Lecture 11: Decision Trees

TTIC 31020: Introduction to Machine Learning

Instructor: Kevin Gimpel

TTI-Chicago

November 5, 2019

Review: classification

- We saw several approaches that can be viewed as minimizing a loss function using (sub)gradient-based optimization
- They have unique histories and motivations; the unifying view as minimizing a loss function is more recent

Review: classification

- We saw several approaches that can be viewed as minimizing a loss function using (sub)gradient-based optimization
- They have unique histories and motivations; the unifying view as minimizing a loss function is more recent
    - **Squared loss**: inspired by viewing classification as regression; easy to optimize (closed-form solution) though not often used for classification in practice

Review: classification

- We saw several approaches that can be viewed as minimizing a loss function using (sub)gradient-based optimization
- They have unique histories and motivations; the unifying view as minimizing a loss function is more recent
  - **Squared loss**: inspired by viewing classification as regression; easy to optimize (closed-form solution) though not often used for classification in practice
  - **Log loss**: loss for logistic regression which has a probabilistic interpretation; can be motivated by defining log-odds in terms of a linear boundary

Review: classification

- We saw several approaches that can be viewed as minimizing a loss function using (sub)gradient-based optimization
- They have unique histories and motivations; the unifying view as minimizing a loss function is more recent
  - **Squared loss**: inspired by viewing classification as regression; easy to optimize (closed-form solution) though not often used for classification in practice
  - **Log loss**: loss for logistic regression which has a probabilistic interpretation; can be motivated by defining log-odds in terms of a linear boundary
  - **Perceptron loss**: reverse-engineered from the perceptron algorithm, a mistake-driven learning algorithm with guarantees

Review: classification

- We saw several approaches that can be viewed as minimizing a loss function using (sub)gradient-based optimization
- They have unique histories and motivations; the unifying view as minimizing a loss function is more recent
  - **Squared loss**: inspired by viewing classification as regression; easy to optimize (closed-form solution) though not often used for classification in practice
  - **Log loss**: loss for logistic regression which has a probabilistic interpretation; can be motivated by defining log-odds in terms of a linear boundary
  - **Perceptron loss**: reverse-engineered from the perceptron algorithm, a mistake-driven learning algorithm with guarantees
  - **Hinge loss**: exported from SVMs; motivated as finding max-margin separator

Review: binary vs. multi-class classification

- We talked about binary classification in class
- Extensions to multi-class classification exist (e.g., softmax classifier on pset2)
- Sometimes has form of training multiple binary "one vs. all" classifiers

Review: sparsity in classification

- Solving dual of SVM problem leads to sparsity in dual parameters (a relatively small number of "support vectors")
- Solving primal problem with subgradient descent can also lead to sparsity (depending on regularization)
  - If classification is correct (perceptron) or correct with large-enough margin (hinge), there is no loss suffered and therefore no parameter update
  - Intuitively: if a feature is zero for all examples for which a parameter update is performed, its weight won't change (amount of sparsity depending on regularization and initialization)

Review: kernels

- We talked about kernels in the context of SVMs
- But kernels can be used any time we can write a machine learning method in terms of dot products between inputs
- SVMs are a good fit for kernels because of sparse solutions (only support vectors are needed, which reduces computational requirements of using kernels)

Review: (sub)gradient descent optimization

General "pipeline" of developing a learning algorithm:

- Write down the complete parameterized objective:

$$\min_{\theta} \sum_{i} f(\theta; \mathbf{x}_i, y_i)$$

Review: (sub)gradient descent optimization

General "pipeline" of developing a learning algorithm:

- Write down the complete parameterized objective:

$$\min_{\theta} \sum_{i} f(\theta; \mathbf{x}_i, y_i)$$

- Write down the gradient of $f_i = f(\theta; \mathbf{x}_i, y_i)$

$$g_i(\theta, \mathbf{x}_i, y_i) = \frac{\partial f_i}{\partial \theta}$$

Review: (sub)gradient descent optimization

General "pipeline" of developing a learning algorithm:

- Write down the complete parameterized objective:

$$\min_{\theta} \sum_{i} f(\theta; \mathbf{x}_i, y_i)$$

- Write down the gradient of $f_i = f(\theta; \mathbf{x}_i, y_i)$

$$g_i(\theta, \mathbf{x}_i, y_i) \; = \; \frac{\partial f_i}{\partial \theta}$$

- If f_i is not differentiable at a particular θ', but subgradient exists, write it instead. Typically, the form will be

$$g_i(\theta, \mathbf{x}_i, y_i) \; = \; \begin{cases} g_i^1(\theta, \mathbf{x}_i, y_i) & \text{if } \theta \leq \theta', \\ g_i^2(\theta, \mathbf{x}_i, y_i) & \text{if } \theta > \theta'. \end{cases}$$

Review: (sub)gradient descent optimization

General "pipeline" of developing a learning algorithm:

- Write down the complete parameterized objective:

$$\min_{\theta} \sum_{i} f(\theta; \mathbf{x}_i, y_i)$$

- Write down the gradient of $f_i = f(\theta; \mathbf{x}_i, y_i)$

$$g_i(\theta, \mathbf{x}_i, y_i) \;=\; \frac{\partial f_i}{\partial \theta}$$

- If f_i is not differentiable at a particular θ', but subgradient exists, write it instead. Typically, the form will be

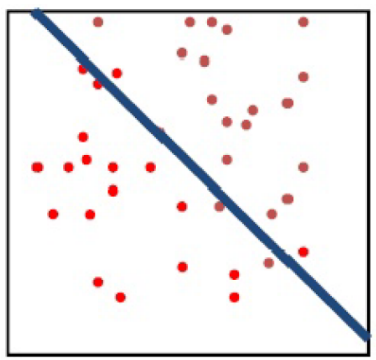
$$g_i(\theta, \mathbf{x}_i, y_i) \;=\; \begin{cases} g_i^1(\theta, \mathbf{x}_i, y_i) & \text{if } \theta \leq \theta', \\ g_i^2(\theta, \mathbf{x}_i, y_i) & \text{if } \theta > \theta'. \end{cases}$$

- Implement gradient descent, e.g., stochastic:

$$\theta^{(t+1)} \;=\; \theta^{(t)} - \eta_t g_{i(t)}(\theta, \mathbf{x}_{i(t)}, y_{i(t)})$$

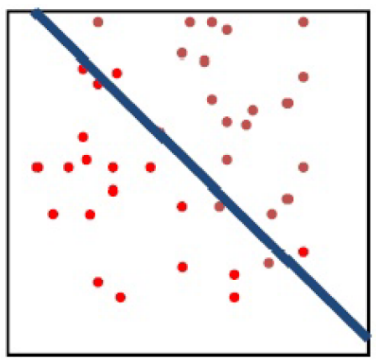
Space partition by classifiers

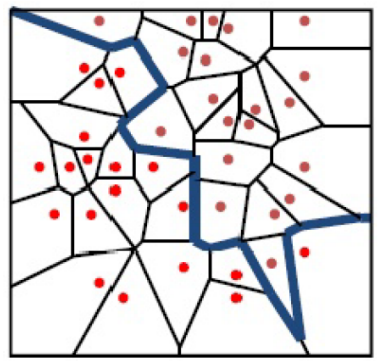
- We learn a classifier to partition the space of $\mathcal{X}$ according to predicted $\mathcal{Y}$
- Linear classifier: linear partition (hyperplane)



Space partition by classifiers

- We learn a classifier to partition the space of $\mathcal{X}$ according to predicted $\mathcal{Y}$
- Linear classifier: linear partition (hyperplane)

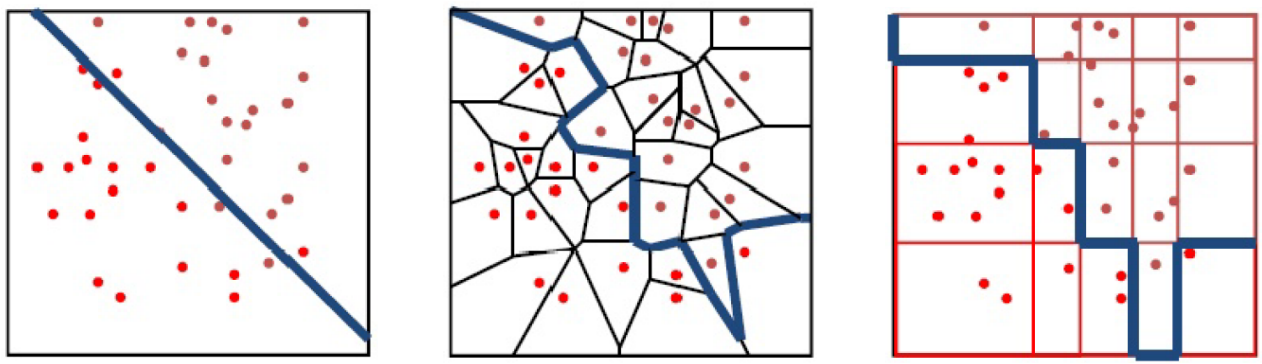




- kNN: Voronoi partition – data driven, non-linear, no simple parametric form

Space partition by classifiers

- We learn a classifier to partition the space of $\mathcal{X}$ according to predicted $\mathcal{Y}$
- Linear classifier: linear partition (hyperplane)



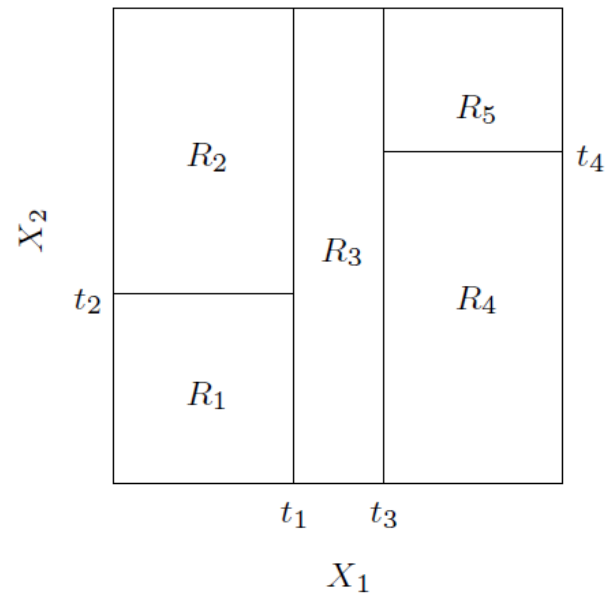
- kNN: Voronoi partition – data driven, non-linear, no simple parametric form
- Today: decision trees – non-linear partition, but with a simple parametric form

Space partition

- Decision boundary: partition the space into regions with fixed prediction value
- Linear classifier: regions = half spaces

Space partition

- Decision boundary: partition the space into regions with fixed prediction value
- Linear classifier: regions = half spaces
- More general partition: **hyper-rectangles**

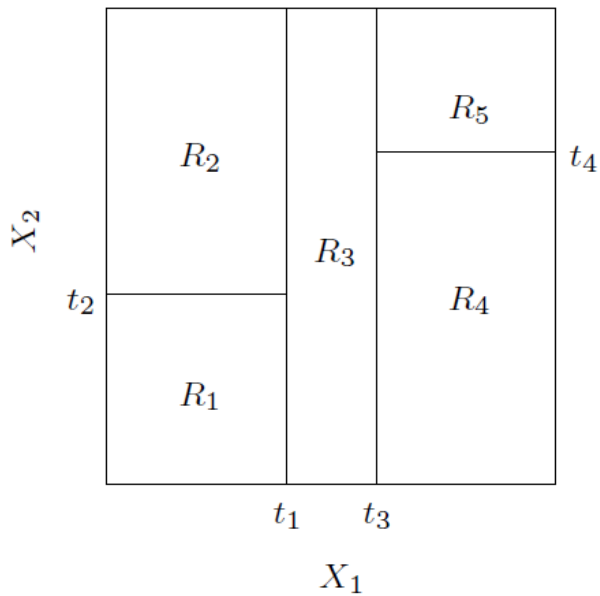


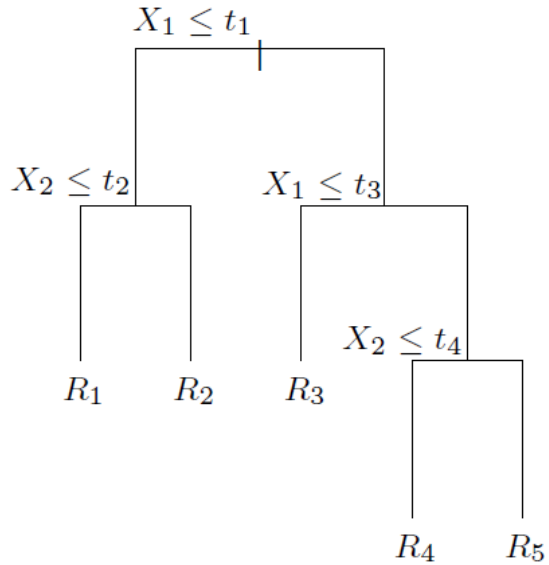
- The regions are easy to describe, e.g.,

$$R_2 = \{\mathbf{x} : x_1 < t_1 \text{ and } x_2 > t_2\}$$

Partition tree

- We can describe a non-overlapping partition of the space into hyper-rectangles via a tree:

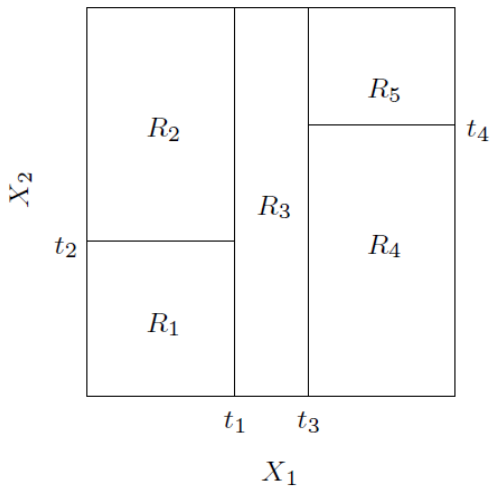


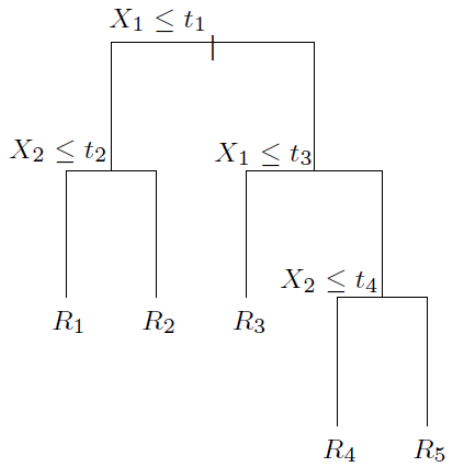


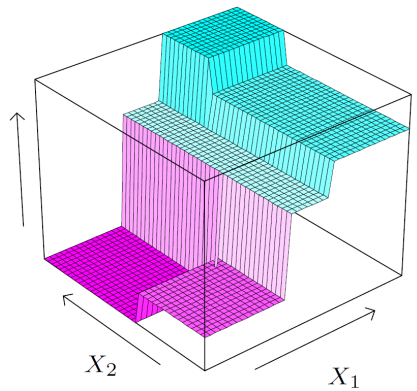
- Regions correspond to leaves
- A point is placed in a region by "dropping" it down the tree, applying a test at each node

Classification and regression trees

- Associate each leaf with a fixed prediction value
- E.g., regression:



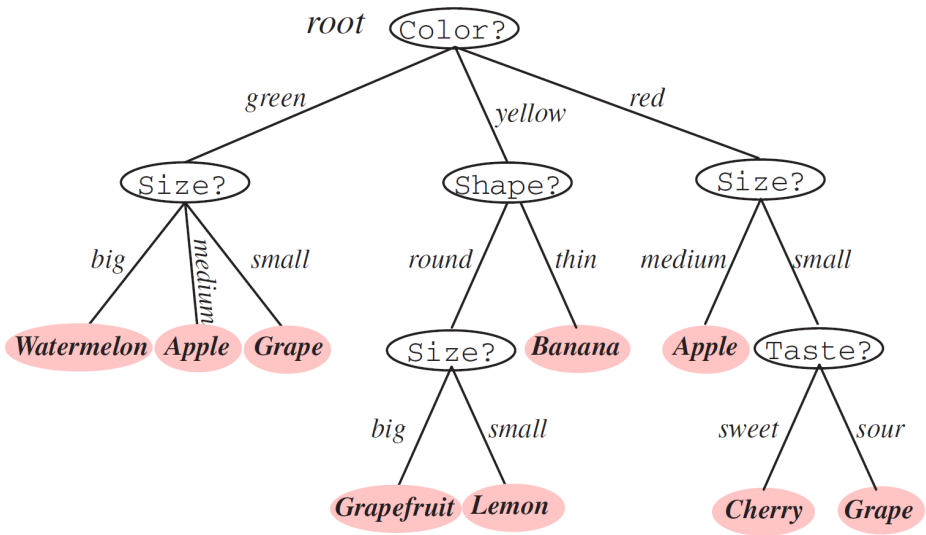


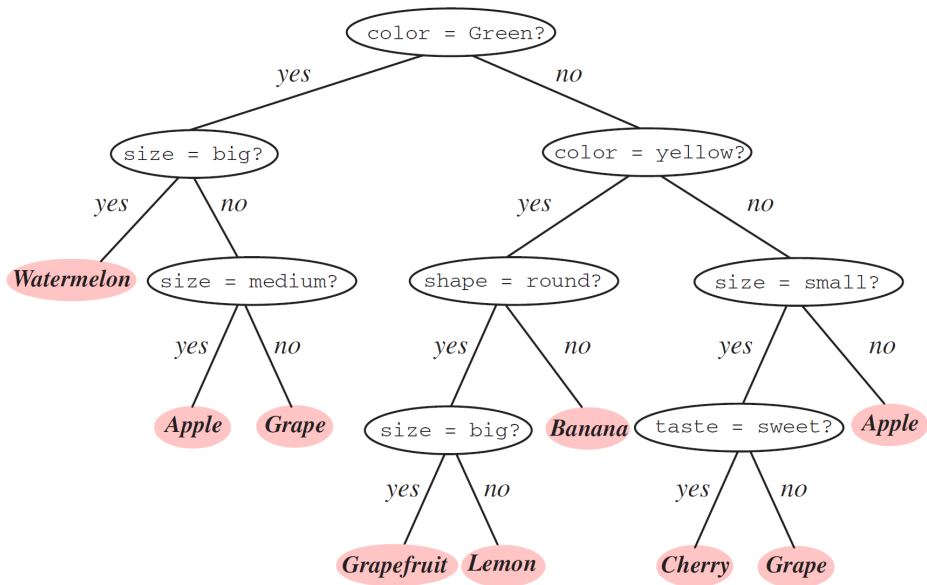


- Key questions:
  How do we build (learn) the tree?
  What's the bias/variance tradeoff?

Split factor

- We will focus on binary trees
- This is sufficient: can always convert any tree to binary





from Duda, Hart, and Stork

Regression trees

- Model corresponding to a tree with M leaves; leaf m corresponds to region R_m which predicts value f_m

$$f(\mathbf{x}) = \sum_{m=1}^{M} f_m \, \mathbb{I}[\mathbf{x} \in R_m]$$

  where $\mathbb{I}[A] = 1$ if the predicate A is true and 0 otherwise

Regression trees

- Model corresponding to a tree with M leaves; leaf m corresponds to region R_m which predicts value f_m

$$f(\mathbf{x}) = \sum_{m=1}^{M} f_m \, \mathbb{I}[\mathbf{x} \in R_m]$$

  where $\mathbb{I}[A] = 1$ if the predicate A is true and 0 otherwise
- Given a tree (partition), how do we minimize the squared loss?

$$\min \sum_{i=1}^{n} (f(\mathbf{x}_i) - y_i)^2 \;\Rightarrow\; f_m = \frac{1}{|\mathcal{I}_m|} \sum_{i \in \mathcal{I}_m} y_i$$

  where $\mathcal{I}_m = \{i : \mathbf{x}_i \in R_m\}$; i.e., f_m is the average value of the training points in R_m.

Regression tree construction

- Goal: find $R_1, \ldots, R_M$ to minimize

$$\sum_{i=1}^{n} \left(\left(\sum_{m=1}^{M} f_m \, \mathbb{I}[\mathbf{x}_i \in R_m] \right) - y_i \right)^2$$

  This is not computationally tractable

- Greedy algorithm instead: consider a split at s along j-th feature,

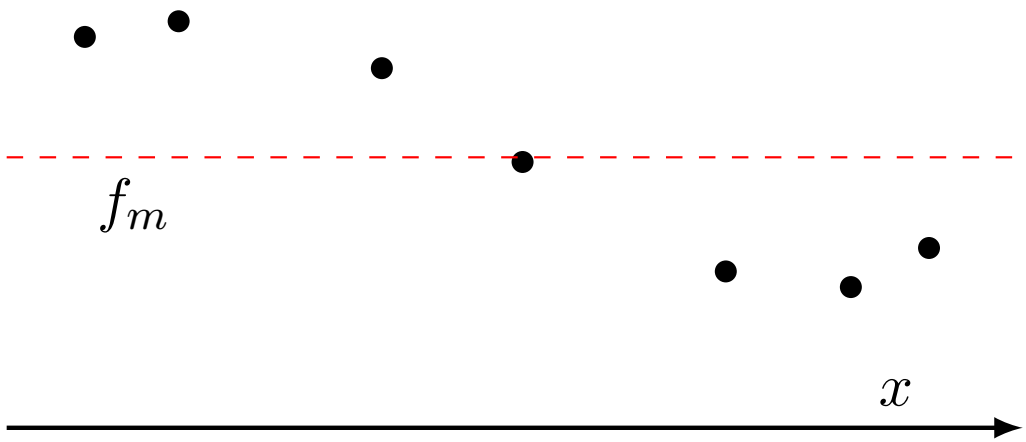
$$R_L(j, s) = \{\mathbf{x} : \phi_j(\mathbf{x}) \leq s\}, \quad R_U(j, s) = \{\mathbf{x} : \phi_j(\mathbf{x}) > s\}$$

- Cost of the split, assigning $R_L \to f_L$, $R_U \to f_U$:

$$\min_{f_L} \sum_{i:\mathbf{x}_i \in R_L} (y_i - f_L)^2 + \min_{f_U} \sum_{i:\mathbf{x}_i \in R_U} (y_i - f_U)^2$$

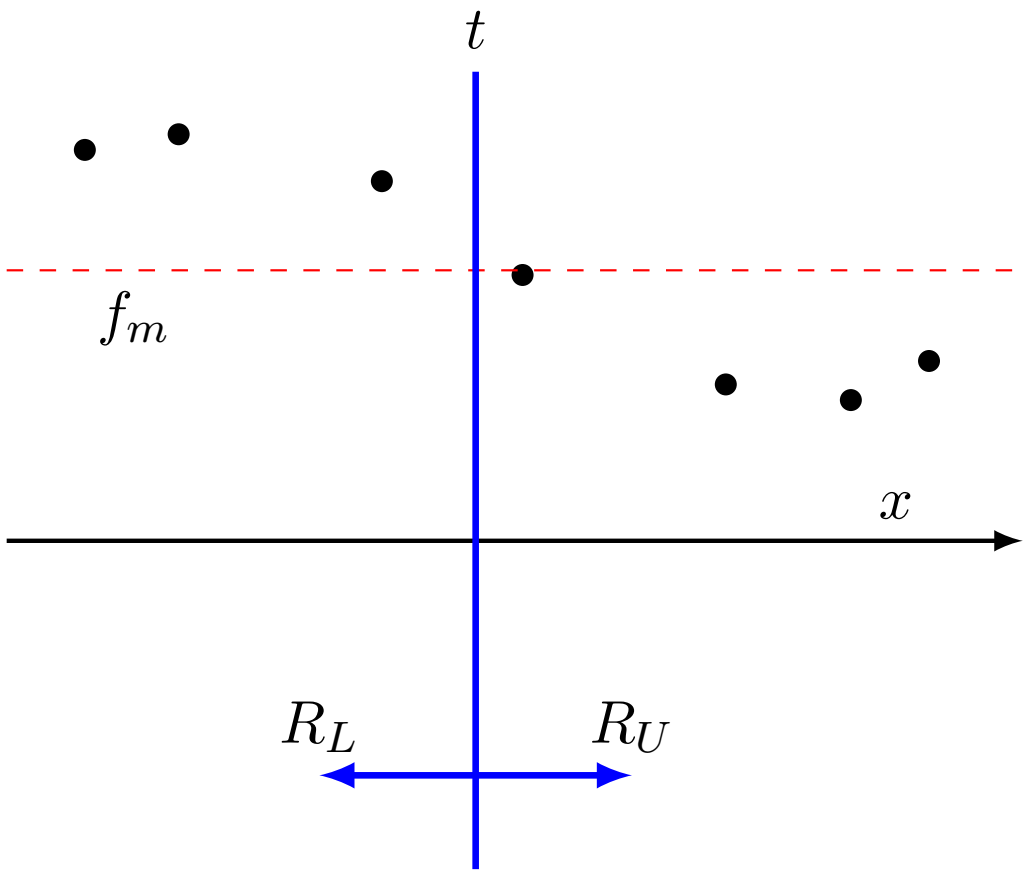
Region splitting: intuition

- 7 points in this R_m: mean value of y is f_m, the loss for this region is the mean squared difference $\frac{1}{7}\sum_i (y_i - f_m)^2$



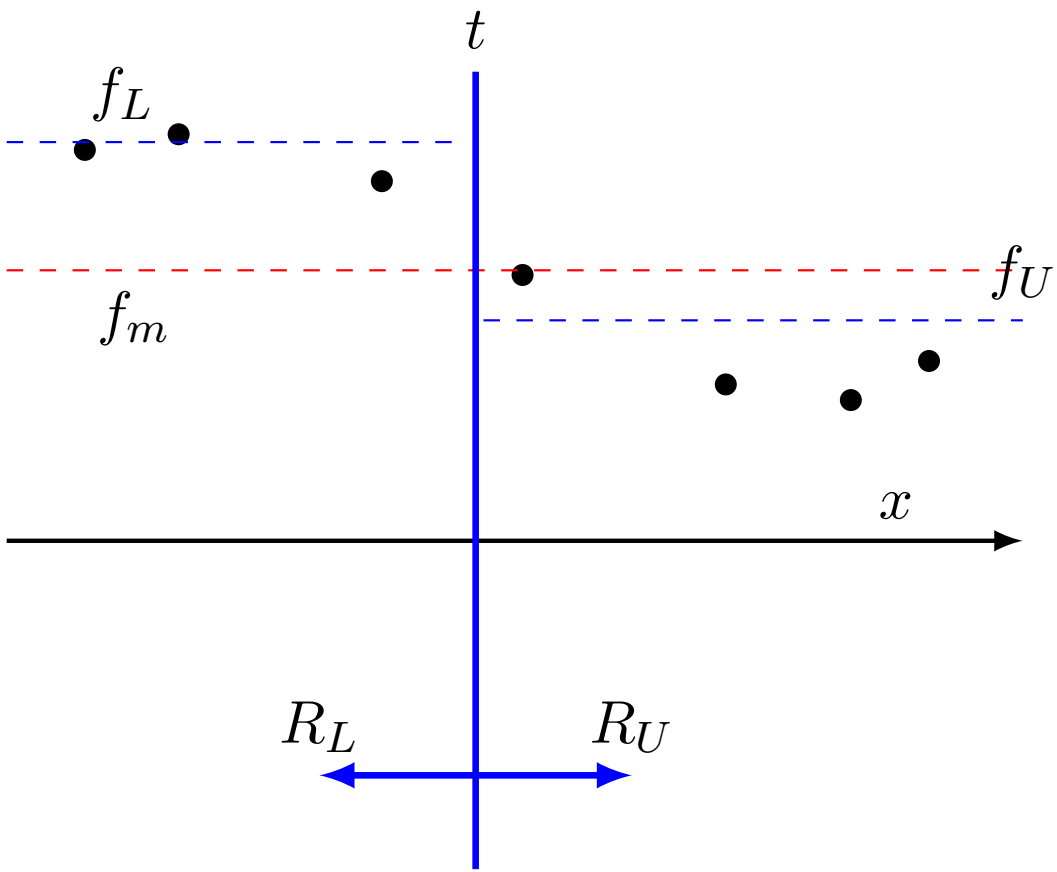
Region splitting: intuition

- 7 points in this R_m: mean value of y is f_m, the loss for this region is the mean squared difference $\frac{1}{7}\sum_i (y_i - f_m)^2$



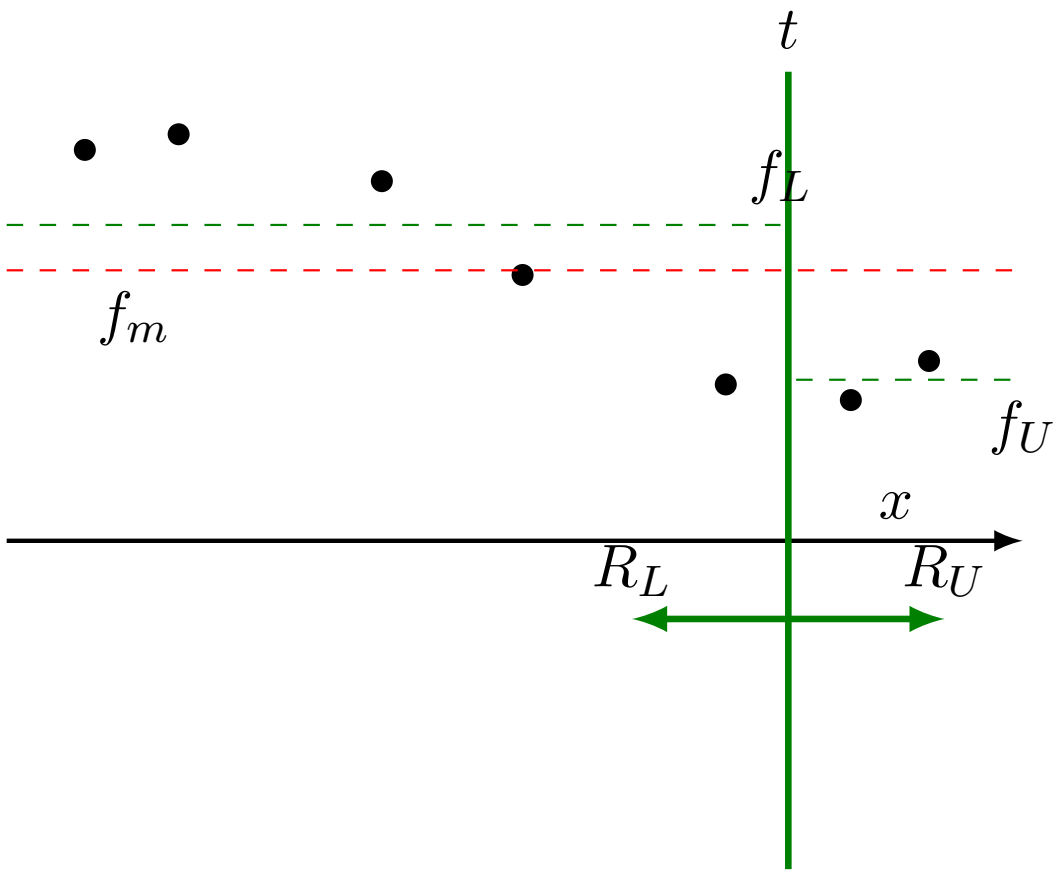
Region splitting: intuition

- 7 points in this R_m: mean value of y is f_m, the loss for this region is the mean squared difference $\frac{1}{7}\sum_i (y_i - f_m)^2$



Region splitting: intuition

- 7 points in this R_m: mean value of y is f_m, the loss for this region is the mean squared difference $\frac{1}{7}\sum_i (y_i - f_m)^2$



Regression tree construction

- So, we need to solve

$$\min_{j,s} \left\{ \min_{f_L} \sum_{i:\mathbf{x}_i \in R_L} (y_i - f_L)^2 + \min_{f_U} \sum_{i:\mathbf{x}_i \in R_U} (y_i - f_U)^2 \right\}$$

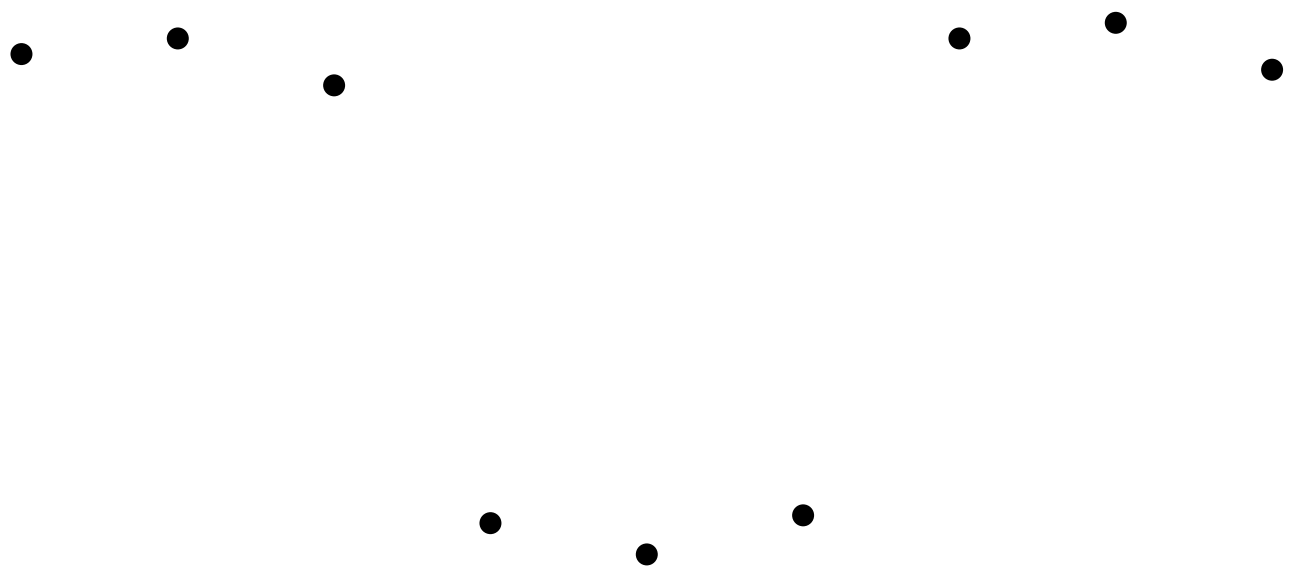
- We already know: for any j, s the f_L, f_U are the averages of the labels in induced R_L, R_U.
- We can exhaustively evaluate all distinct j, s pairs. (What's the running time?)
- Proceed recursively, partitioning R_U and R_L by the same procedure

Regression tree complexity

- Can easily overfit, getting zero training error!
- Idea for limiting model complexity: define notion of "gain", e.g., reduction in loss.
- Now, don't split if the gain is small.
  Turns out to be short-sighted (why?)

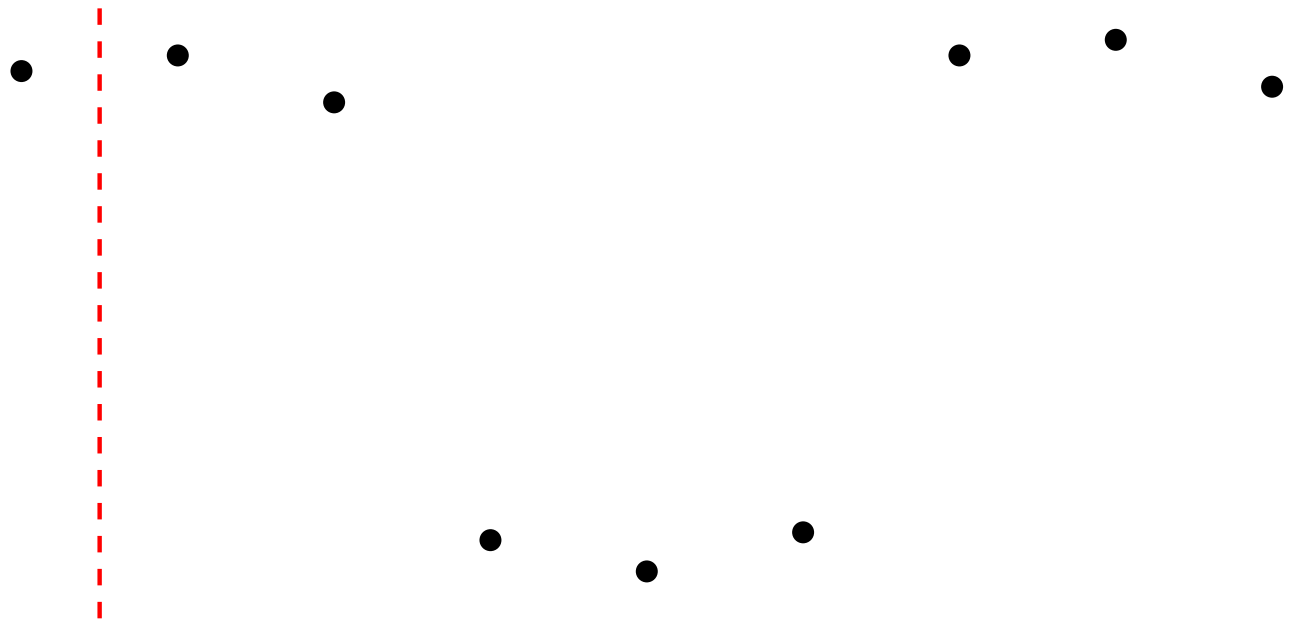
Regression tree complexity

- Can easily overfit, getting zero training error!
- Idea for limiting model complexity: define notion of "gain", e.g., reduction in loss.
- Now, don't split if the gain is small.
  Turns out to be short-sighted (why?)



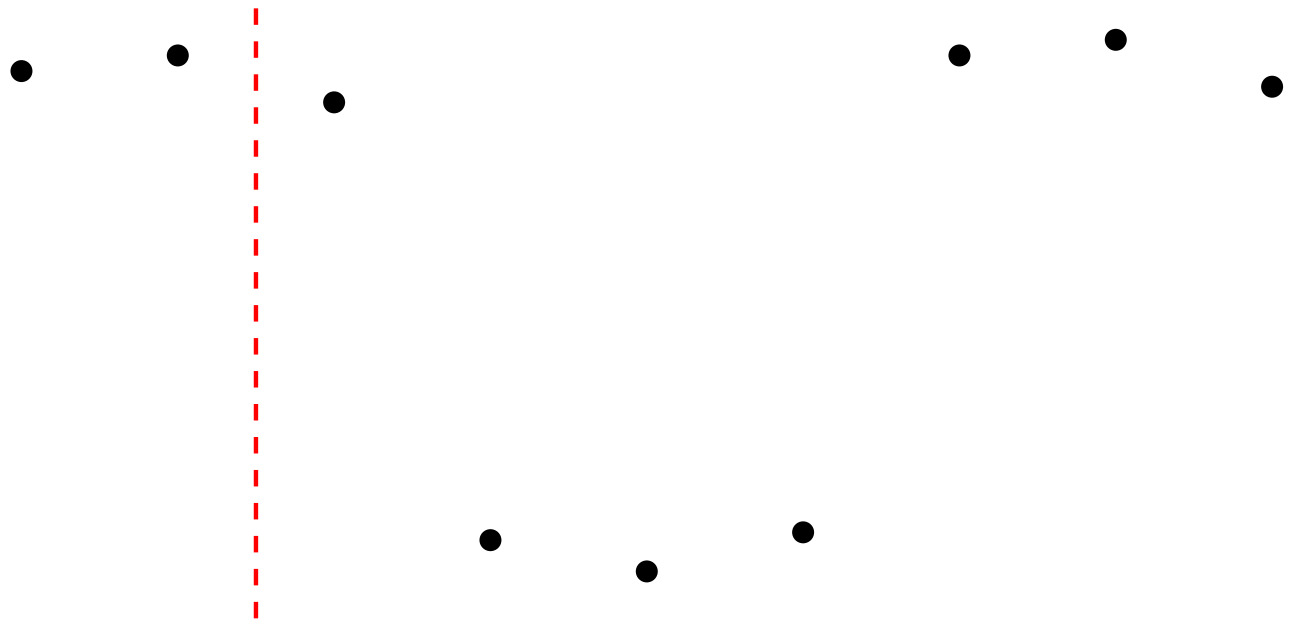
Regression tree complexity

- Can easily overfit, getting zero training error!
- Idea for limiting model complexity: define notion of "gain", e.g., reduction in loss.
- Now, don't split if the gain is small.
  Turns out to be short-sighted (why?)



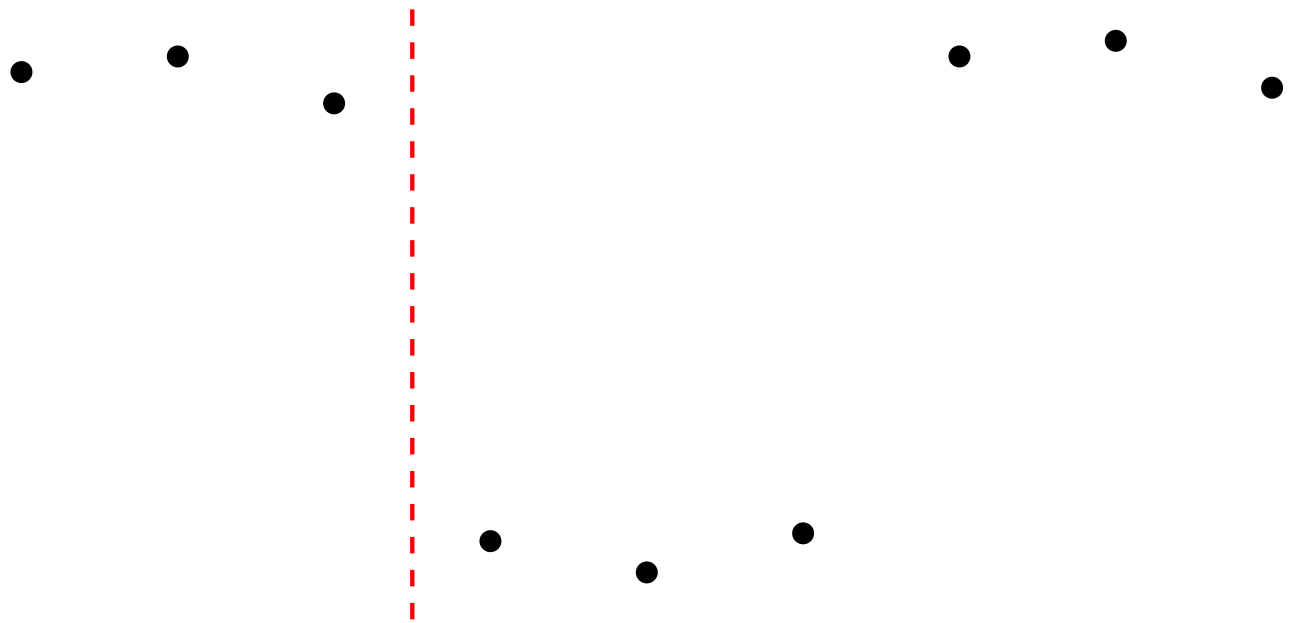
Regression tree complexity

- Can easily overfit, getting zero training error!
- Idea for limiting model complexity: define notion of "gain", e.g., reduction in loss.
- Now, don't split if the gain is small.
  Turns out to be short-sighted (why?)



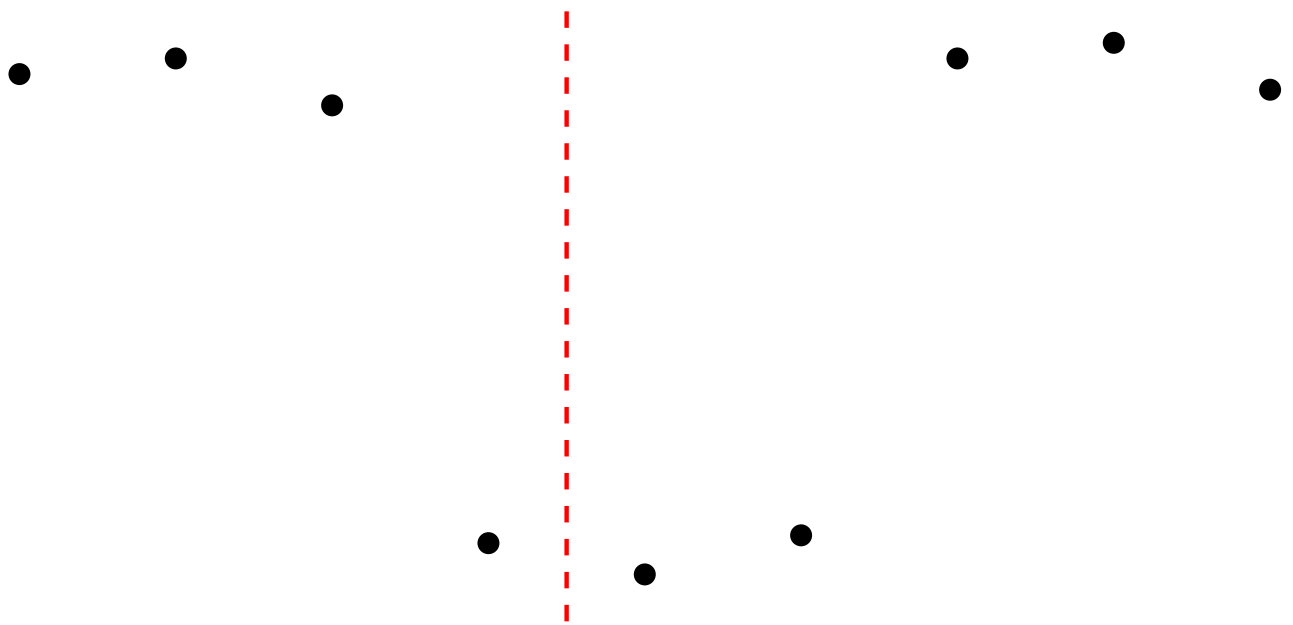
Regression tree complexity

- Can easily overfit, getting zero training error!
- Idea for limiting model complexity: define notion of "gain", e.g., reduction in loss.
- Now, don't split if the gain is small.
  Turns out to be short-sighted (why?)



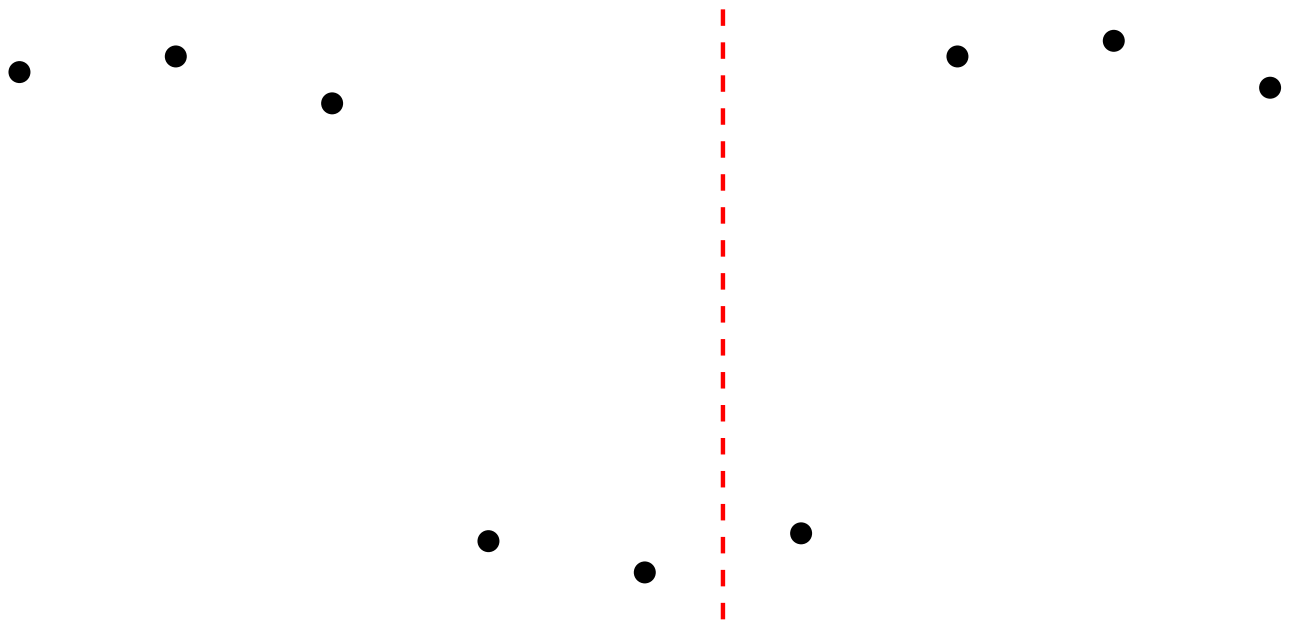
Regression tree complexity

- Can easily overfit, getting zero training error!
- Idea for limiting model complexity: define notion of "gain", e.g., reduction in loss.
- Now, don't split if the gain is small.
  Turns out to be short-sighted (why?)



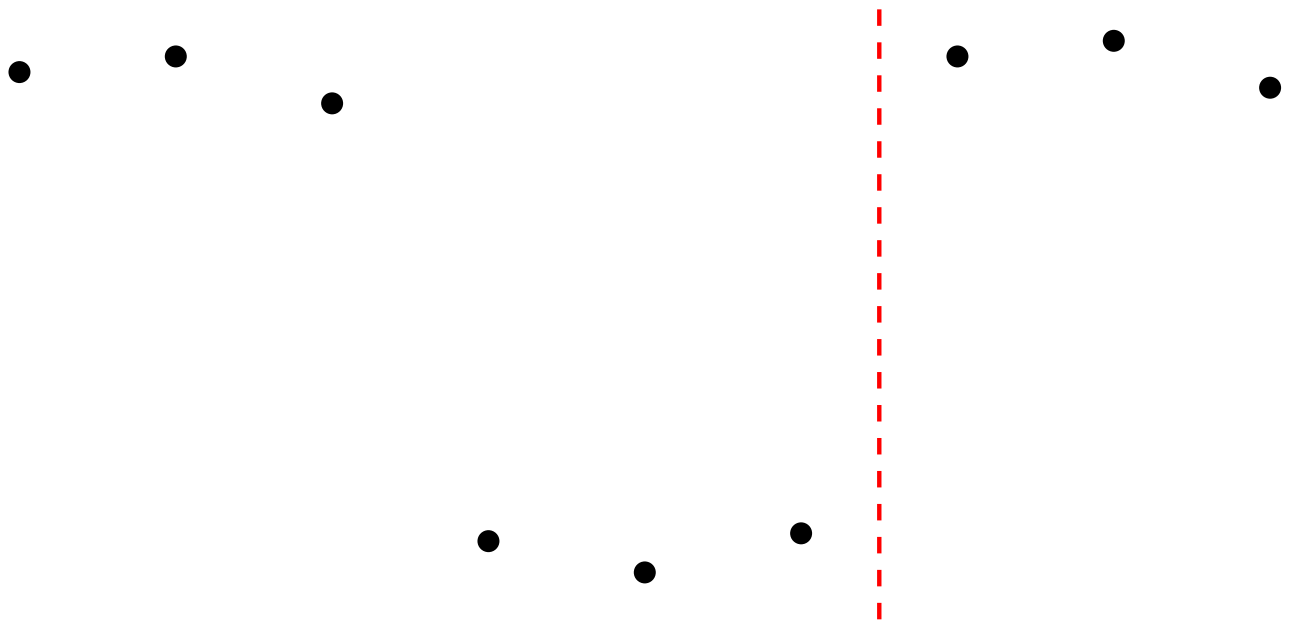
Regression tree complexity

- Can easily overfit, getting zero training error!
- Idea for limiting model complexity: define notion of "gain", e.g., reduction in loss.
- Now, don't split if the gain is small.
  Turns out to be short-sighted (why?)



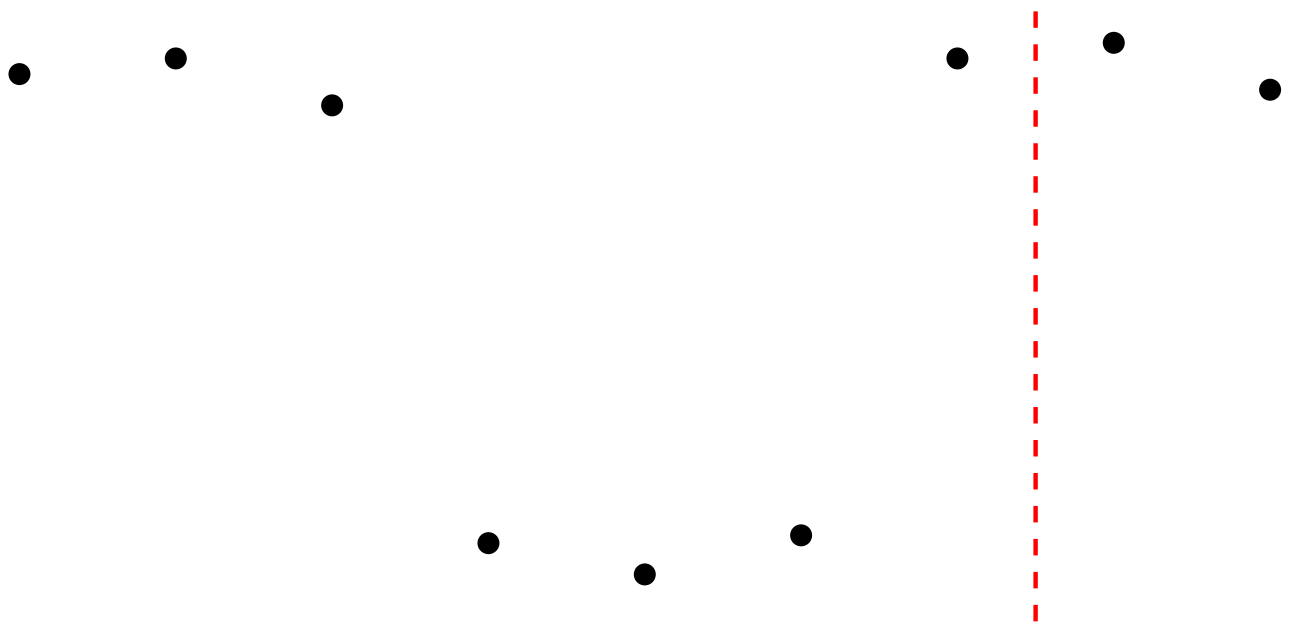
Regression tree complexity

- Can easily overfit, getting zero training error!
- Idea for limiting model complexity: define notion of "gain", e.g., reduction in loss.
- Now, don't split if the gain is small.
  Turns out to be short-sighted (why?)



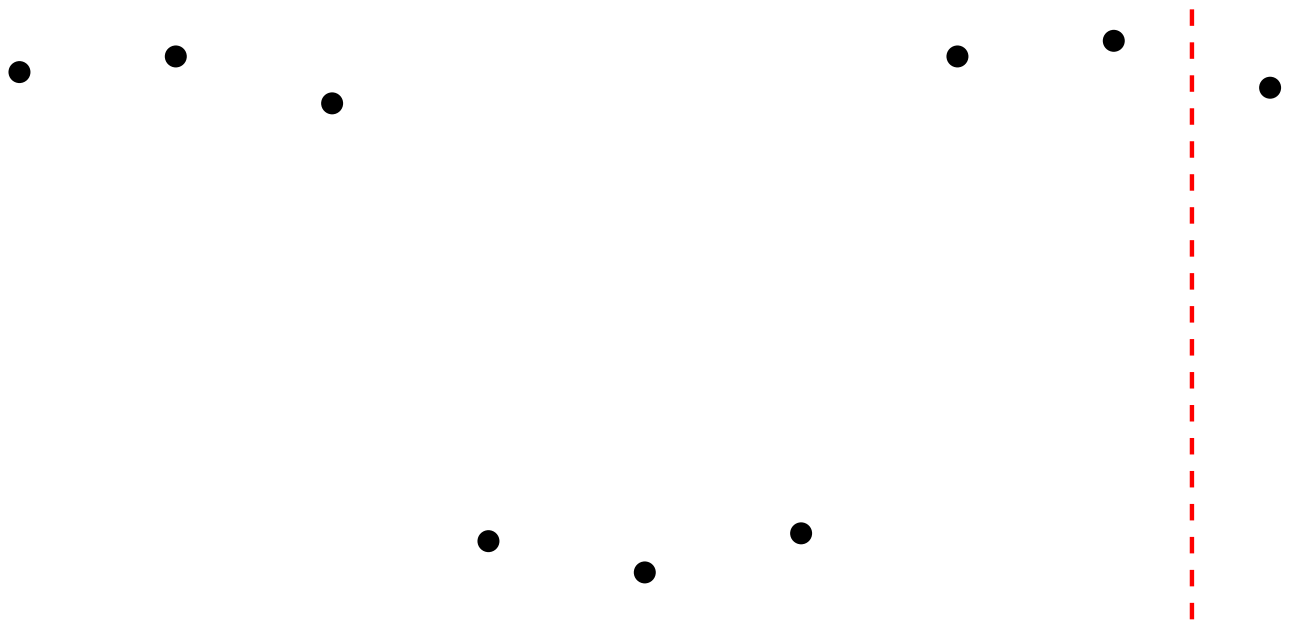
Regression tree complexity

- Can easily overfit, getting zero training error!
- Idea for limiting model complexity: define notion of "gain", e.g., reduction in loss.
- Now, don't split if the gain is small.
  Turns out to be short-sighted (why?)



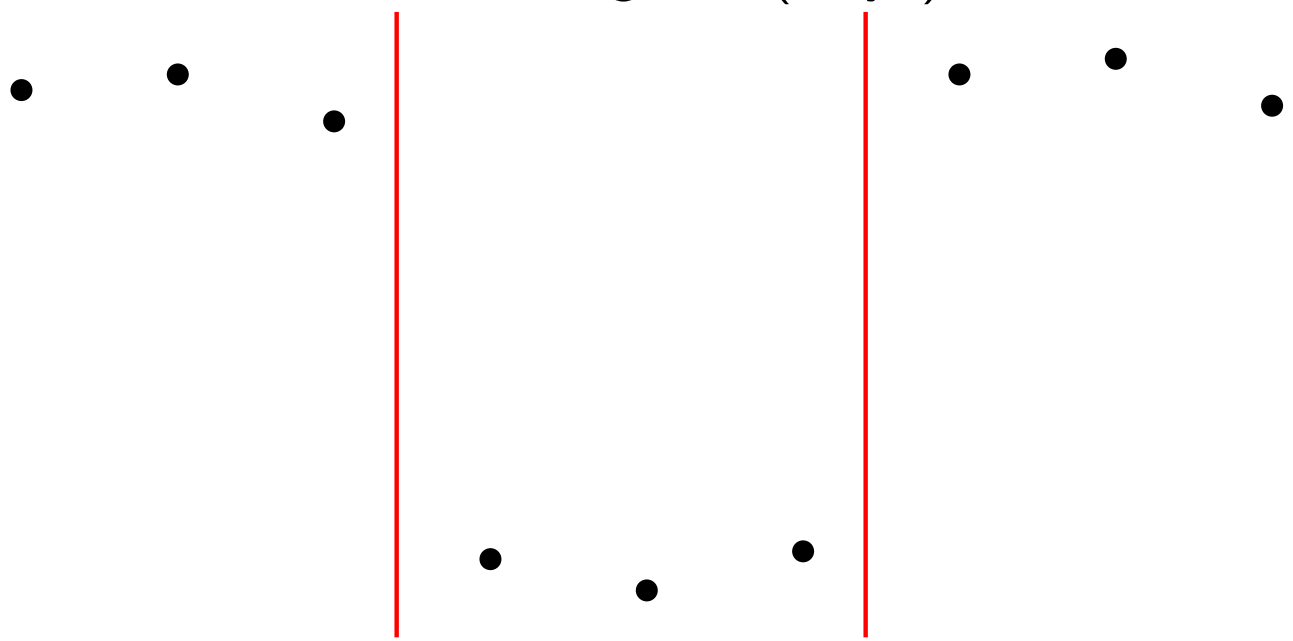
Regression tree complexity

- Can easily overfit, getting zero training error!
- Idea for limiting model complexity: define notion of "gain", e.g., reduction in loss.
- Now, don't split if the gain is small.
  Turns out to be short-sighted (why?)



Regression tree complexity

- Can easily overfit, getting zero training error!
- Idea for limiting model complexity: define notion of "gain", e.g., reduction in loss.
- Now, don't split if the gain is small.
  Turns out to be short-sighted (why?)



Regression tree complexity

- Can easily overfit, getting zero training error!
- Idea for limiting model complexity: don't split if the gain is small. Turns out to be short-sighted.
- Alternative: grow a large tree T_0, then *prune* to $T \subset T_0$

Regression tree complexity

- Can easily overfit, getting zero training error!
- Idea for limiting model complexity: don't split if the gain is small. Turns out to be short-sighted.
- Alternative: grow a large tree T_0, then *prune* to $T \subset T_0$
- Some more notation: $|T|$ is the number of leaves in T,
  $N_m = |\mathcal{I}_m| = |\{i : \mathbf{x}_i \in R_m\}|$ is the size of a leaf,
  $f_m = \frac{1}{N_m} \sum_{i \in \mathcal{I}_m} y_i$ is the value in the leaf, and
  $Q_m(T) = \frac{1}{N_m} \sum_{i \in \mathcal{I}_m} (y_i - f_m)^2$ is the leaf error.
- Cost-complexity criterion of tree $T \subset T_0$:

$$C_\lambda(T) = \lambda |T| + \sum_{m=1}^{|T|} N_m Q_m(T)$$

Regression tree pruning

$$C_\lambda(T) = \lambda|T| + \sum_{m=1}^{|T|} N_m Q_m(T)$$

- T is obtained from T_0 by collapsing some internal nodes (merging multiple leaves)
- For a given $\lambda \geq 0$, there exists a unique $T_\lambda = \text{argmin}_T \; C_\lambda(T)$
- Weakest link pruning: keep collapsing the internal nodes that produce the *smallest increase* in $\sum_m N_m Q_m(T)$, going from T_0 to a single node.
- Can show: the resulting sequence must contain T_λ!
- How do we set λ?

Classification trees

- What class label $\hat{y}_m$ should we assign to a leaf R_m?
- Compute fraction of examples from class c in R_m:

$$\hat{p}_{m,c} = \frac{1}{N_m} \sum_{i \in \mathcal{I}_m} \mathbb{I}[y_i = c]$$

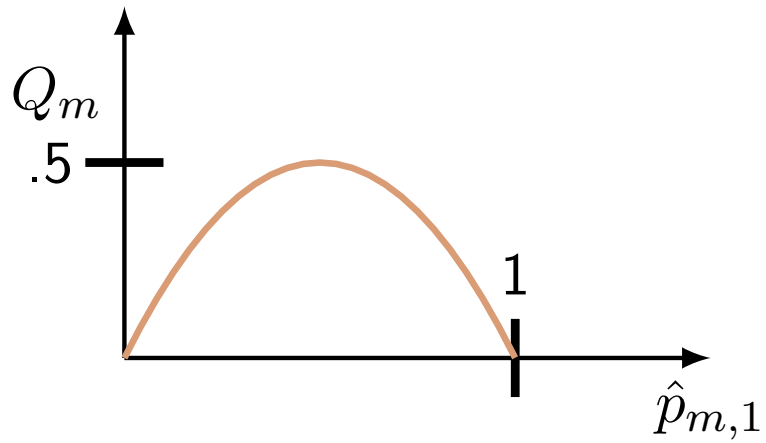
- Want to minimize 0/1 loss, per leaf:

$$\operatorname*{argmin}_{\hat{y}_m} \sum_{i \in \mathcal{I}_m} \mathbb{I}[y_i \neq \hat{y}_m]$$

- Solution: $\hat{y}_m = \operatorname{argmax}_c \hat{p}_{m,c}$

Classification trees: leaf impurity

- In regression trees, squared error is a measure of "leaf impurity"
- In classification, we have a few choices.
- Gini index ("Gini impurity") of leaf m in tree T:



$$Q_m(T) = \sum_{c=1}^{C} \hat{p}_{m,c}(1 - \hat{p}_{m,c})$$

(plot for two classes)

- An alternative Q_m: misclassification rate
- Common practice: use Gini to grow the tree, misclassification rate to prune.

Trees: summary

$$C_\lambda(T) \;=\; \lambda|T| + \sum_{m=1}^{|T|} N_m Q_m(T)$$

- This approach to regularized tree building is called CART (classification and regression trees); other methods exist
- Important advantages:

Trees: summary

$$C_\lambda(T) \;=\; \lambda|T| + \sum_{m=1}^{|T|} N_m Q_m(T)$$

- This approach to regularized tree building is called CART (classification and regression trees); other methods exist
- Important advantages:
  interpretable,
  can deal with non-numerical features naturally,
  naturally multi-class and nonlinear.

Trees: summary

$$C_\lambda(T) \;=\; \lambda|T| + \sum_{m=1}^{|T|} N_m Q_m(T)$$

- This approach to regularized tree building is called CART (classification and regression trees); other methods exist
- Important advantages:
  interpretable,
  can deal with non-numerical features naturally,
  naturally multi-class and nonlinear.
- Important limitations:

Trees: summary

$$C_\lambda(T) \;=\; \lambda|T| + \sum_{m=1}^{|T|} N_m Q_m(T)$$

- This approach to regularized tree building is called CART (classification and regression trees); other methods exist
- Important advantages:
  interpretable,
  can deal with non-numerical features naturally,
  naturally multi-class and nonlinear.
- Important limitations:
  hard splits (non-smooth regression),
  limited to axis-aligned splits,
  often high variance despite regularization.

Ensembles of models

- So far, we have considered a single model approach: train a model, apply it on test data
- What if we have multiple (different) models?
- **Ensemble** of models: given M models $\{f_1(\mathbf{x}), \ldots, f_M(\mathbf{x})\}$, at test time combine them to make a single prediction
- Simplest way to combine: average

$$\widehat{f}(\mathbf{x}) = \frac{1}{M} \sum_j f_j(\mathbf{x})$$

- When does it help?
  How do we come up with the ensemble?
  Can we do better than just average?

Combining trees

- Deep decision trees have low bias, high variance
- CART pruning may lead to poor bias/variance tradeoff
- Idea: let trees be deep (low bias), **average** many trees (low variance)

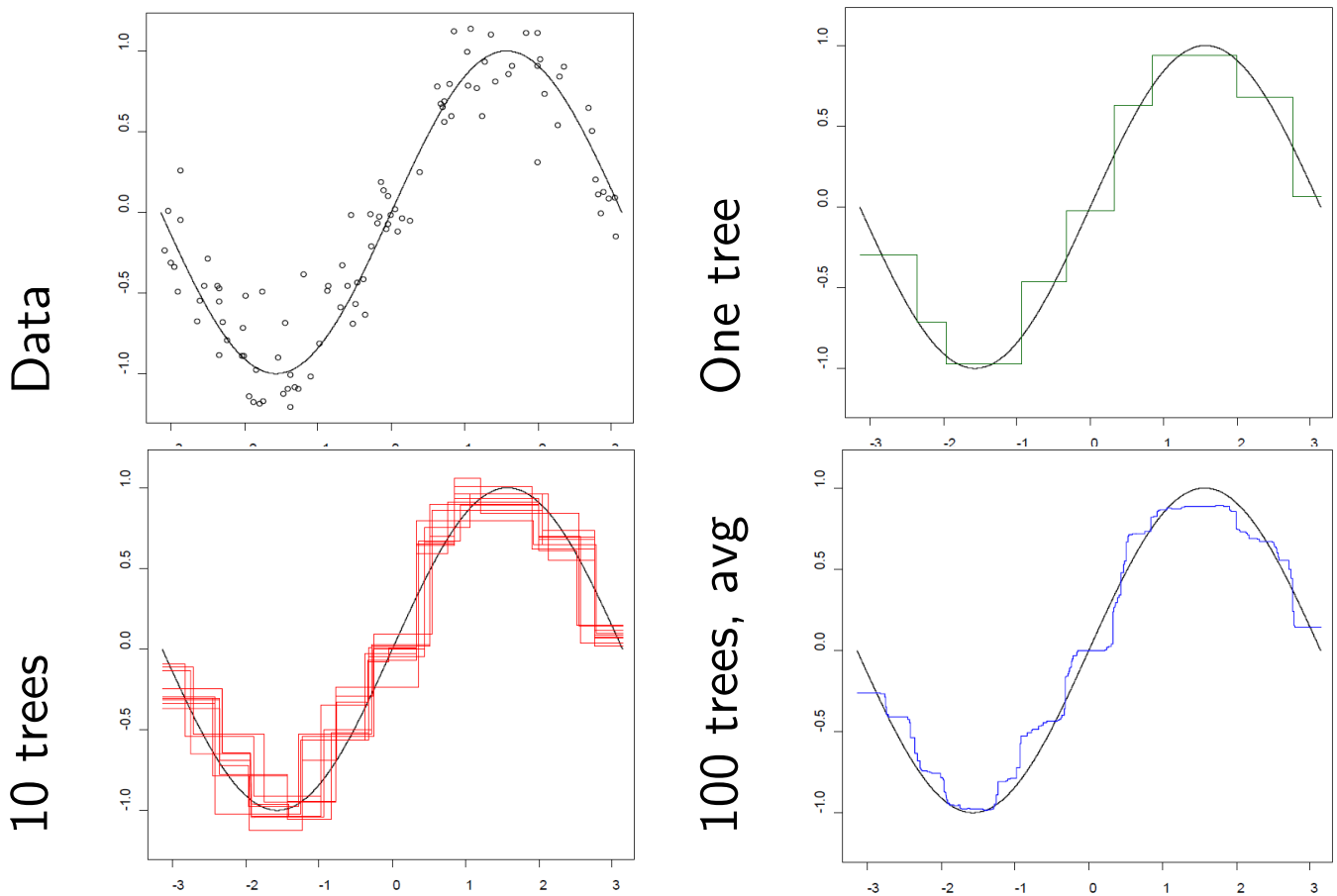


figure: A. Cutler

Combining trees

- Deep decision trees have low bias, high variance
- CART pruning may lead to poor bias/variance tradeoff
- Idea: let trees be deep (low bias), **average** many trees (low variance)

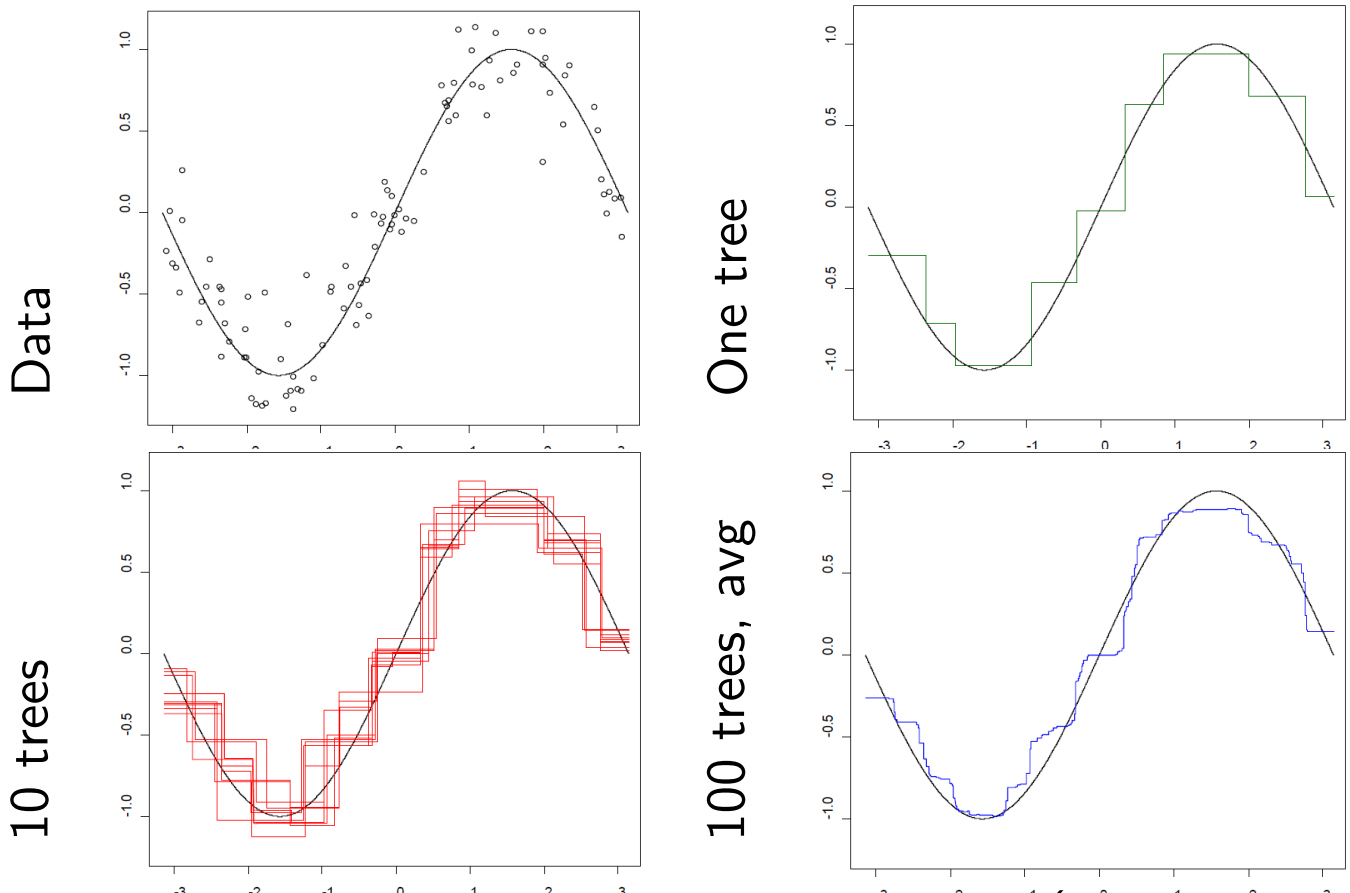


figure: A. Cutler

- We will now develop a **bagging** approach (**b**ootstrap **agg**regation)

Random forests

- In order to benefit from many trees (lower variance), we need them to be different (diverse)
- We will obtain diversity by injecting *randomness* into tree construction

Random forests

- In order to benefit from many trees (lower variance), we need them to be different (diverse)
- We will obtain diversity by injecting *randomness* into tree construction
- Two sources of randomness
- **Bootstrap** sampling: out of n training examples, sample n with replacement
  some points will appear more than once, some (approx. 37%) will not appear at all

Random forests

- In order to benefit from many trees (lower variance), we need them to be different (diverse)
- We will obtain diversity by injecting *randomness* into tree construction
- Two sources of randomness
- **Bootstrap** sampling: out of n training examples, sample n with replacement
  some points will appear more than once, some (approx. 37%) will not appear at all
- **Sampling features** in each node, when considering splits, only look at a random $m < d$ features.
- Each tree is less likely to overfit
- The "overfitting quirks" of different trees are likely to cancel out in averaging

Classification with random forests

- Tree classifier: compute scores $f_c(\mathbf{x})$, then $h(\mathbf{x}) = \text{argmax}_c\, f_c(\mathbf{x})$
- Can either average *scores*, or let trees vote

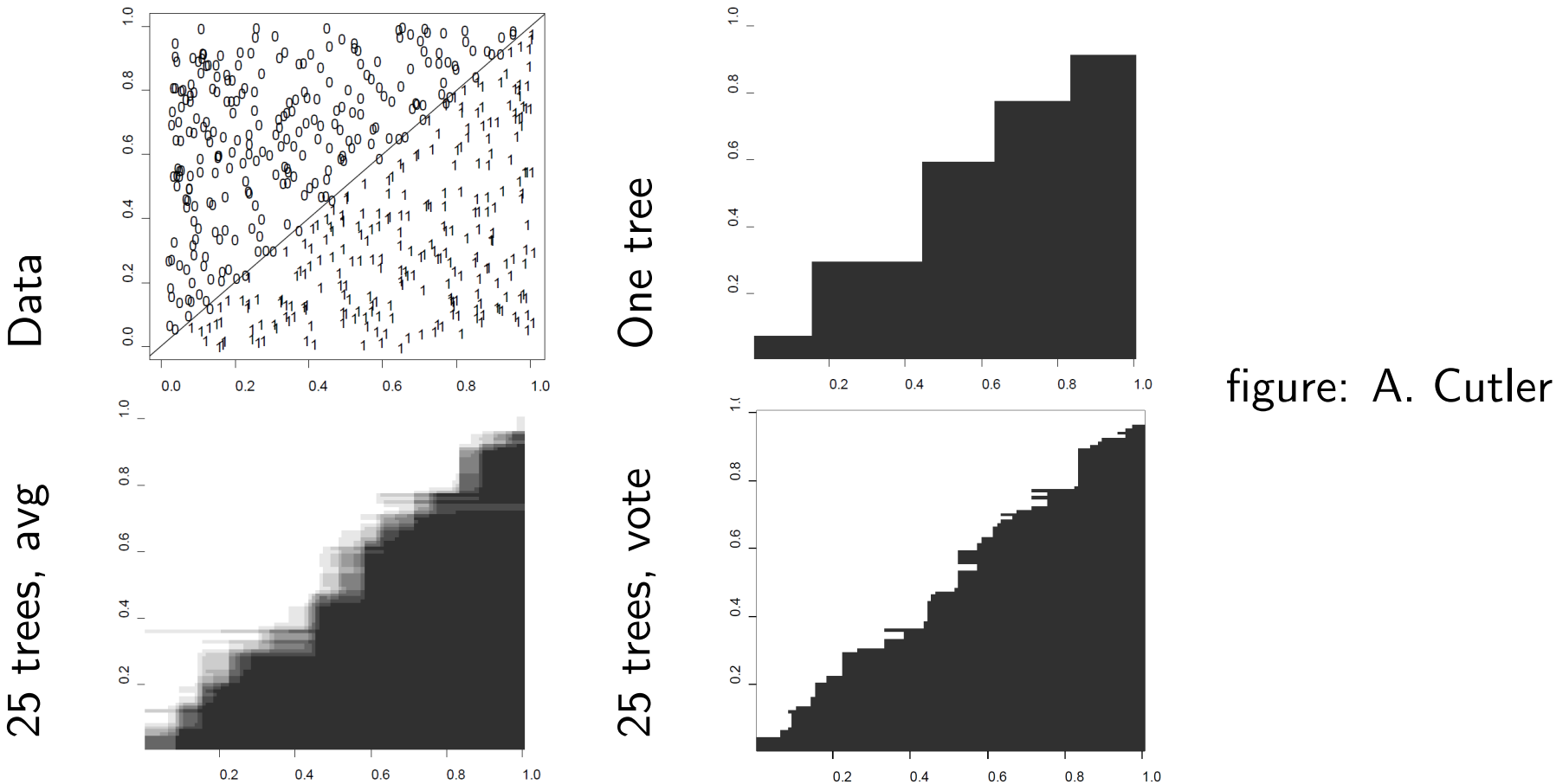


figure: A. Cutler

Random forests summary

- n data points, each with d features
- Build T trees independently (in parallel)
- For each tree:
  sample n points with replacement (or $n' < n$ without)
  Grow CART tree; in each node, only look at a random subset of $m < d$ features
  Do not prune the trees
- To make a prediction: average (regression) or vote (classification)
- Parameters to tune:

Random forests summary

- n data points, each with d features
- Build T trees independently (in parallel)
- For each tree:
  sample n points with replacement (or $n' < n$ without)
  Grow CART tree; in each node, only look at a random subset of $m < d$ features
  Do not prune the trees
- To make a prediction: average (regression) or vote (classification)
- Parameters to tune: number of trees T; feature set size m; tree depth/min number of points in leaves
- Recommended heuristic values: $m = \sqrt{d}$ for classification, $d/3$ for regression;
  min leaf size $= 5$

Bagging in general

- Instead of trees, could apply bagging to any predictor family
- Power of bagging: variance reduction through averaging
- Typically, benefit is highest with unstable, highly nonlinear predictors (e.g., trees)
- Linear predictors: no benefit from bagging
- Useful property of bagging: "out of bag" (OOB) data
  in each tree, treat the $\approx$37% of the examples that didn't make it to the sample as a kind of validation set
- While assembling trees, keep track of OOB accuracy, stop upon seeing plateau

Combining classifiers

- Classifying a point using a decision tree can be seen as a sequence of classifiers, refined as we follow the path to a leaf
- A more general formulation: combine classifiers $h_1(\mathbf{x}), \ldots, h_m(\mathbf{x})$

$$H(\mathbf{x}) \;=\; \alpha_1 h_1(\mathbf{x}) + \ldots + \alpha_m h_m(\mathbf{x})$$

- α_j is the weight of the vote assigned to classifier h_j
  - Votes should have higher weight for more reliable classifiers
- Prediction (for binary classification):

$$\hat{y}(\mathbf{x}) \;=\; \text{sign}(H(\mathbf{x}))$$

- Classifiers h_j can be simple (e.g., based on a single feature)

Greedy assembly of classifier combination

- Consider a family of classifiers $\mathcal{H}$ parametrized by θ.
- Setting θ_1: minimize the training error

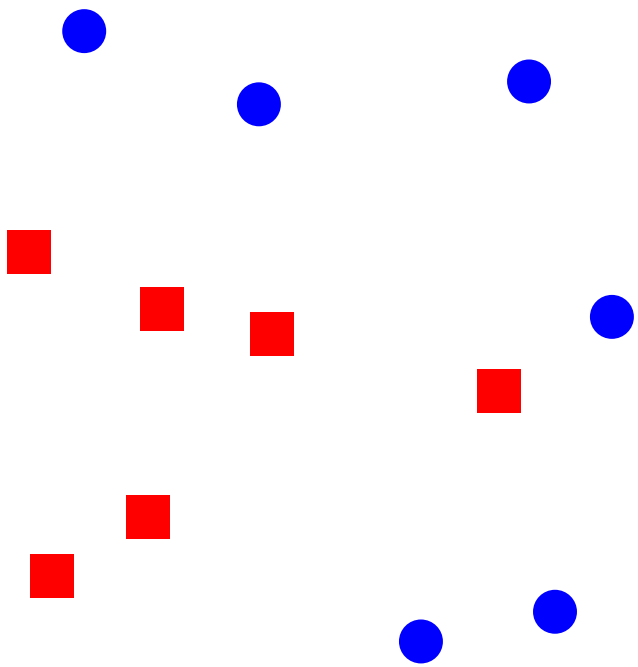
$$\sum_{i=1}^{n} L(h(\mathbf{x}_i; \theta_1),\, y_i)$$

  where L is some surrogate for the 0/1 loss.
- How do we set θ_2?
- We would like to minimize the (surrogate) loss of the combination,

$$\sum_{i=1}^{n} L(H(\mathbf{x}_i),\, y_i)$$

  where $H(\mathbf{x}) \;=\; \operatorname{sign}\left(\alpha_1 h(\mathbf{x}; \theta_1) + \alpha_2 h(\mathbf{x}; \theta_2)\right)$

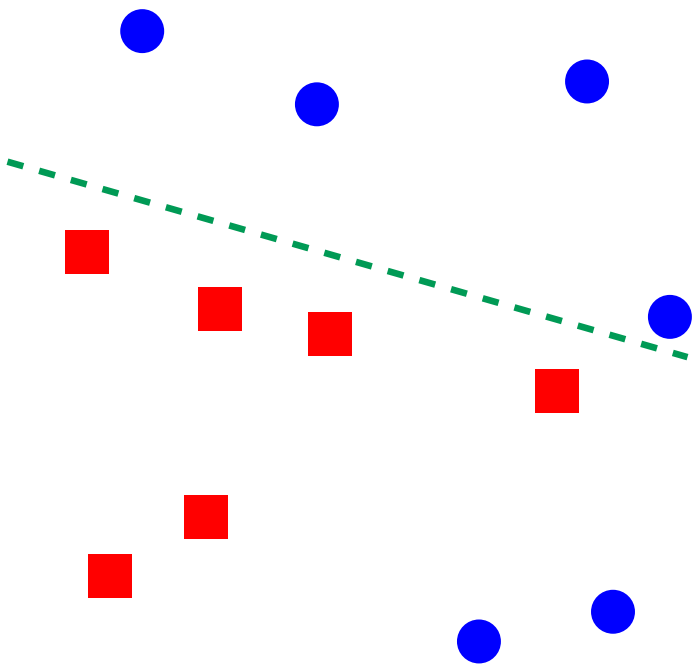
AdaBoost: intuition



Greedy alg. for $m = 1, \ldots, M$

- Maintain weights $W_i^{(m)}$, initially all $1/n$
- Pick a weak classifier h_m minimizing error ϵ_m weighted by $W^{(m-1)}$
- Set $\alpha_m = \frac{1}{2} \log \frac{1-\epsilon_m}{\epsilon_m}$
- Update weights $W_i^{(m)}$ based on mistakes of h_m and on α_m
- Final (strong) classifier $\text{sign}\left(\sum_m \alpha_m h_m(\cdot)\right)$

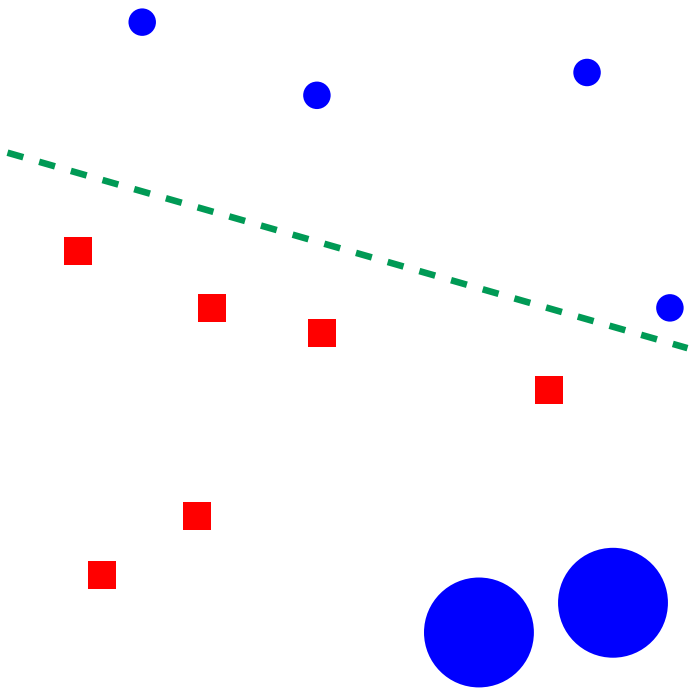
AdaBoost: intuition



Greedy alg. for $m = 1, \ldots, M$

- Maintain weights $W_i^{(m)}$, initially all $1/n$
- Pick a weak classifier h_m minimizing error ϵ_m weighted by $W^{(m-1)}$
- Set $\alpha_m = \frac{1}{2} \log \frac{1-\epsilon_m}{\epsilon_m}$
- Update weights $W_i^{(m)}$ based on mistakes of h_m and on α_m
- Final (strong) classifier $\text{sign}\left(\sum_m \alpha_m h_m(\cdot)\right)$

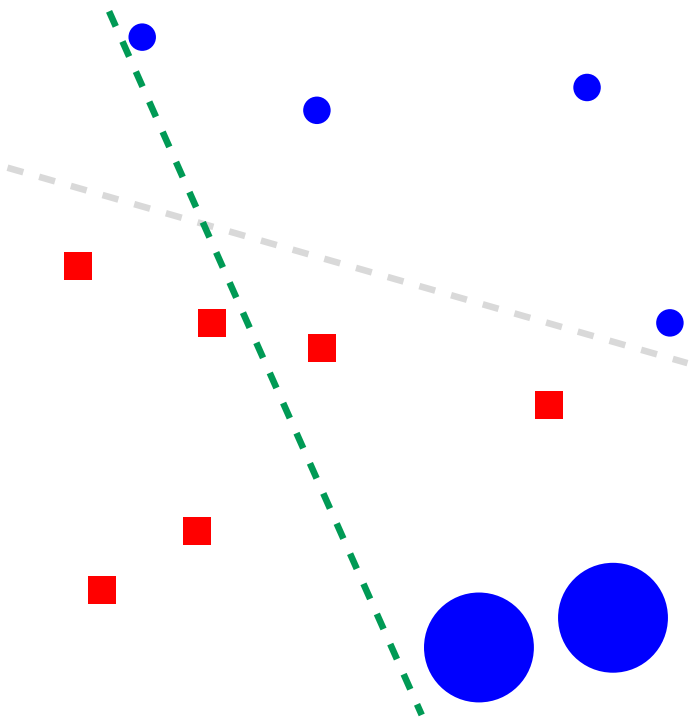
AdaBoost: intuition



Greedy alg. for $m = 1, \ldots, M$

- Maintain weights $W_i^{(m)}$, initially all $1/n$
- Pick a weak classifier h_m minimizing error ϵ_m weighted by $W^{(m-1)}$
- Set $\alpha_m = \frac{1}{2} \log \frac{1-\epsilon_m}{\epsilon_m}$
- Update weights $W_i^{(m)}$ based on mistakes of h_m and on α_m
- Final (strong) classifier $\text{sign}\left(\sum_m \alpha_m h_m(\cdot)\right)$

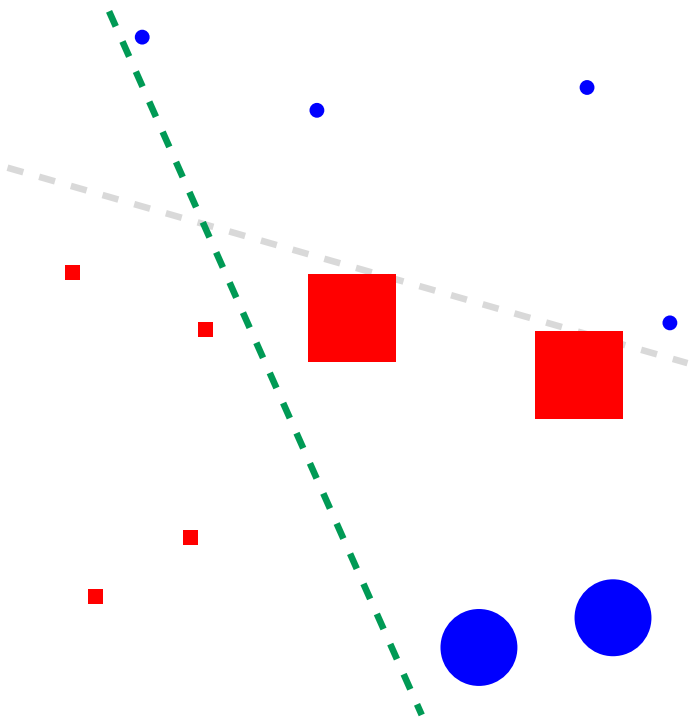
AdaBoost: intuition



Greedy alg. for $m = 1, \ldots, M$

- Maintain weights $W_i^{(m)}$, initially all $1/n$
- Pick a weak classifier h_m minimizing error ϵ_m weighted by $W^{(m-1)}$
- Set $\alpha_m = \frac{1}{2} \log \frac{1-\epsilon_m}{\epsilon_m}$
- Update weights $W_i^{(m)}$ based on mistakes of h_m and on α_m
- Final (strong) classifier $\text{sign}\left(\sum_m \alpha_m h_m(\cdot)\right)$

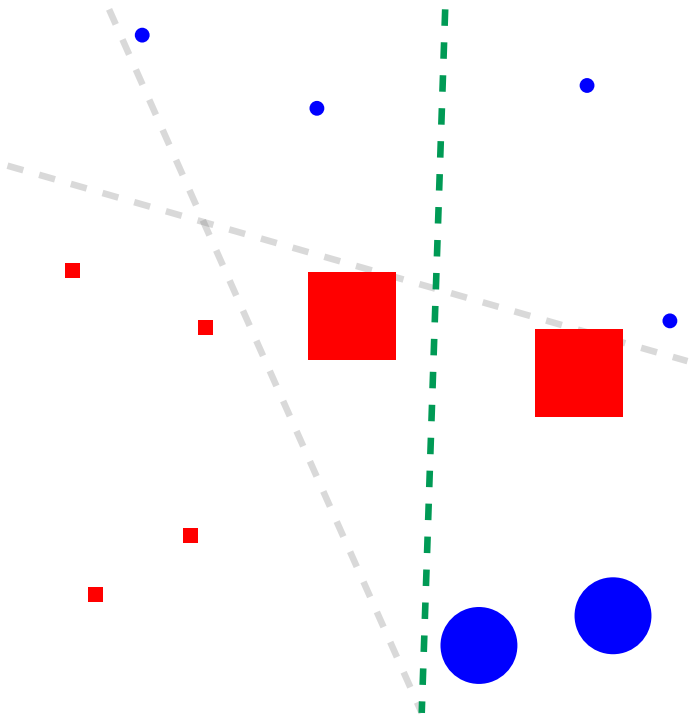
AdaBoost: intuition



Greedy alg. for $m = 1, \ldots, M$

- Maintain weights $W_i^{(m)}$, initially all $1/n$
- Pick a weak classifier h_m minimizing error ϵ_m weighted by $W^{(m-1)}$
- Set $\alpha_m = \frac{1}{2} \log \frac{1-\epsilon_m}{\epsilon_m}$
- Update weights $W_i^{(m)}$ based on mistakes of h_m and on α_m
- Final (strong) classifier $\text{sign}\left(\sum_m \alpha_m h_m(\cdot)\right)$

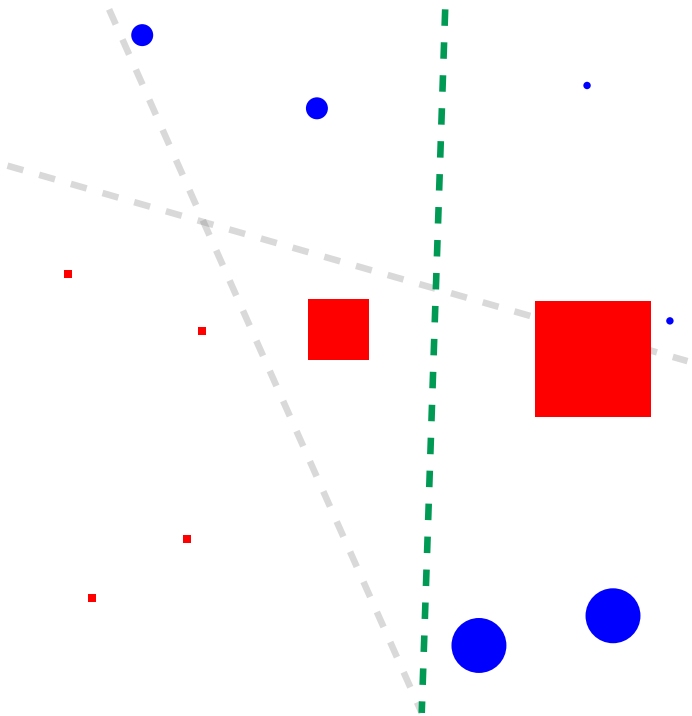
AdaBoost: intuition



Greedy alg. for $m = 1, \ldots, M$

- Maintain weights $W_i^{(m)}$, initially all $1/n$
- Pick a weak classifier h_m minimizing error ϵ_m weighted by $W^{(m-1)}$
- Set $\alpha_m = \frac{1}{2} \log \frac{1-\epsilon_m}{\epsilon_m}$
- Update weights $W_i^{(m)}$ based on mistakes of h_m and on α_m
- Final (strong) classifier $\text{sign}\left(\sum_m \alpha_m h_m(\cdot)\right)$

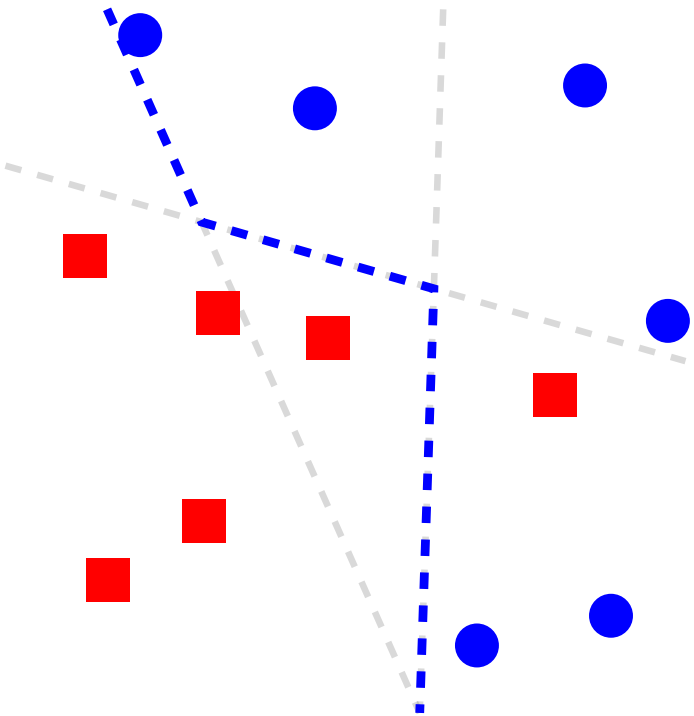
AdaBoost: intuition



Greedy alg. for $m = 1, \ldots, M$

- Maintain weights $W_i^{(m)}$, initially all $1/n$
- Pick a weak classifier h_m minimizing error ϵ_m weighted by $W^{(m-1)}$
- Set $\alpha_m = \frac{1}{2} \log \frac{1-\epsilon_m}{\epsilon_m}$
- Update weights $W_i^{(m)}$ based on mistakes of h_m and on α_m
- Final (strong) classifier $\text{sign}\left(\sum_m \alpha_m h_m(\cdot)\right)$

AdaBoost: intuition



Greedy alg. for $m = 1, \ldots, M$

- Maintain weights $W_i^{(m)}$, initially all $1/n$
- Pick a weak classifier h_m minimizing error ϵ_m weighted by $W^{(m-1)}$
- Set $\alpha_m = \frac{1}{2} \log \frac{1-\epsilon_m}{\epsilon_m}$
- Update weights $W_i^{(m)}$ based on mistakes of h_m and on α_m
- Final (strong) classifier $\text{sign}\left(\sum_m \alpha_m h_m(\cdot)\right)$