Data Mining: Ensembles

Mirek Riedewald

This work is licensed under the Creative Commons Attribution 4.0 International License.
To view a copy of this license, visit http://creativecommons.org/licenses/by/4.0/.

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 with a bagged ensemble.

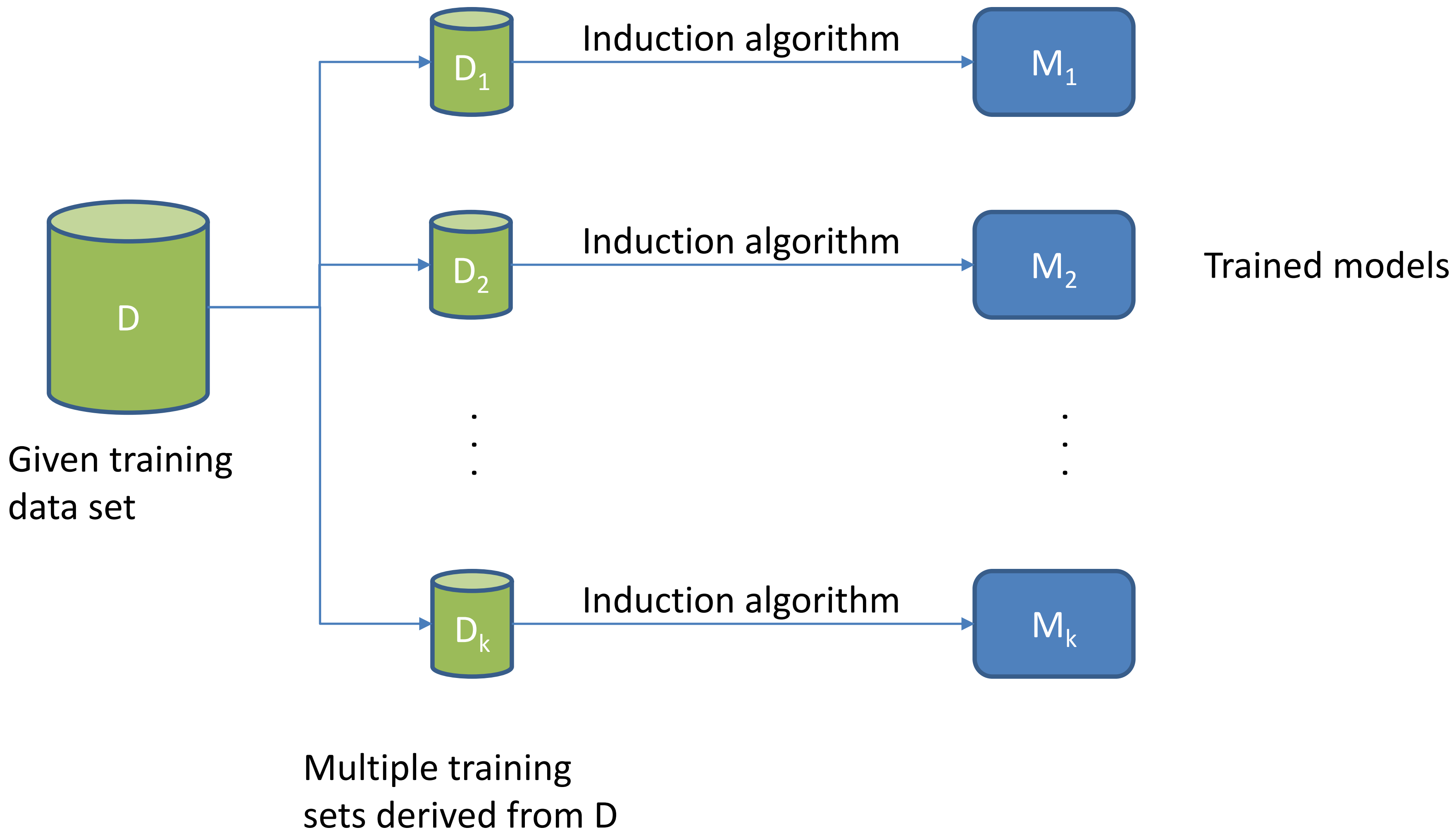
Introduction

- 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.

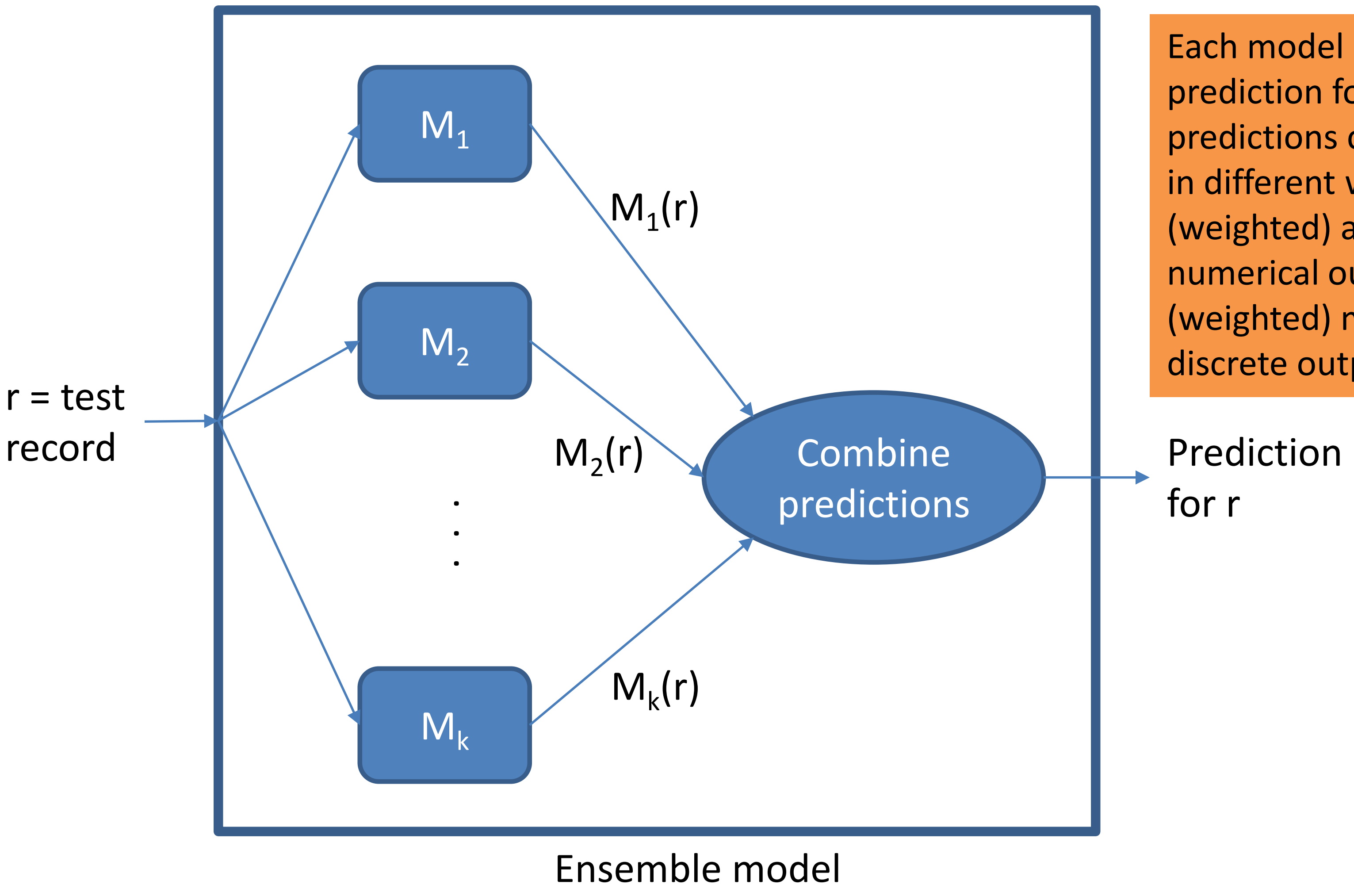
Ensemble Methods

- 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 r, the individual predictions of all models in the ensemble are aggregated appropriately.
- The next two pages illustrate this idea.

Training an Ensemble



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 have to 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's.
  - **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 only provides anecdotal evidence why ensembles improve prediction quality. Fortunately, this property can also be shown in general. Here we focus on bagging.
- 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 will usually 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, and Bayesian networks.

Mathematical Derivation of the Bias-Variance Tradeoff

- The bias-variance tradeoff can be shown mathematically, using statistical decision theory.
- Consider two real-valued random variables X and Y, where X denotes the input attributes and Y the output attribute.
- Given a training set D of (X, Y) combinations, we want to construct a function f(X), the prediction model, that returns good approximations of Y for future inputs X. To make the dependence of function f on data set D explicit, we write f(X; D).
- The goal is to find the best model f. A common measure of the quality of a prediction model is mean squared error. Formally, the goal is to minimize $E_{X,D,Y}[\,(Y - f(X; D))^2\,]$, the expected squared error over all X, Y, and D.

Why Does The Analysis Consider the Training Dataset D?

- Data set D is a sample from an unknown data distribution. The goal of classification and prediction is to learn this data distribution as accurately as possible. Some models might work very well for one sample, but poorly for another. Since we do not know which type of sample the given D represents, the analysis needs to take them all into account by treating D as another random variable.
- We discuss next how to transform mean squared error $E_{X,D,Y}[\ (Y - f(X; D))^2\ ]$ into a format that clearly shows bias and variance.

We use a standard property of expectation to obtain

$$E_{X,Y,D}[(Y - f(X;D))^2] = E_X E_D \boldsymbol{E_Y}\left[(\mathbf{Y} - \mathbf{f}(\mathbf{X};\mathbf{D}))^2 \mid \mathbf{X}, \boldsymbol{D}\right]$$

Now consider inner term $\boldsymbol{E_Y}\left[(\mathbf{Y} - \mathbf{f}(\mathbf{X};\mathbf{D}))^2 \mid \mathbf{X}, \boldsymbol{D}\right]$.

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

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

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

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

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:

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

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

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

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

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

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 $E_Y[(\mathrm{Y} - \mathrm{E}[\mathrm{Y} \mid \mathrm{X}])^2 \mid X]$ does not depend on model f, nor on data sample D. This means that no matter what prediction model we choose, this component of the prediction error will remain the same.
- The term measures the inherent noisiness of the data. In particular, if the same input X=x is associated with different values of Y, then there cannot be a single "correct" Y for that input x—no matter the prediction for that x, any model will have to make an error.
  - For example, consider a model f to predict income based on GPA; i.e., X is GPA and Y is income. Assume the following about the (usually 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 $E_D[(\mathrm{f}(\mathrm{X};\mathrm{D}) - E_D[f(X;D)])^2]$ should not be confused with the notion of variance of a random variable. It measures how much predictions for a model trained on a specific random data sample differs from the average prediction of all models trained over different random data samples.
  - Note that f(X=x; D=d) is the Y-value returned by a model f that was trained on data sample d, for input x.
  - Similarly, E_D[f(X=x; D)] represents the average prediction for x, taken over all models trained on all possible random data samples d.
- Variance is zero if and only if $\mathrm{f}(\mathrm{X};\mathrm{D}) = E_D[f(X;D)]$ for all X. Intuitively, this happens when an individual model trained on a single data sample D=d predicts the same as an ensemble of models trained from all possible data samples D, and those predictions are averaged.
  - When would this happen in practice? Consider a simple model such as $\mathrm{f}(\mathrm{X};\mathrm{D}) = 1$, which always returns the value 1. Clearly, for all possible inputs X=x, $\mathrm{f}(\mathrm{X};\mathrm{D}) = E_D[f(X;D)] = 1$.
- When is variance high? This happens when $\mathrm{f}(\mathrm{X};\mathrm{D})$ returns values that very closely track the (X, Y) pairs found in a single sample D=d.
  - Consider a region X=x of the X-space that contains Y-values 0 and 1, each with probability 0.5. Further consider a family of models that simply predicts the average Y for the records sampled from that region. Assume sample D contains a single record from that region—with probability 0.5, this record has Y=0, otherwise Y=1—therefore $E_D[f(X=x;D)] = 0.5 \cdot 0 + 0.5 \cdot 1 = 0.5$. Hence $E_D[(\mathrm{f}(\mathrm{X};\mathrm{D}) - 0.5)^2] = 0.5(0-0.5)^2 + 0.5(1-0.5)^2 = 0.25$.
  - Model variance would decrease, if larger samples were considered. It would increase, if the Y-values in the region come from a larger range, e.g., between 0 and 10.
- In summary, model variance is high, if function values are closely tracking the Y-values found in small subsets of data records in a given data sample.

Model Bias

- Model bias $(E_D[f(X;D)] - E[Y \mid X])^2$ should not be confused with the notion of bias for estimators. It measures how closely f can model the (X,Y) combinations occurring in any possible training data set D.
  - $E[Y \mid X]$ does not depend on the model, but only on the data distribution—it is the expected value of output Y for a given input X.
  - Term $E_D[f(X;D)]$ describes the prediction for some input X, averaged over the models trained for all possible data samples D.
- 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 has to be flexible enough to represent the relationship between X and Y.
  - Let us return to the example of the region X=x containing Y=0 and Y=1 with probability 0.5 each, and the family of models that predicts the average Y for the records sampled from that region. Clearly, $E[Y \mid X = x] = 0.5$. Assume as before that sample d contains a single record from that region, which with probability 0.5 has Y=0, otherwise Y=1. This implies $E_D[f(X;D)] = 0.5$ as before, resulting in bias of zero.
- When is bias high? This happens when the model is not flexible enough to represent the data distribution.
  - Consider again the constant model $\mathrm{f(X;D)} = 1$, which always returns the value 1. For the above distribution with $E[Y \mid X = x] = 0.5$, the bias of this model is $(1\text{-}0.5)^2$=0.25 for X=x.

Overfitting

- In the context of machine learning, the notion of overfitting plays an important role. 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.
- From the viewpoint of the bias-variance tradeoff, 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.

Where is the Tradeoff?

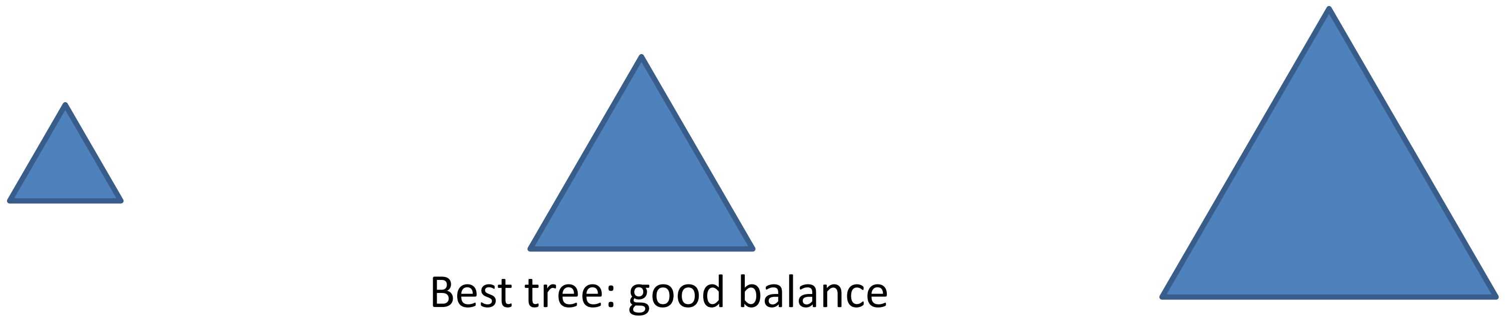
- The above examples for bias and variance revealed interesting relationships:
  - When estimating Y for some X=x based on a single sample record, variance is high, but bias is low.
  - When using the constant model, variance is low, but bias is high.
- This showcases a typical behavior of machine learning models: reducing bias typically leads to higher variance, and vice versa.
  - As we make the model more flexible to capture complex relationships present in the given data sample D=d to reduce bias, the model also picks up spurious relationships that hold for d, but would not be present in many other data samples D.

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 X=x, we need to estimate the expected output Y. To do so reliably, we need multiple training records for every possible input X=x. In practice, most inputs are not present in the training data at all; others occur just once.

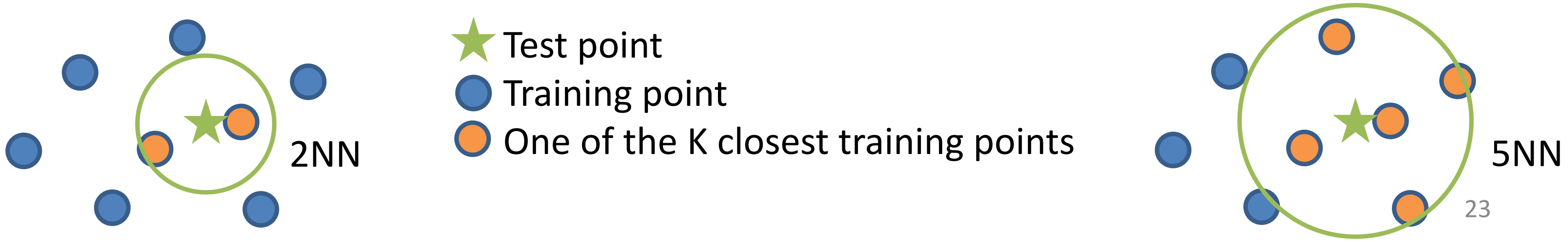
Bias and Variance for Decision Trees

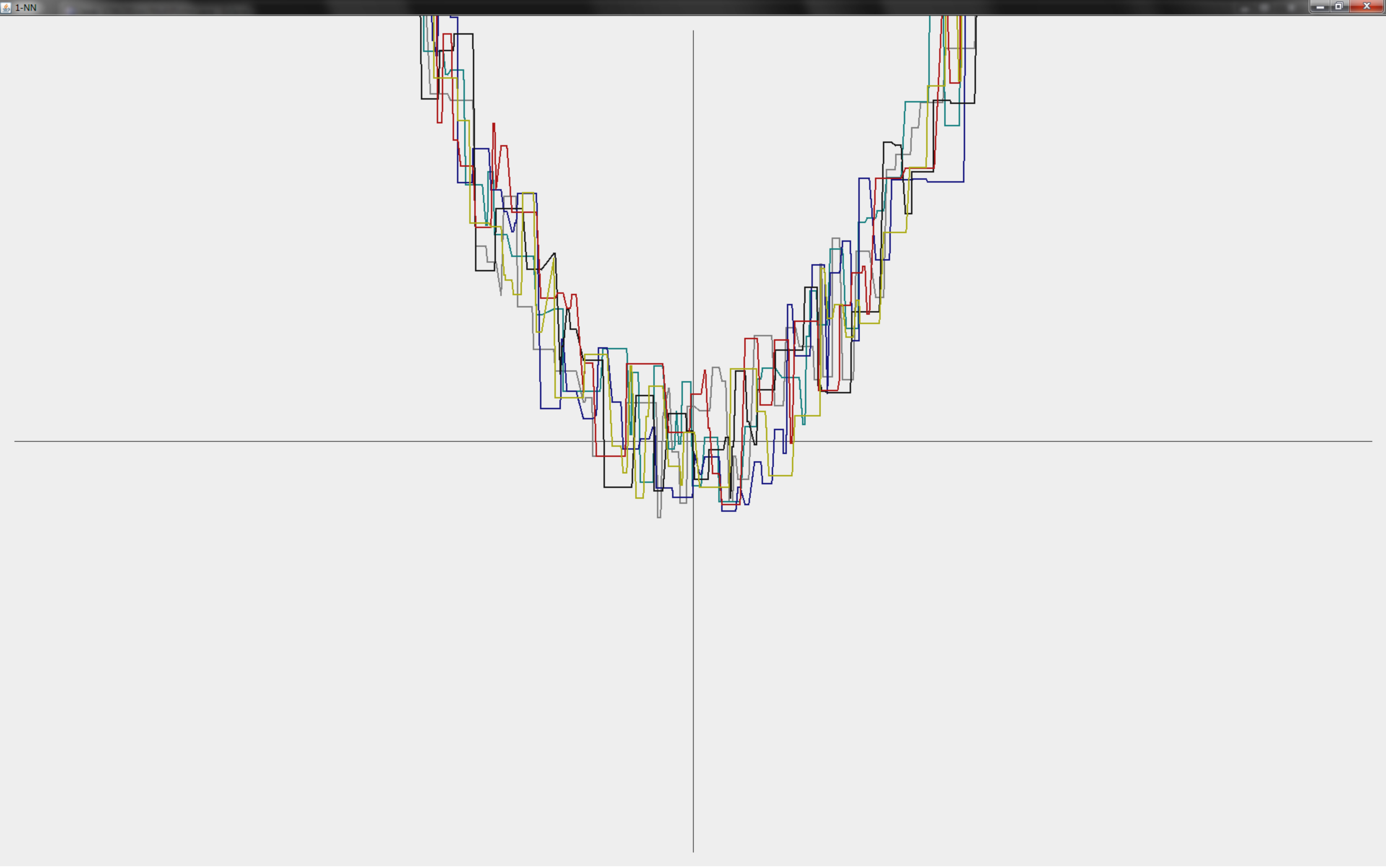
- Consider a binary decision stump, i.e., a tree with a single binary split node. It partitions the training set into only two (large) subsets. Hence the average Y for each subset will be close to the true expectation of Y for the corresponding region of the data space.
  - This implies very low variance. On the other hand, the stump cannot model complex interactions between attributes and hence has high bias.
- On the other extreme, consider a large deep tree where each training record is in a different leaf. Even a small modification of the training data directly affects the predictions in the corresponding leaves. (This corresponds to the 1-record sample example discussed earlier).
  - This tree has a high variance. On the other hand, bias is low because the large tree can model very complex boundaries, perfectly separating the different classes from each other.
- In general, for trees a larger number of split nodes results in higher variance and lower bias.



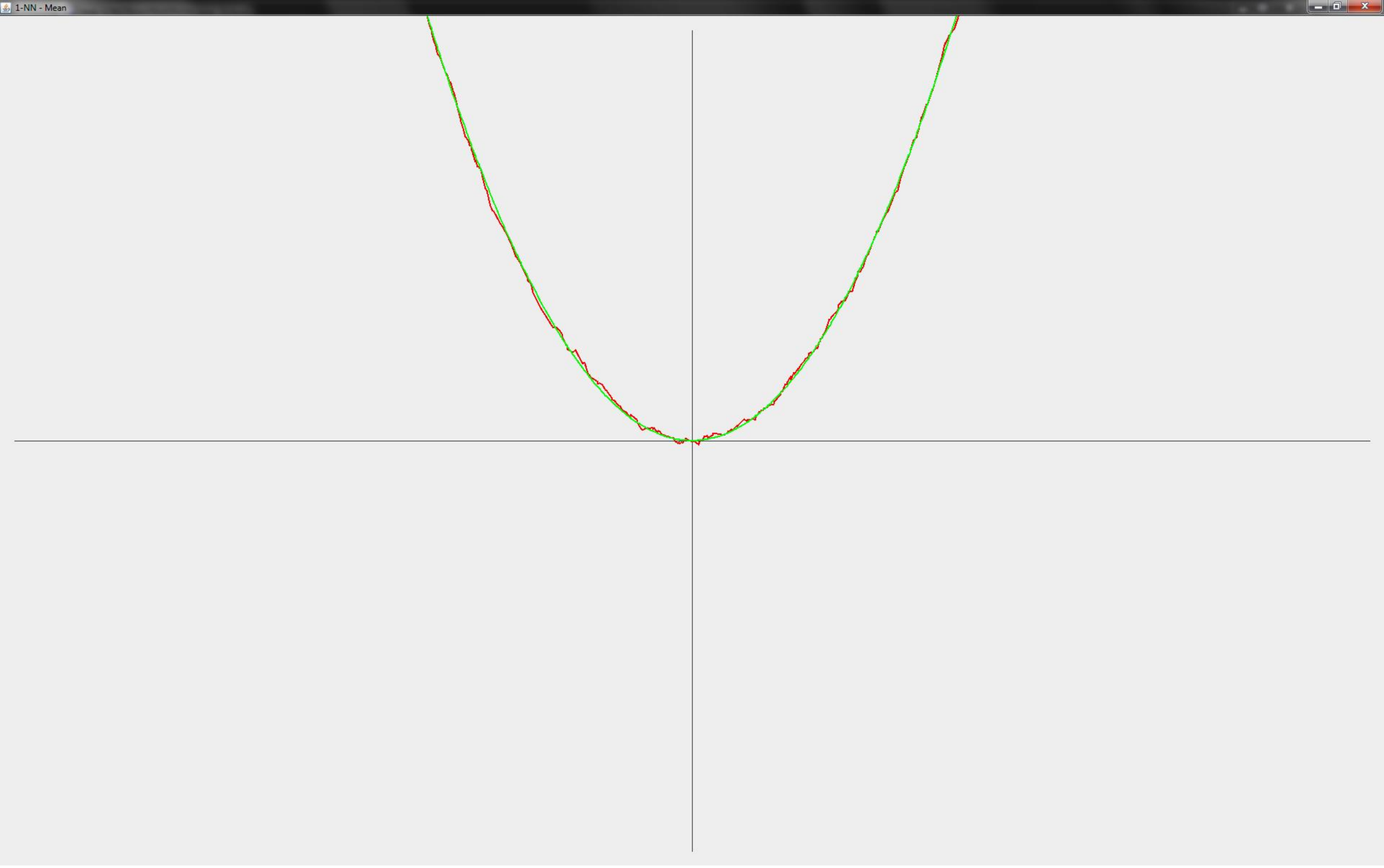
Bias-Variance Tradeoff in Action

- 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 intuitively decreases variance (as variations in training data are averaged out), but increases bias (as local trends are smoothed out).
- Consider quadratic function $\mathrm{f(X)} = X^2$. Given a training set D that approximately reflects this function, the goal is to train a KNN model that learns the function as accurately as possible.
- Each training set D 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.

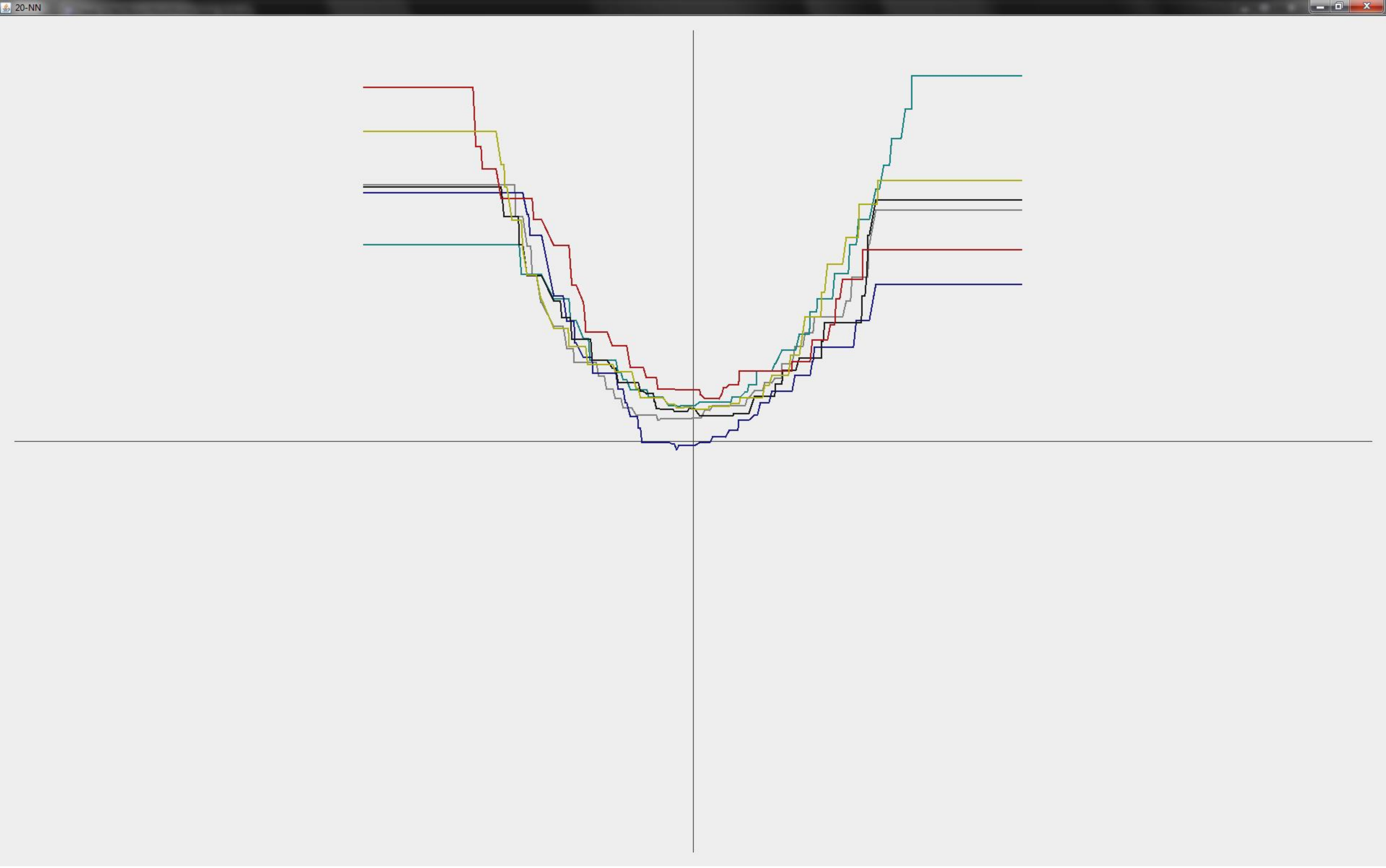




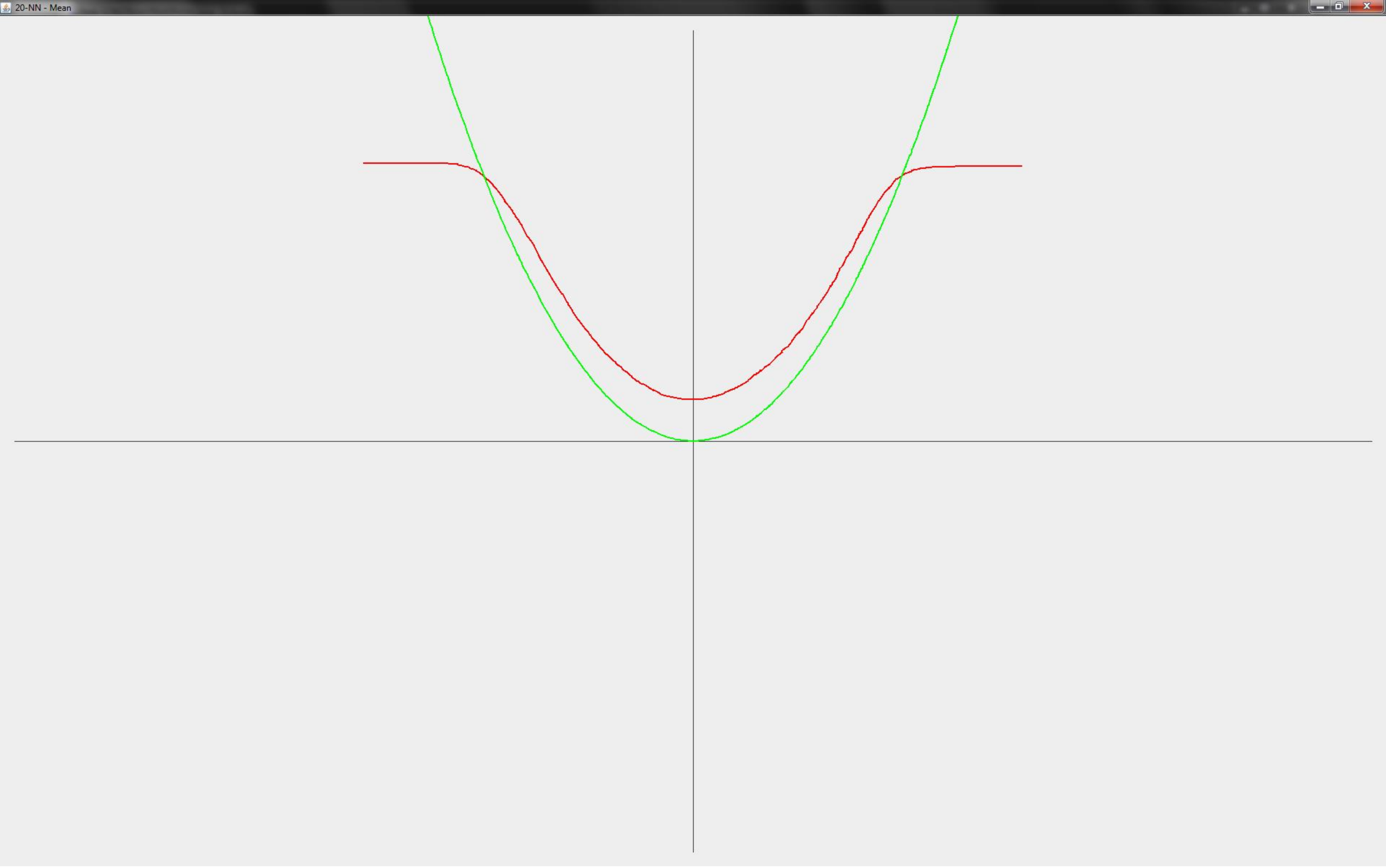
Predictions made by five different KNN models, each trained on a different data sample, for K=1. Notice that the models reflect the noise in the training data and hence differ significantly from each other. This reflects their high variance caused by considering only the single nearest neighbor.



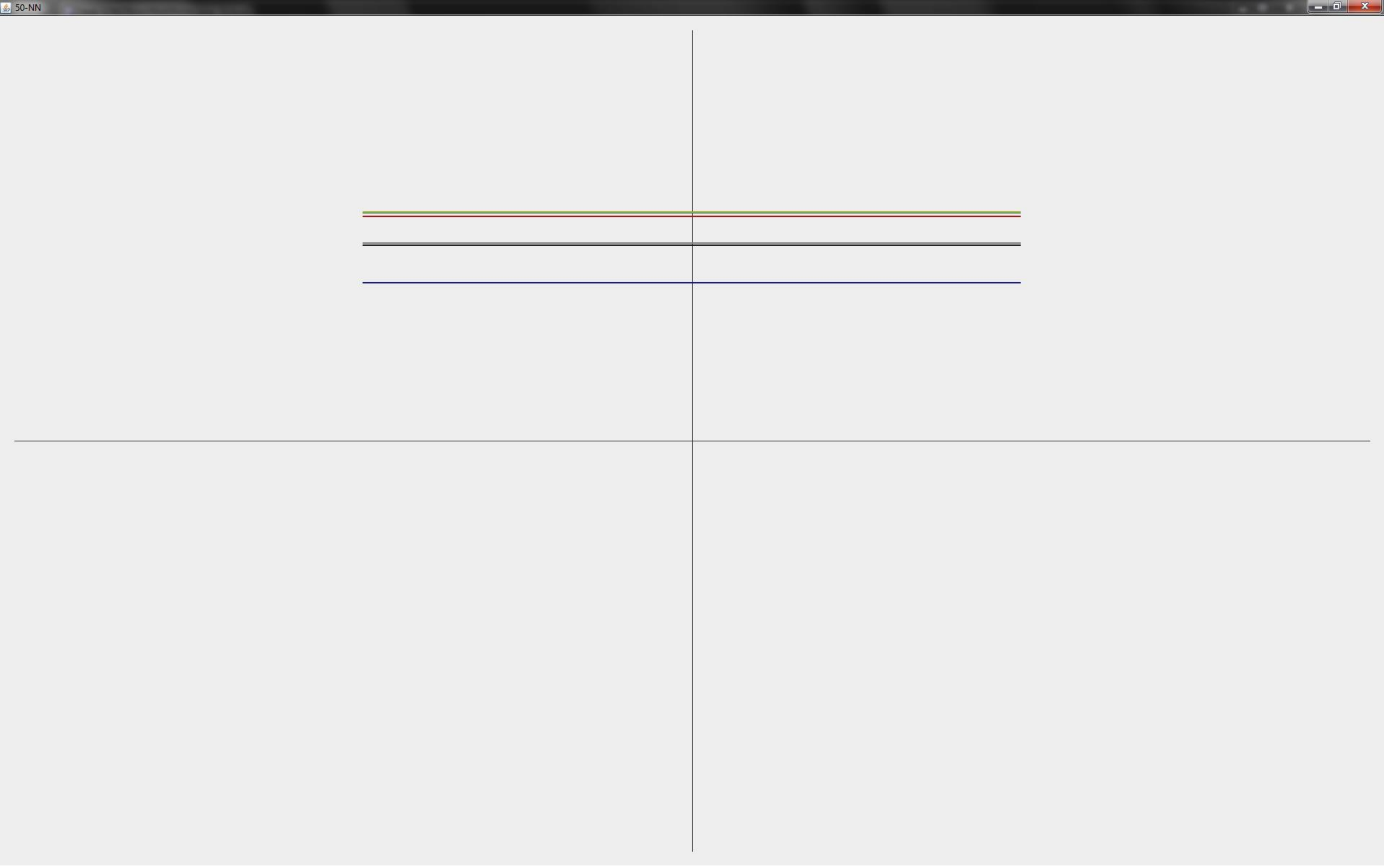
Average prediction over 200 different KNN models , each trained on a different data sample, for K=1 (red line) compared to the correct function $Y=X^2$ (green line). This plot shows the bias of the 1NN model. As expected, since 1NN makes predictions based on very small local neighborhoods, it can closely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



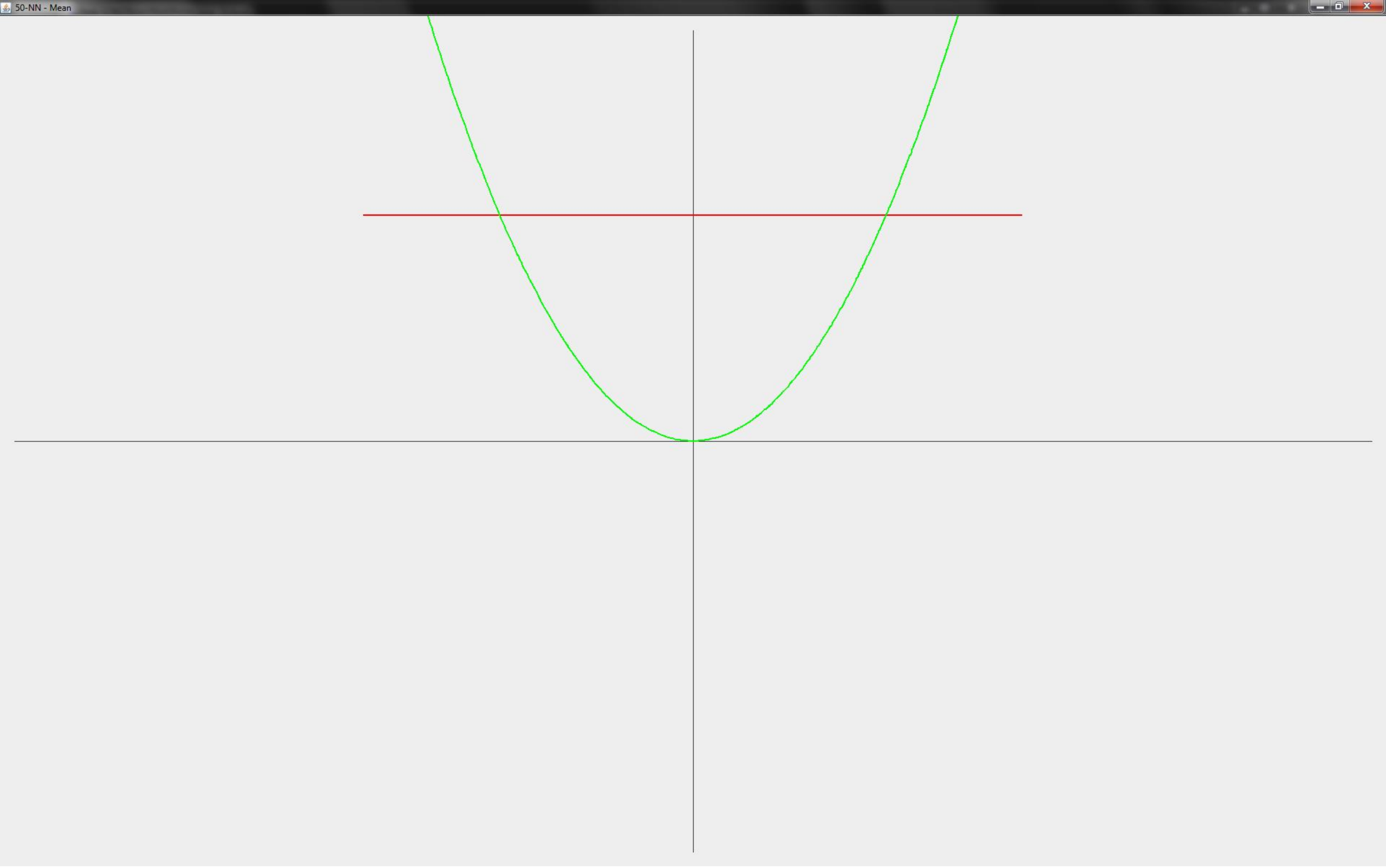
Predictions made by five different KNN models , each trained on a different data sample, for K=20. It is clearly visible that the individual models are less “noisy” than for 1NN, because 20NN averages over larger neighborhoods. (The horizontal lines at the left and right are caused by boundary effects 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 to 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 k independent bootstrap samples of D.
  - Train k 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 k models $M_1, \ldots, M_k$.
  - 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

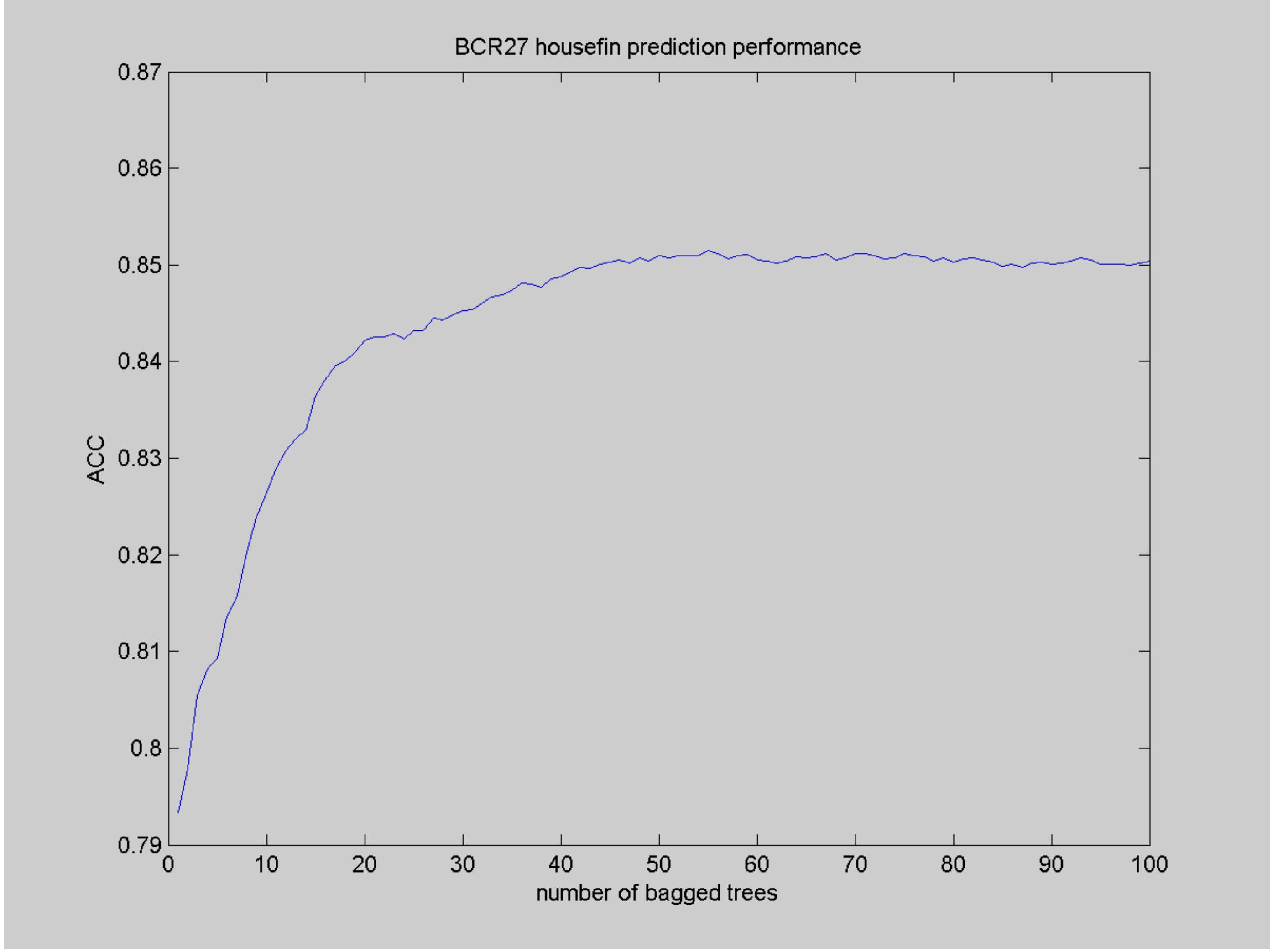
Typical bootstrap samples of D

Bagging Challenges

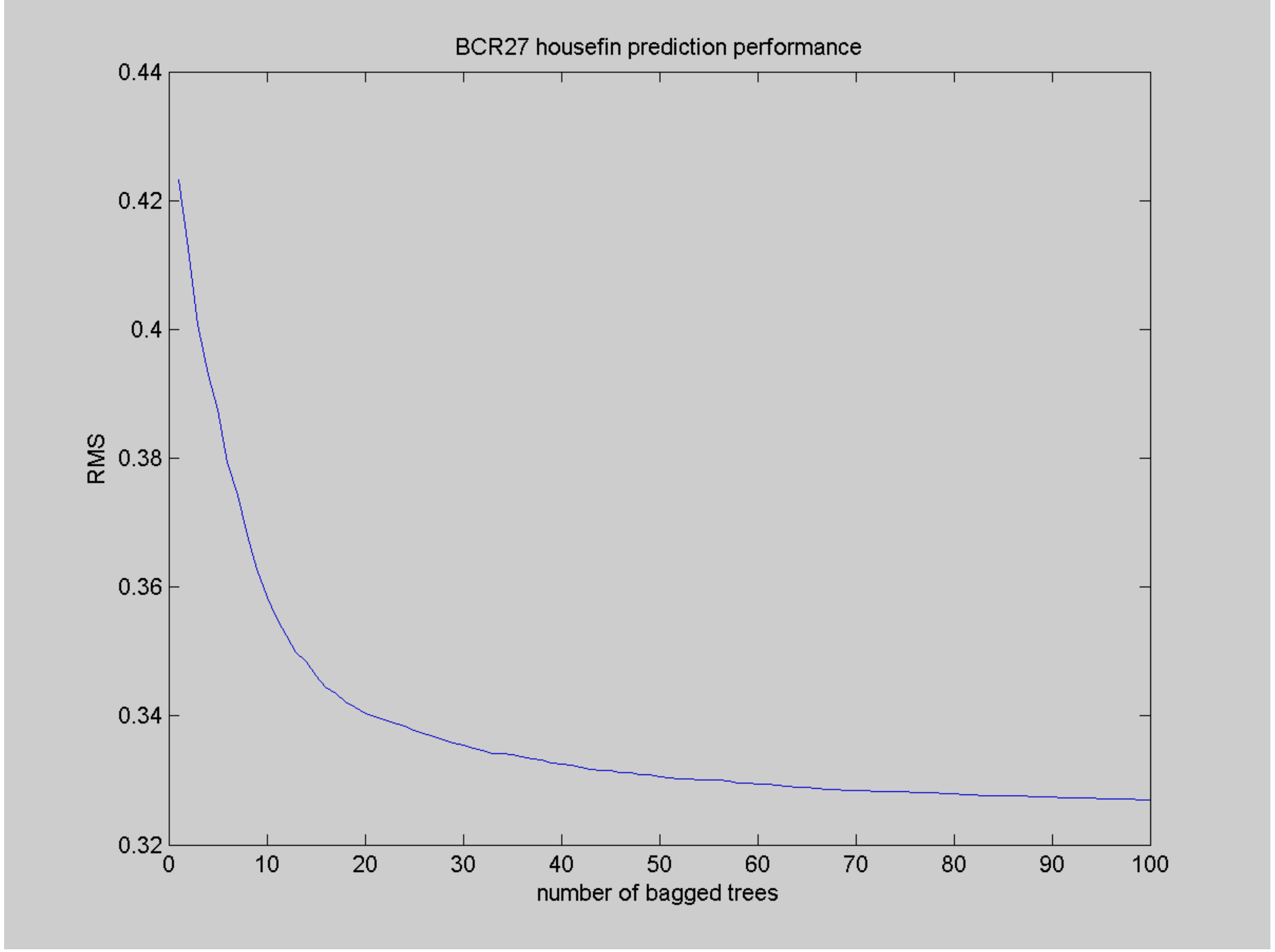
- The individual models in a bagged ensemble should be independent of each other. Only then can variance be effectively reduced through model averaging.
- Ideally model independence can be achieved through training on independent data samples. In practice, we usually has to settle for less because (1) 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.
  - Using small training data jeopardizes prediction quality, which is particularly dangerous for ensemble models. (Recall that each model has to be more than 50% accurate for the ensemble to improve over an individual model.)
- Bootstrap sampling represents a practical solution to achieve both reasonably independent training samples and large sample size. In contrast, simply partitioning D into k subsets would create more-independent training sets, but each would have only n/k records and hence might not capture the data distribution well.
- Independence can also 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

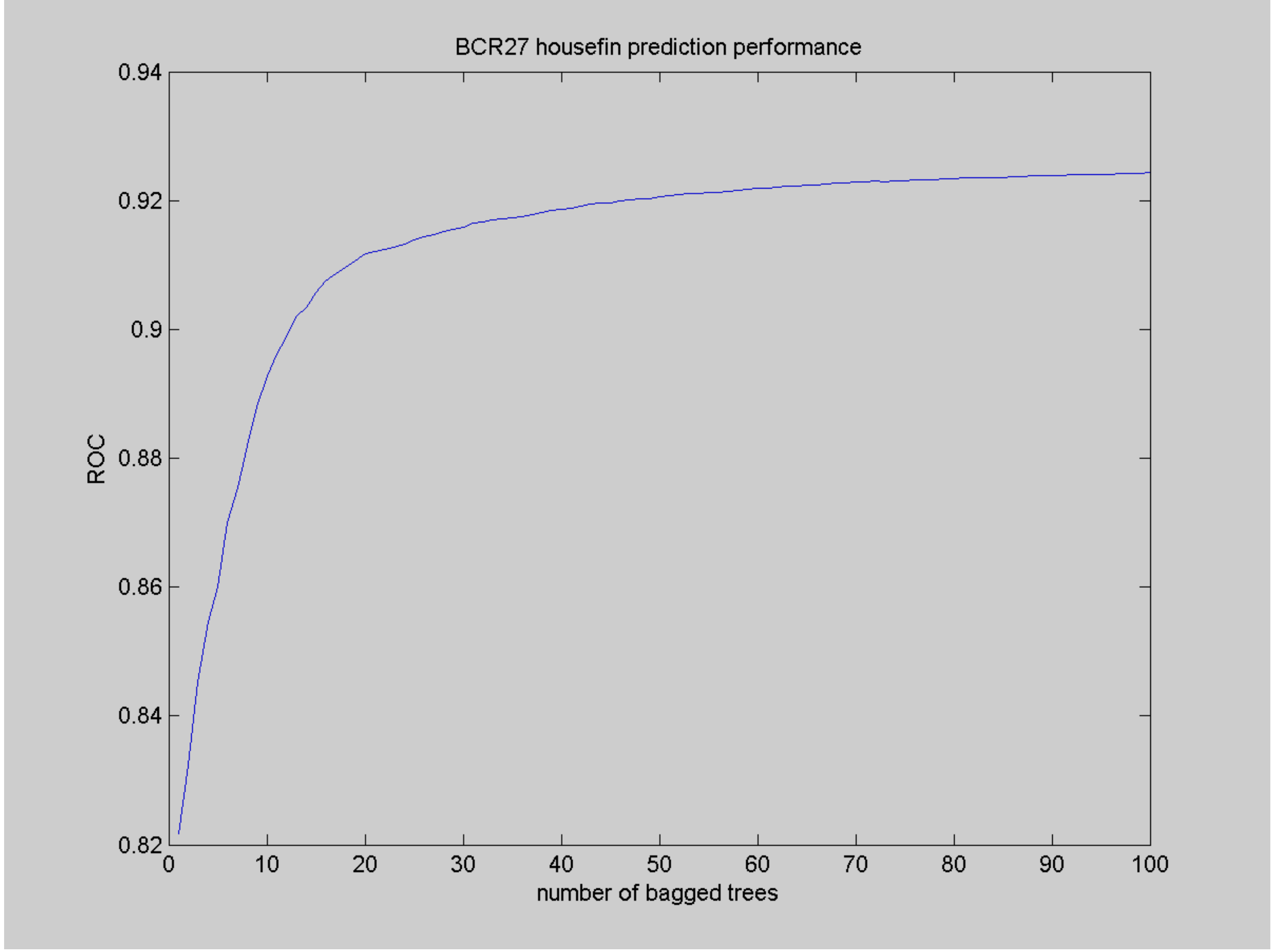
- For bagging to improve significantly over individual models, the ensemble might need dozens or even hundreds of individual models. Since bagging reduces variance without affecting bias, each individual model should overfit, having low bias even at the cost of high variance.
  - For decision trees, 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 comparably easy to distribute 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.
- Existing libraries for non-distributed model training can be leveraged by having each individual model trained on a single worker machine.

Parallel Training

- Assume a machine learning library is available for in-memory training on a single machine. Each of the k models in the ensemble can be trained on a different worker. This worker only needs access to a bootstrap sample and model parameters to control the training process.
  - Model parameters can be passed to all Reducers using the distributed file cache. Each line in the file would state the model identifier and corresponding parameters.
- Mappers create k copies of each data record and send them to k 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. (This also improves model independence.)

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

Model parameter file:

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 distributed file cache}

 reduce(i, [r1, r2,…])
  R = load input list 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·k copies of each input record from Mappers to Reducers. The Map-only program below avoids this transfer:
  - The entire training set is copied to all worker machines using the distributed file cache.
  - Mappers locally create bootstrap samples from this file, training a model for each sample.
- This program transfers as many copies of each input record from the distributed file system to the worker machines 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?
  - This information can be provided in a file, where each line contains a number from 1 to k (the model identifier) and the parameters for that model. By making this file the input of the MapReduce job, each Map call can train the corresponding model.
  - Since the input file is small, but each line creates a large amount of work for sampling and model training, the default setting of a single Map task per file split would be too coarse-grained, possibly resulting in only a single Map task. The NLineInputFormat class can be used to create smaller input splits.

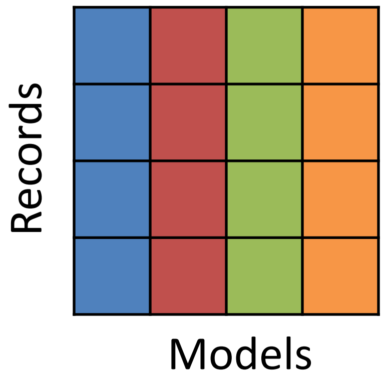
```
map( model number i, model_parameters ) {
  read the training data from the distributed file cache, creating a sample S that fits in memory, using sampling rate p

  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 cross-product between the set of models and the set of test records.
  – The cross-product is followed by a simple aggregation, computing the average prediction (or majority vote) for each test record.
- What is the best way to partition the cross-product over multiple tasks? We will discuss three options and compare their properties.

Partitioning on Models and Test Records: Vertical Stripes

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

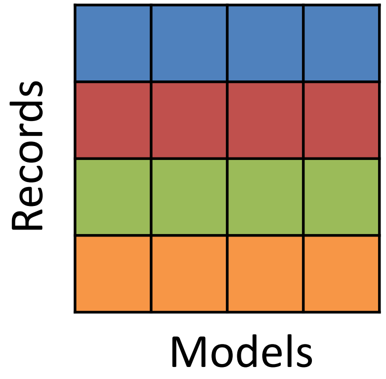


```
Class Mapper {
  T = read all test records from file cache

  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)
}
```

Partitioning on Models and Test Records: Horizontal Stripes

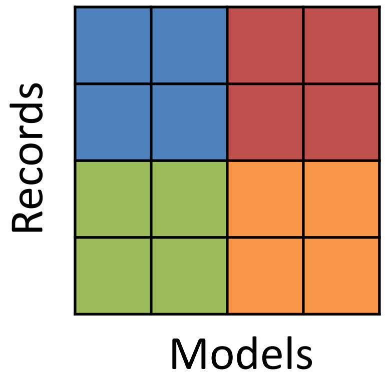
- To implement this horizontal partitioning in MapReduce, the model file is copied to all Mapper machines using the distributed file cache. The file containing the test records is the input of the MapReduce job.
- Since each Map task has the entire bagged model, it can compute the average prediction 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.
  - If the entire ensemble does not fit in memory, this approach would page models between disk and memory for every test record. To avoid this, in-mapper combining could be used to collect all test records. Then predictions are computed by going through models in the outer loop and test records in the inner loop, keeping track of running sum and count for each test record. (This assumes that the test file split fits in memory, which it usually does for typical split sizes like 256 MB.)
- Data transfer cost (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 )
 }
}
```

Partitioning on Models and Test Records: Blocks

- To implement block partitioning into A by B blocks in MapReduce, both test record and model file have to be appropriately partitioned and duplicated. The algorithm is identical to 1-Bucket-Random, which we discuss in another module. Note that for the final result, another post-processing job is needed to aggregate predictions across the different blocks (not shown below).
- Data transfer cost (without in-reducer combining, 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: A * B * file with prediction sum and count for each test record
  - Post-processing: read from HDFS A * B * file 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 currently (October 2018) offers two types of ensembles: Random Forest (variation of bagging) and Gradient-Boosted Trees (GBT).
- 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}")
```

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 predictions errors made by the i-th model affect the training of the (i+1)-st model.
- Hence the only source of parallelism is in the approach of training an individual model.
- Since Spark already offers parallel training of individual tree models, Gradient-Boosted Trees 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)

// Make predictions
```

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).

4. CYK: Q1

- Based on the bias-variance tradeoff, what is the best way to achieve high prediction accuracy for a bagged ensemble of decision trees? Choose the best answer:

1. Trees should not be used for bagging because they have poor prediction accuracy.
2. Make each individual tree in the ensemble as accurate as possible on its own, i.e., without considering other trees in the ensemble.
3. Make each tree significantly “smaller”, i.e., shallower and with fewer split nodes, than it would be if one tried to make it as accurate as possible on its own.
4. Make each tree significantly “bigger”, i.e., deeper and with more split nodes, than it would be if one tried to make it as accurate as possible on its own.
5. It does not really matter, as long as we add sufficiently many trees to the ensemble.

Q2

- Consider test data set $T=\{t_1, t_2,\ldots, t_n\}$ and a bagged model consisting of individual models $M=\{M_1, M_2,\ldots, M_k\}$. Recall that the ensemble prediction for test record t is the average of individual predictions $M_1(t)$, $M_2(t)$,…, $M_k(t)$. For each of the following statements, indicate if it is true or false.

1. Consider a problem partitioning where each worker receives a subset of T and the entire M. Using this partitioning, all ensemble predictions can be computed efficiently in parallel in a Map-only job.
2. Consider a problem partitioning where each worker receives a subset of M and the entire T. Using this partitioning, all ensemble predictions can be computed efficiently in parallel in a Map-only job.
3. Consider a problem partitioning where each worker receives a subset of T and a subset of M. Using this partitioning, all ensemble predictions can be computed efficiently in parallel in a Map-only 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=1175397538205464231O&hl=en&as_sdt=0,22