)

Note that δ is an indicator function returning 1 if M_i returns prediction y, and zero otherwise.

Ensemble prediction for input x:

$$M(x) = \arg\max_y \left[\sum_{i=1}^{j} \alpha_i \cdot \delta(M_i(x) = y) \right]$$

Weighted Sampling Intuition

- Records that were incorrectly classified will have their weights increased. Records that were classified correctly will have their weights decreased.
- Assume record 3 is hard to classify. Its weight keeps increasing; therefore it is more likely to be chosen again in subsequent rounds.

Given data	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Sample in iteration 1	5	9	1	7	**3**	5	2	5	1
Sample in iteration 2	8	**3**	2	1	**3**	9	**3**	2	8
Sample in iteration 3	**3**	4	**3**	7	**3**	**3**	1	4	**3**

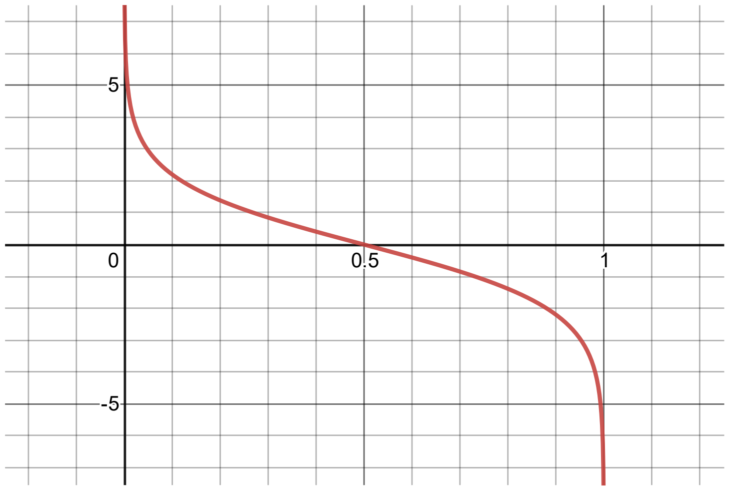
AdaBoost Details

- We are given n training records $(x_l, y_l)_{1 \le l \le n}$, which AdaBoost samples from to train individual models $M_1, \dots, M_j$. Record l has weight w_l, such that $\sum_l w_l = 1$.
- The error rate of model M_i is

$$\varepsilon_i = \sum_{l=1}^{n} w_l \cdot \delta(M_i(x_l) \neq y_l)$$

- δ is an indicator function that returns 1 if model M_i misclassifies record (x_l, y_l); 0 otherwise.
- The weight of model M_i (i.e., its importance in the ensemble) is

$$\alpha_i = \ln \frac{1 - \varepsilon_i}{\varepsilon_i}$$



α_i as a function of ε_i. Note how a perfect model ($\varepsilon_i = 0$) results in infinite weight, while error rate above 0.5 results in negative weight. Both should never happen. The former is caused by overfitting and requires use of a model type with lower variance. The latter triggers a reset of all training-record weights and a re-training of this individual model.

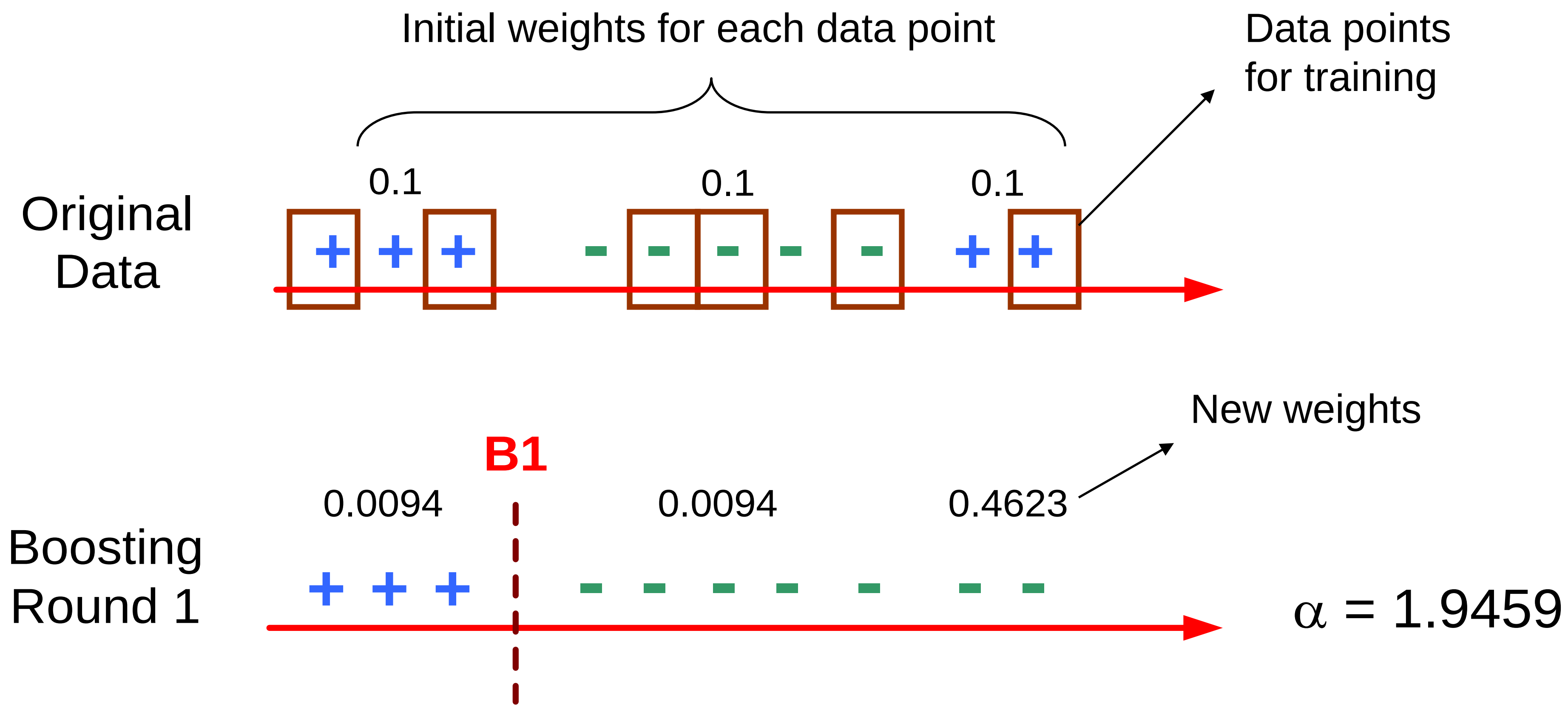
AdaBoost Details (cont.)

- After training individual model M_i, the weights of all training records are updated as

$$w_l^{(i+1)} = \frac{w_l^{(i)}}{Z_i} \cdot \begin{cases} \dfrac{\varepsilon_i}{1-\varepsilon_i} & \text{if } M_i(x_l) = y_l \\ 1 & \text{if } M_i(x_l) \neq y_l \end{cases}$$

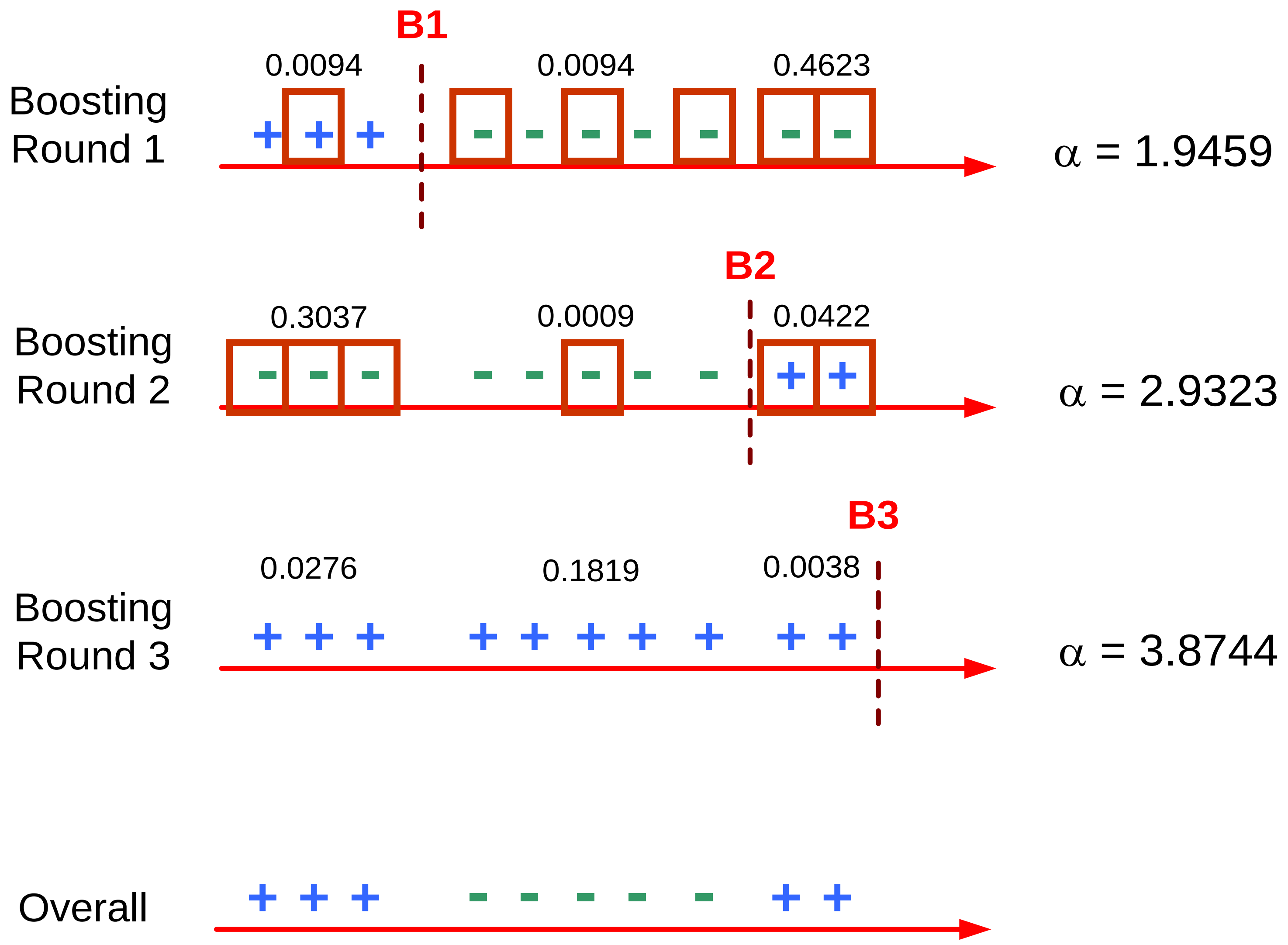
- Z_i is a normalization factor to ensure that all new weights for iteration $i+1$ add up to 1.
- If an iteration's error rate exceeds 0.5, then all weights are reverted to 1/n and that iteration is repeated.

Illustrating AdaBoost



Example based on slides for textbook [Introduction to Data Mining by Tan, Steinbach, and Kumar. Pearson, 1st edition]. Note that the numbers appear incorrect, but they convey the flavor of the technique.

Illustrating AdaBoost



Note: These numbers also appear incorrect.

Bagging vs. Boosting

- Analogy
  - Bagging: diagnosis = multiple doctors’ simple-majority vote
  - Boosting: weighted vote, based on each doctor’s previous diagnosis accuracy
- Sampling procedure
  - Bagging: records have the same weight; easy to train in parallel
  - Boosting: weights a training record higher if a model predicts it wrong; inherently sequential process
- Overfitting
  - Bagging is robust against overfitting
  - Boosting is susceptible to overfitting: make sure individual models do not overfit
- Accuracy is usually significantly better than a single classifier. And the best boosted model is often better than the best bagged model.

Boosting in Spark

- Boosting differs from bagging in a crucial way: models are trained one-at-a-time.
  - This is caused by the property that the prediction errors made by the i-th model affect the training of the (i+1)-st model.
- Hence the only opportunity for parallelism lies in the training of an individual model.
- Since Spark already offers parallel training of individual tree models, Gradient-Boosted Trees in MLlib rely on the parallel training of individual trees for boosting.

Boosting in MLlib With DataSet (from Spark 2.3.2 Documentation)

```
import org.apache.spark.ml.Pipeline
import org.apache.spark.ml.classification.{GBTClassificationModel, GBTClassifier}
import org.apache.spark.ml.evaluation.MulticlassClassificationEvaluator
import org.apache.spark.ml.feature.{IndexToString, StringIndexer, VectorIndexer}

// Load and parse the data file, converting it to a DataFrame.
val data = spark.read.format("libsvm").load("data/mllib/sample_libsvm_data.txt")

// Index labels, adding metadata to the label column. Fit on whole dataset to include all labels in index.
val labelIndexer = new StringIndexer().setInputCol("label").setOutputCol("indexedLabel").fit(data)
// Automatically identify categorical features and index them. Set maxCategories so features with > 4 distinct values are treated as continuous.
val featureIndexer = new VectorIndexer().setInputCol("features").setOutputCol("indexedFeatures").setMaxCategories(4).fit(data)

// Split the data into training and test sets (30% held out for testing).
val Array(trainingData, testData) = data.randomSplit(Array(0.7, 0.3))

// Train a GBT model.
val gbt = new GBTClassifier().setLabelCol("indexedLabel").setFeaturesCol("indexedFeatures").setMaxIter(10).setFeatureSubsetStrategy("auto")

// Convert indexed labels back to original labels.
val labelConverter = new IndexToString().setInputCol("prediction").setOutputCol("predictedLabel").setLabels(labelIndexer.labels)

// Chain indexers and GBT in a Pipeline.
val pipeline = new Pipeline().setStages(Array(labelIndexer, featureIndexer, gbt, labelConverter))

// Train model. This also runs the indexers.
val model = pipeline.fit(trainingData)
```

Boosting in MLlib With DataSet (from Spark 2.3.2 Documentation, cont.)

```
// Make predictions.
val predictions = model.transform(testData)

// Select example rows to display.
predictions.select("predictedLabel", "label", "features").show(5)

// Select (prediction, true label) and compute test error.
val evaluator = new MulticlassClassificationEvaluator()
  .setLabelCol("indexedLabel")
  .setPredictionCol("prediction")
  .setMetricName("accuracy")
val accuracy = evaluator.evaluate(predictions)
println(s"Test Error = ${1.0 - accuracy}")

val gbtModel = model.stages(2).asInstanceOf[GBTClassificationModel]
println(s"Learned classification GBT model:\n ${gbtModel.toDebugString}")
```

Summary

- Ensemble methods achieve high prediction accuracy and are good candidates for distributed computation due to their high cost. This applies particularly to bagging, because its models can be trained and queried independently.
- Boosting trains the ensemble one-model-at-a time, hence parallelism only comes from parallel training of individual models.
- Ensemble prediction follows a cross-product computation pattern with per-model aggregation. Depending on the implementation choice, this could be done without shuffling, a single shuffle phase, or might even require two shuffles (for block partitioning, due to the additional aggregation job).

References

- Data mining textbook: Jiawei Han, Micheline Kamber, and Jian Pei. Data Mining: Concepts and Techniques, 3rd edition, Morgan Kaufmann, 2011
- Biswanath Panda and Joshua S. Herbach and Sugato Basu and Roberto J. Bayardo. PLANET: Massively Parallel Learning of Tree Ensembles with MapReduce. Proc. Int. Conf. on Very Large Data Bases (VLDB), 2009
  - https://scholar.google.com/scholar?cluster=11753975382054642310&hl=en&as_sdt=0,22