

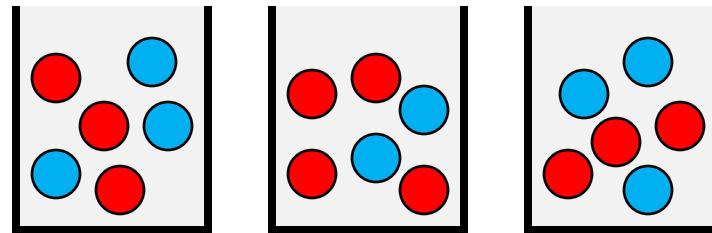
COMP 6321 Machine Learning

Bagging & Boosting

Computer Science & Software Engineering
Concordia University, Fall 2020



Bagging



Bagging

- Bootstrap aggregation (B-agg-ing => Bagging) is a simple technique for combining “unstable” models and making them more “stable”
- *Unstable*: a small change in training data can lead to a large change in predictions.
 - Decision trees, high-degree polynomials, ...
 - Hard-margin SVM unstable inside margin, stable outside
- Random forests are just “bagged decision trees”

Bagging predictors

[L Breiman - Machine learning, 1996 - Springer](#)

Bagging predictors is a method for generating multiple versions of a predictor and using these to get an aggregated predictor. The aggregation averages over the versions when predicting a numerical outcome and does a plurality vote when predicting a class. The ...

Bootstrap sampling

- **Idea:** Train a model that is insensitive to dropping or duplicating individual training examples
 - After all, the observed training data is just *one* possible sampling from the true data distribution.
- Remember considerations for random forests?...
 - Should we *perturb* the data? Possible, but hard to know what perturbations help or ‘corrupt’ in harmful ways.
 - Should we *subsample* the data? Reasonable! But each new training set is smaller than N , and by how much?
 - **Should we *re-sample* the data with replacement?** Good! Some (x_i, y_i) get duplicated, but no parameters to choose.

This is called “bootstrap sampling” and each re-sampling is called a “bootstrap sample”

Proposed for statistical analyses
by Brad Efron in 1979-1982

Sampling with replacement in Python

- See slide from prev lecture on how to sample data for random forests:

```
print(X)
print(y)
```

```
[[0.  0.]
 [1.  1.]
 [2.  2.]
 [3.  3.]
 [4.  4.]]
[0 1 2 3 4]
```

```
sample_indices = np.random.randint(0, N, N)
print(sample_indices)
```

```
[2 4 2 1 3]
```

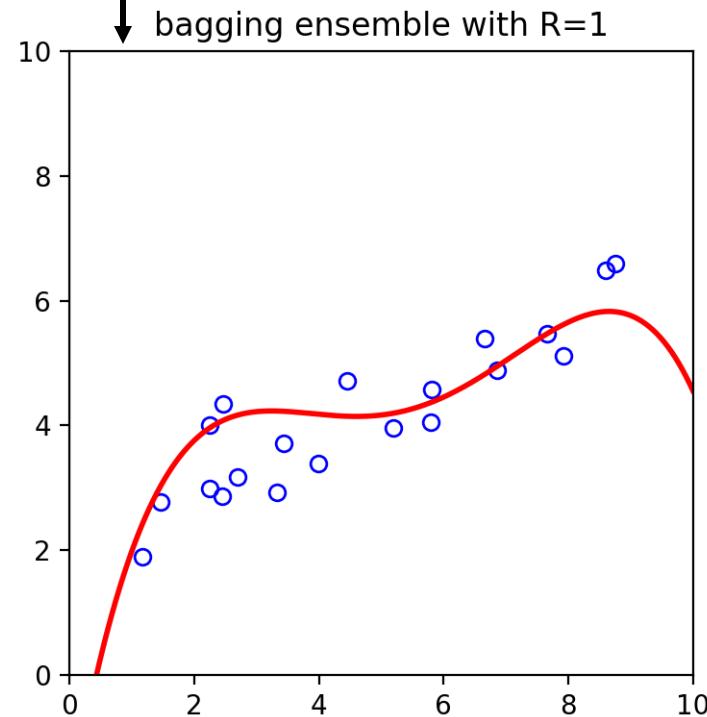
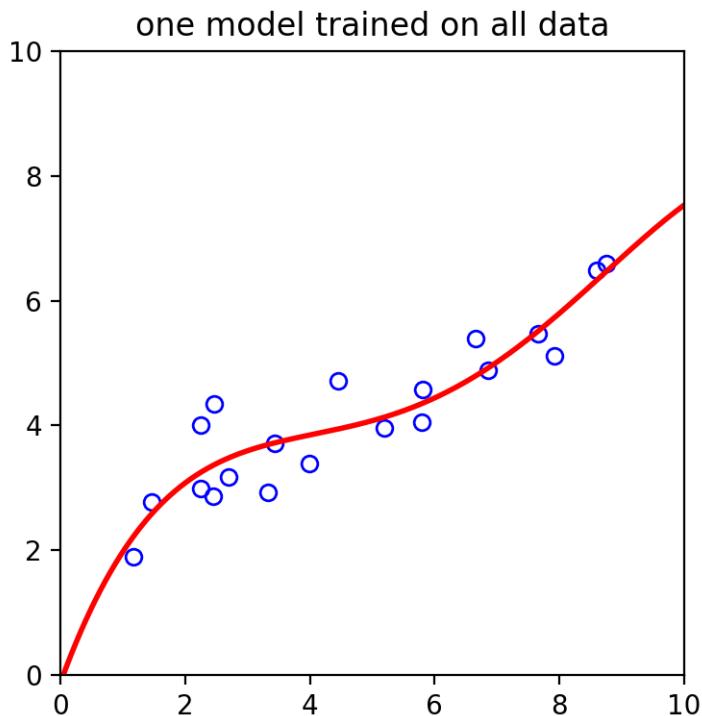
```
X_resampled = X[sample_indices]
y_resampled = y[sample_indices]
print(X_resampled)
print(y_resampled)
```

```
[[2.  2.]
 [4.  4.]
 [2.  2.]
 [1.  1.]
 [3.  3.]]
[2 4 2 1 3]
```

- Bootstrap is a general idea from statistics, *not* specific to random forests!

Bagging Example 1

One model trained on a *bootstrap sample* of the training data.

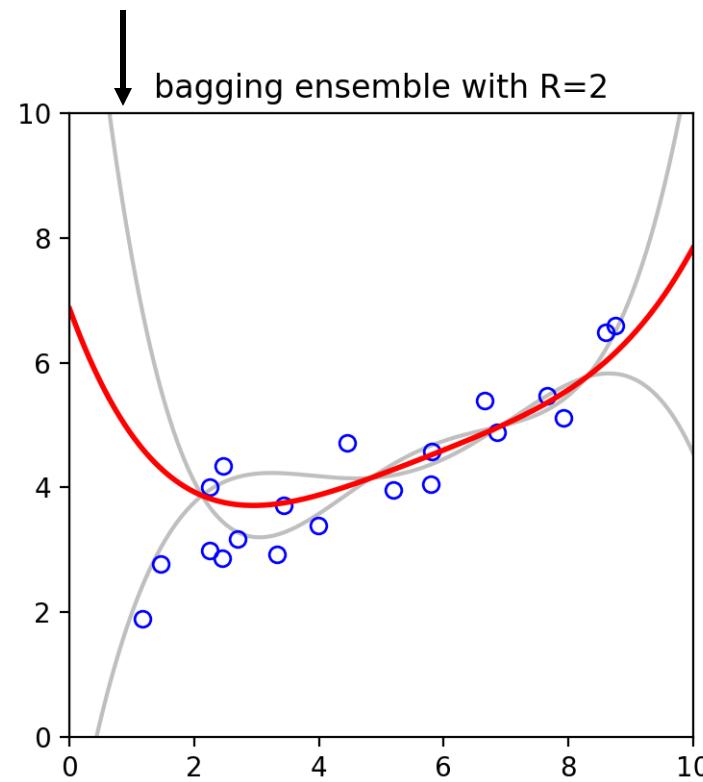
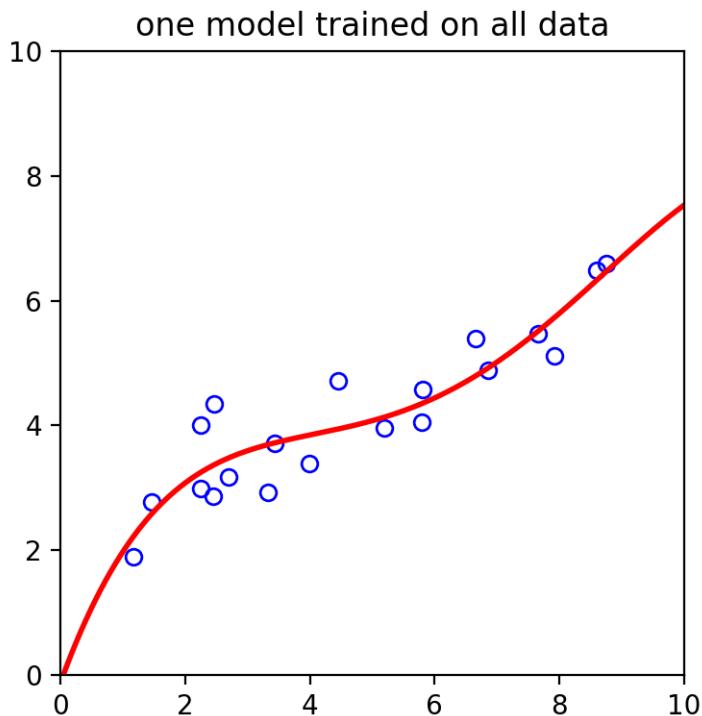


Fit polynomial to this data by using linear regression on feature transformation

$$\phi(x) = [1 \quad x \quad x^2 \quad x^3 \quad x^4]^T$$

Bagging Example 1

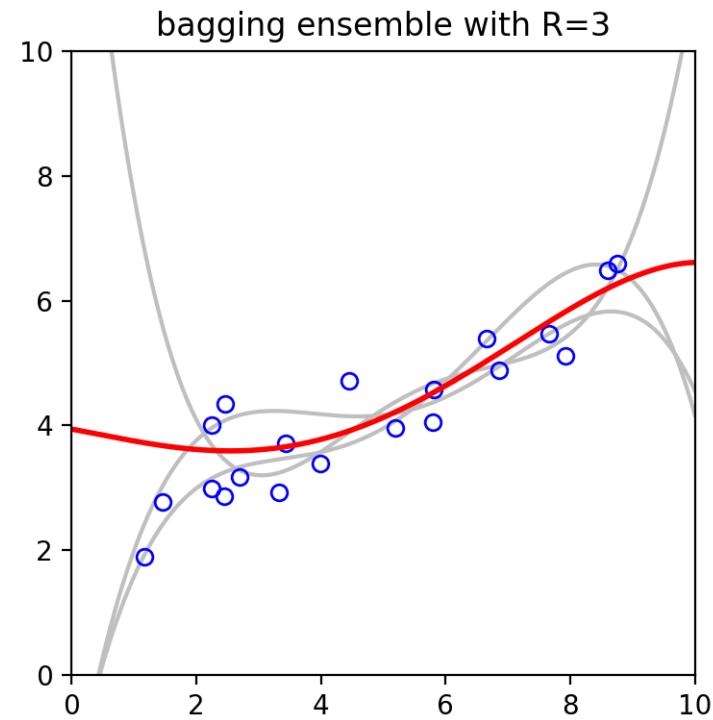
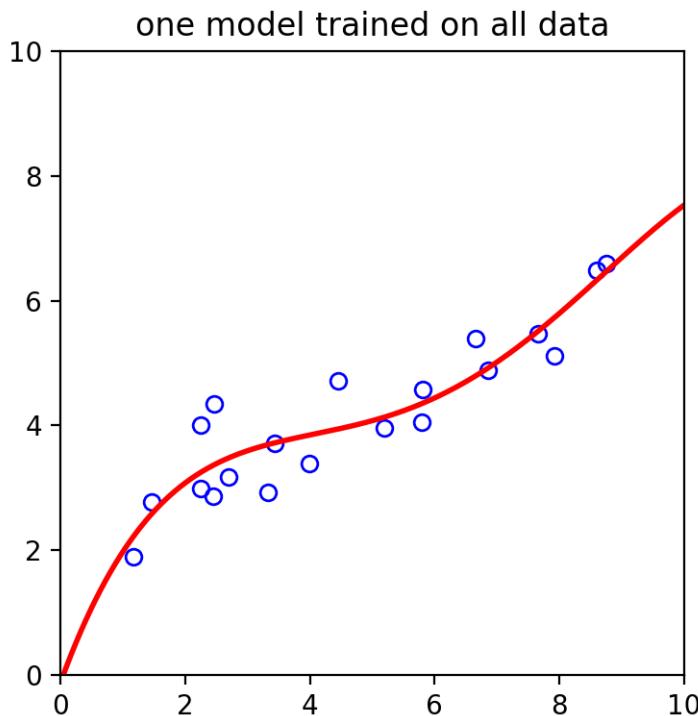
Average of $R=2$ models, each trained on a different bootstrap sample of the training data.



Fit polynomial to this data by using linear regression on feature transformation

$$\phi(x) = [1 \quad x \quad x^2 \quad x^3 \quad x^4]^T$$

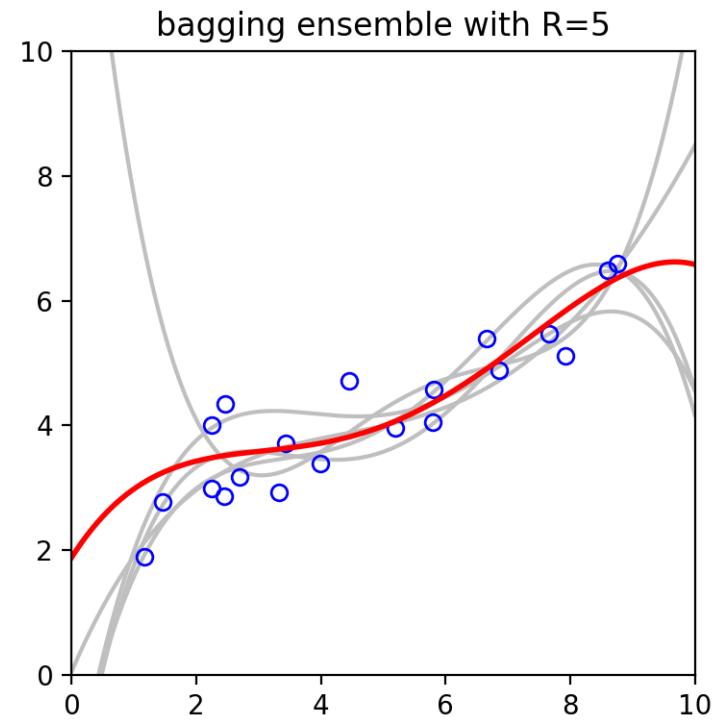
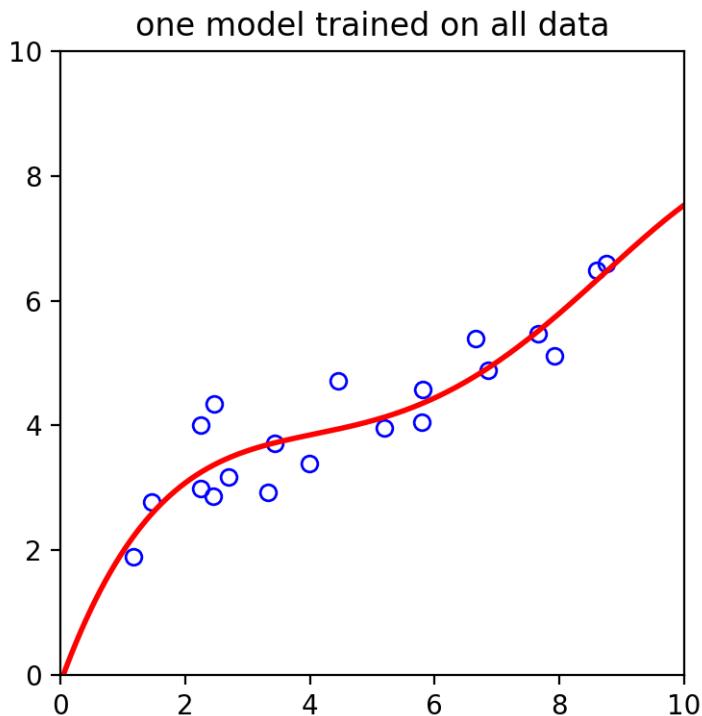
Bagging Example 1



Fit polynomial to this data by using linear regression on feature transformation

$$\phi(x) = [1 \quad x \quad x^2 \quad x^3 \quad x^4]^T$$

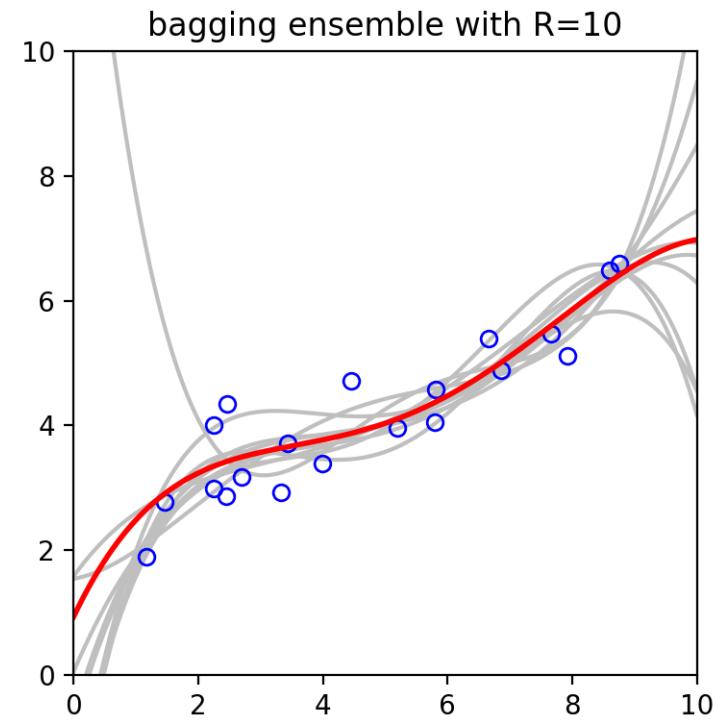
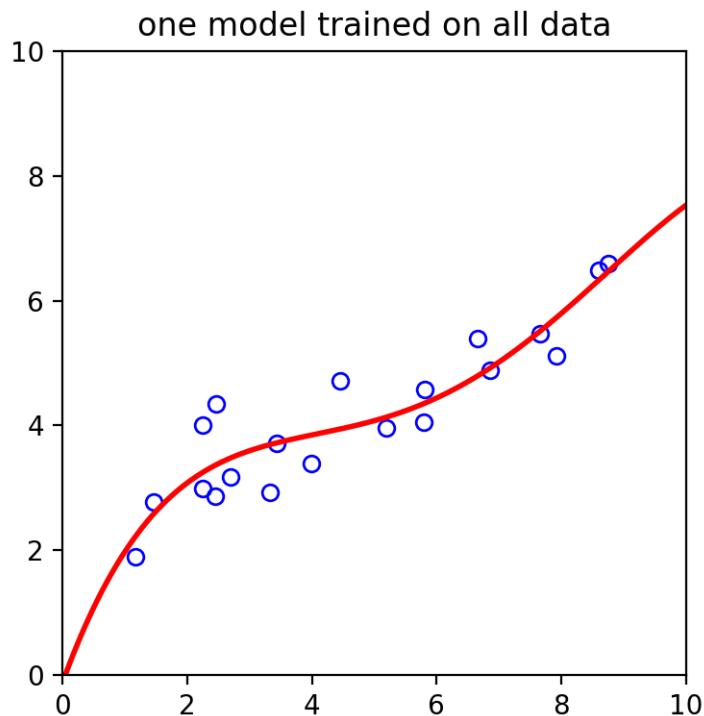
Bagging Example 1



Fit polynomial to this data by using linear regression on feature transformation

$$\phi(x) = [1 \quad x \quad x^2 \quad x^3 \quad x^4]^T$$

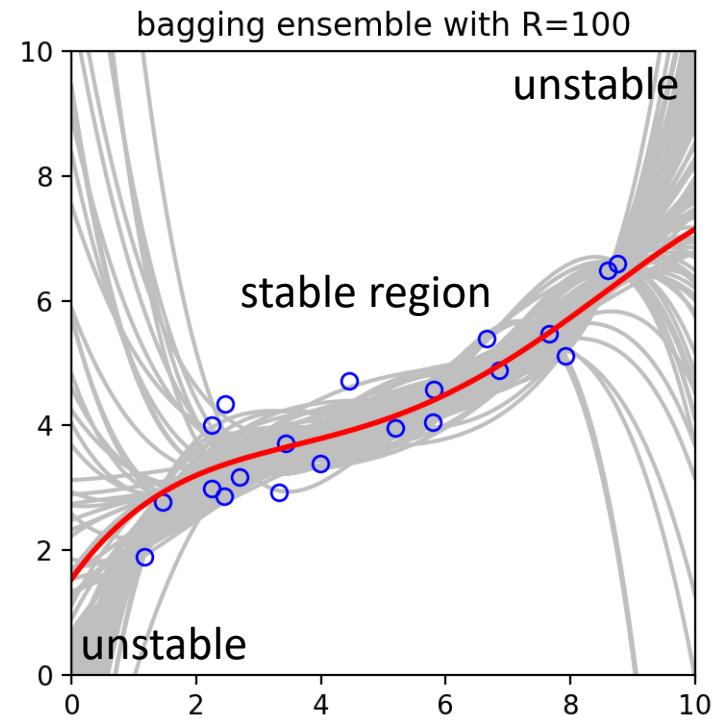
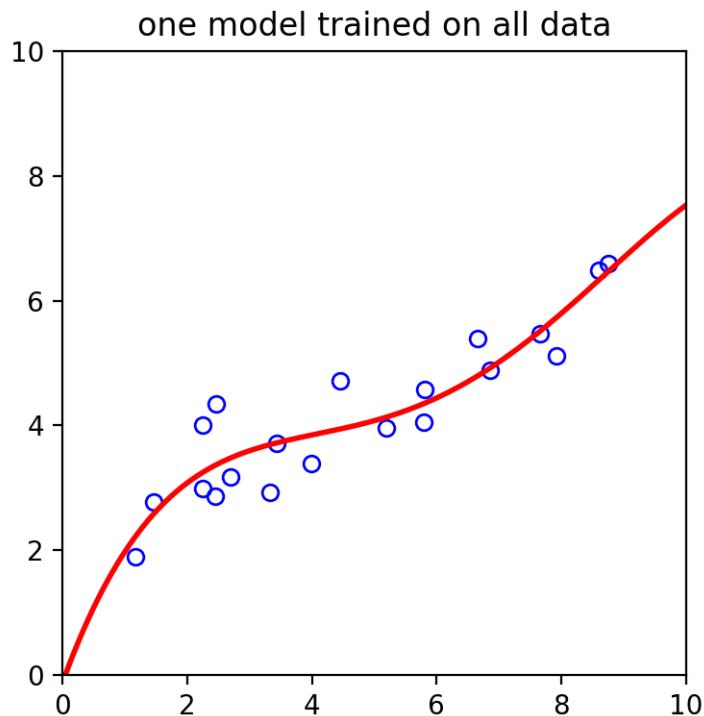
Bagging Example 1



Fit polynomial to this data by using linear regression on feature transformation

$$\phi(x) = [1 \quad x \quad x^2 \quad x^3 \quad x^4]^T$$

Bagging Example 1

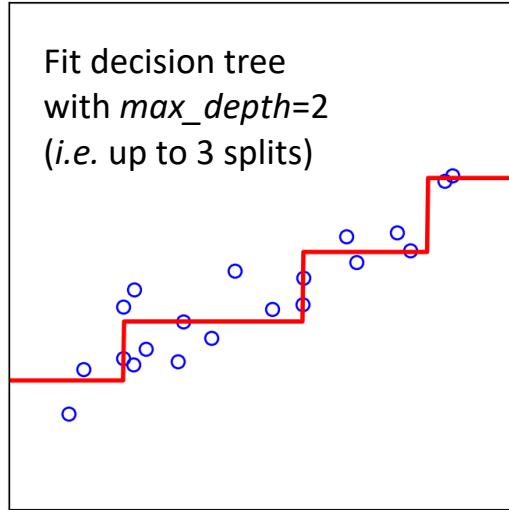


Fit polynomial to this data by using linear regression on feature transformation

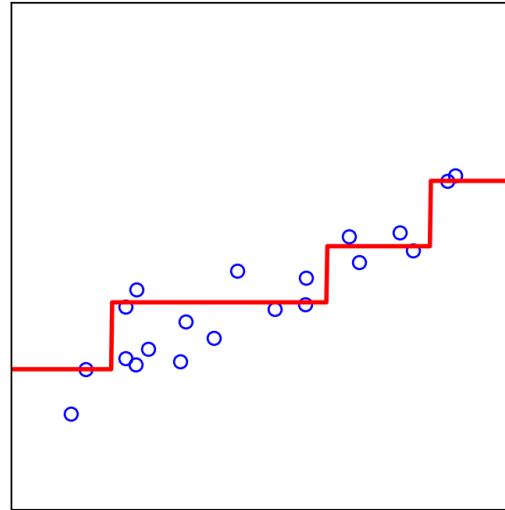
$$\phi(x) = [1 \quad x \quad x^2 \quad x^3 \quad x^4]^T$$

Example 2: Decision Tree Regression

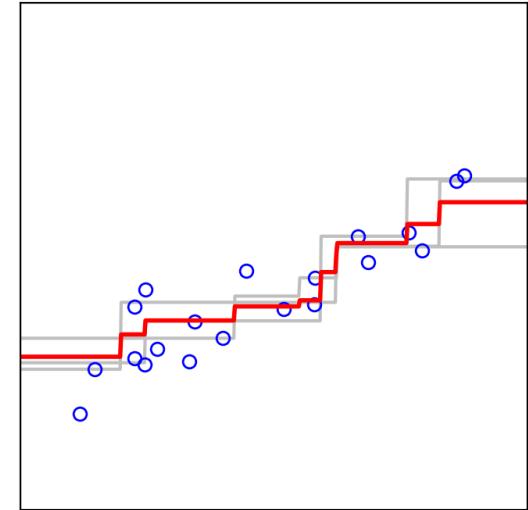
one model trained on all data



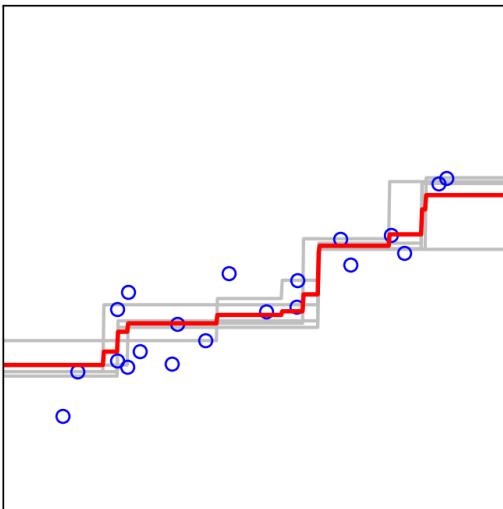
bagging ensemble with R=1



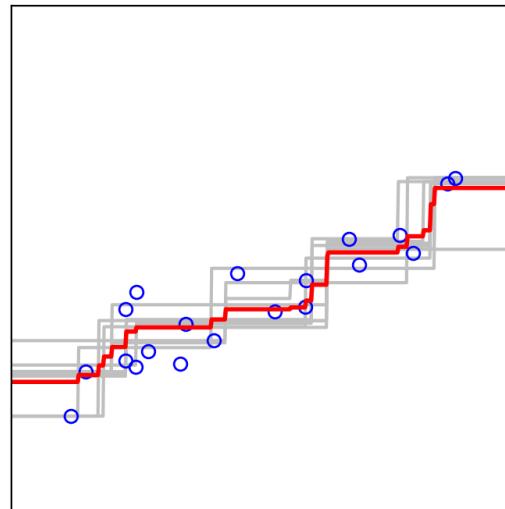
bagging ensemble with R=3



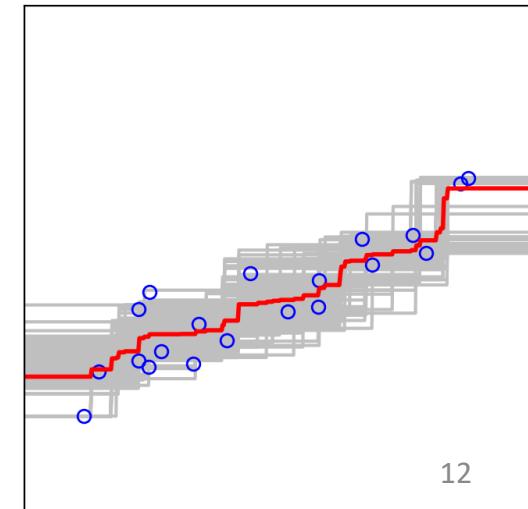
bagging ensemble with R=5



bagging ensemble with R=10

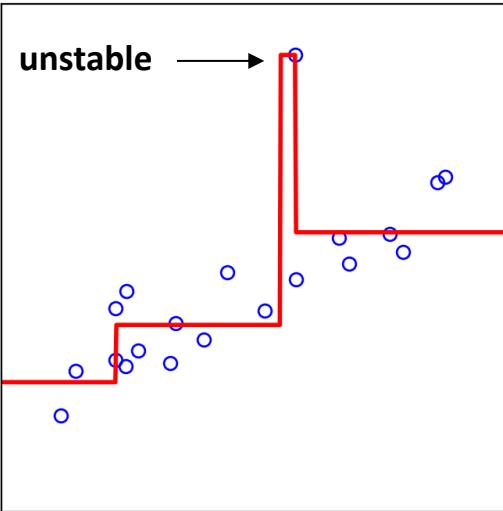


bagging ensemble with R=100

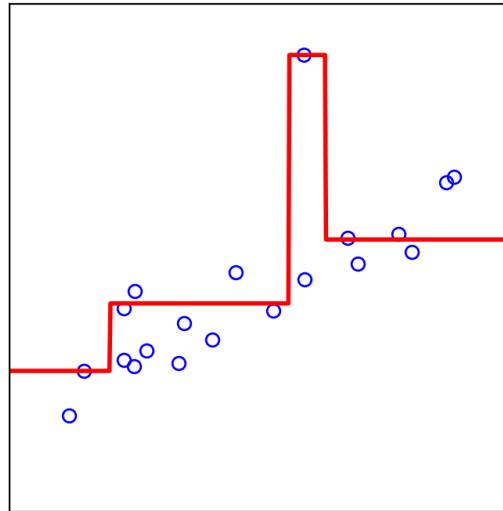


Example 2: Decision Tree Regression

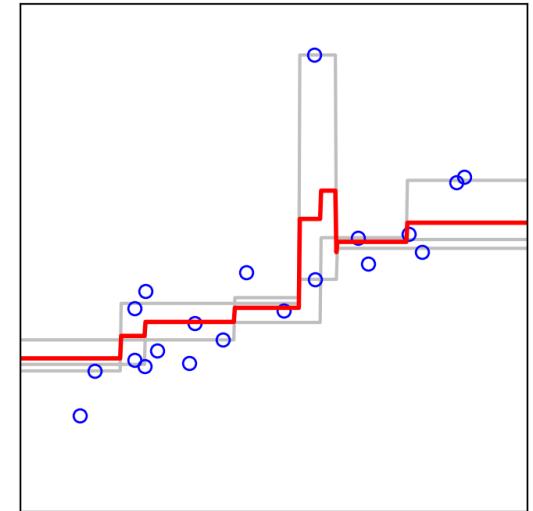
one model trained on all data



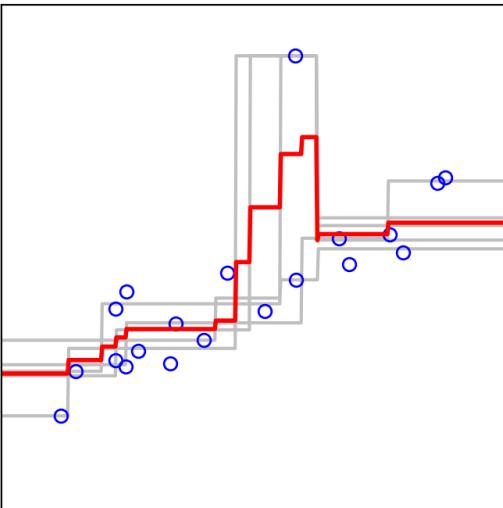
bagging ensemble with R=1



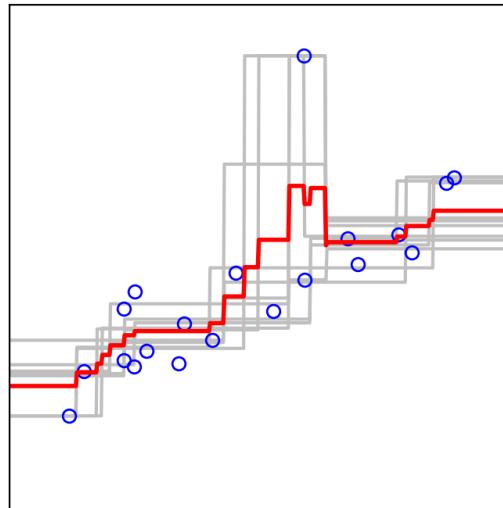
bagging ensemble with R=3



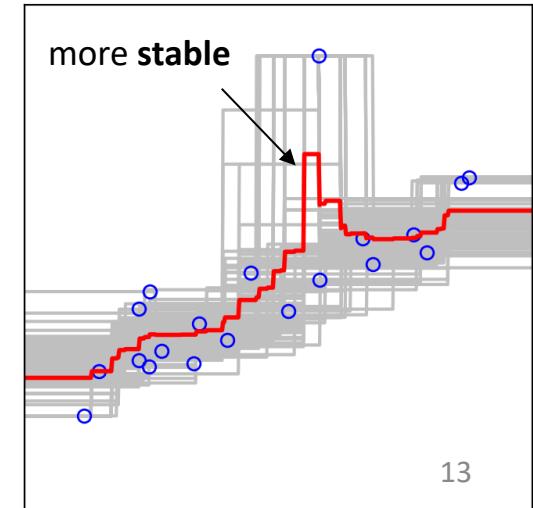
bagging ensemble with R=5



bagging ensemble with R=10

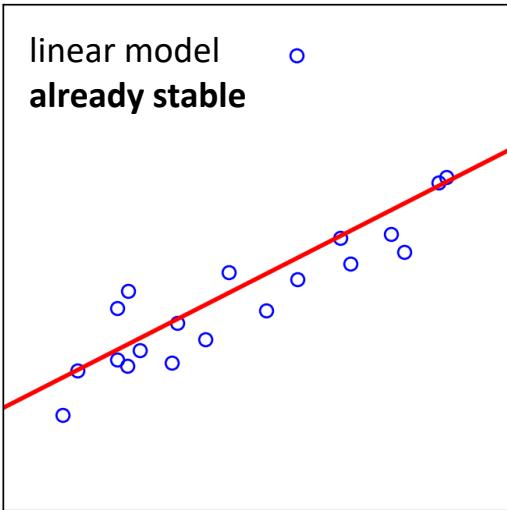


bagging ensemble with R=100

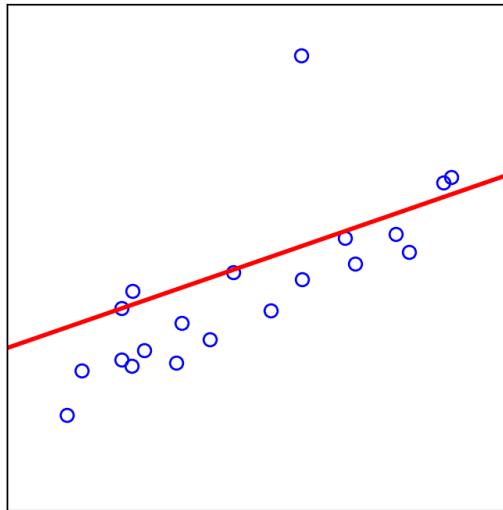


Example 3: Linear Regression

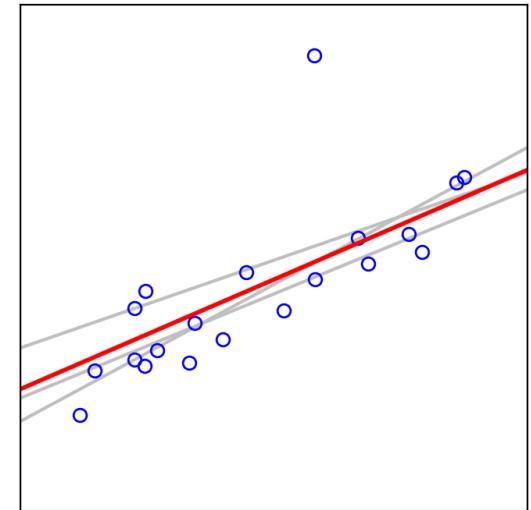
one model trained on all data



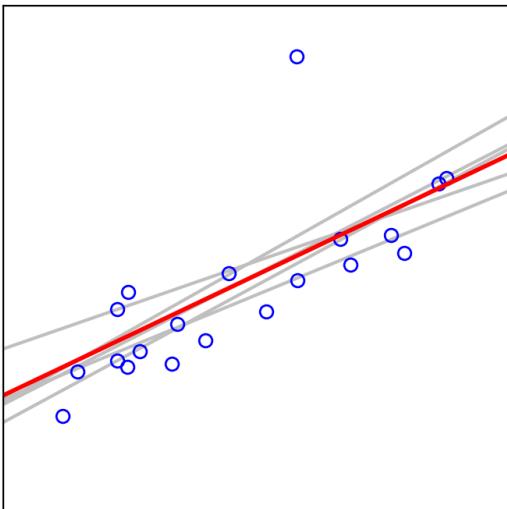
bagging ensemble with R=1



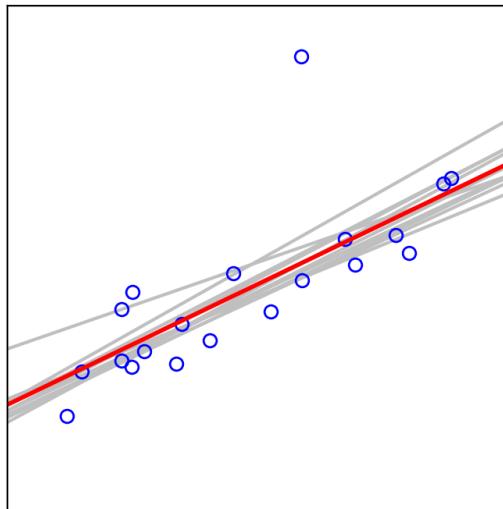
bagging ensemble with R=3



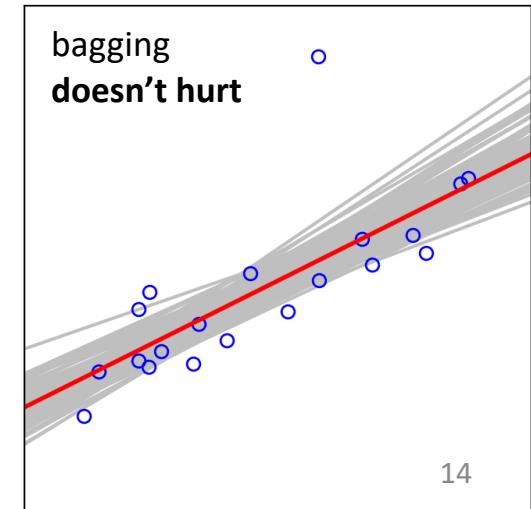
bagging ensemble with R=5



bagging ensemble with R=10



bagging ensemble with R=100



Scikit-learn makes bagging easy!

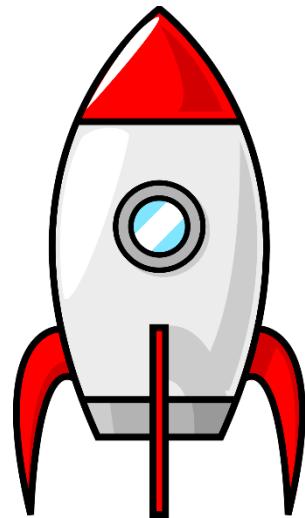
```
# Create a prototype of the kind of model we want BaggingRegressor to fit
tree = DecisionTreeRegressor(max_depth=2)

# Create a model with 10 separate DecisionTreeRegressor instances inside
model = BaggingRegressor(n_estimators=10,
                         base_estimator=tree)

# Train 10 separate decision trees, each on a different bootstrap sample of rows of X, y.
model.fit(X, y)
```

```
BaggingRegressor(base_estimator=DecisionTreeRegressor(criterion='mse',
                                                       max_depth=2,
                                                       max_features=None,
                                                       max_leaf_nodes=None,
                                                       min_impurity_decrease=0.0,
                                                       min_impurity_split=None,
                                                       min_samples_leaf=1,
                                                       min_samples_split=2,
                                                       min_weight_fraction_leaf=0.0,
                                                       presort=False,
                                                       random_state=None,
                                                       splitter='best'),
                 bootstrap=True, bootstrap_features=False, max_features=1.0,
                 max_samples=1.0, n_estimators=10, n_jobs=None, oob_score=False,
                 random_state=None, verbose=0, warm_start=False)
```

Boosting



Bagging vs Boosting

- In *bagging*, we train models $\{1, \dots, R\}$ *independently*
- In *boosting*, we train models $\{1, \dots, R\}$ *sequentially*
- **General idea of boosting:** If we train the models sequentially, then we can train the r^{th} model to *fix the errors* that models $\{1, \dots, r - 1\}$ are currently making on the training set, “boosting” performance.
- **AdaBoost:** The r^{th} model focuses on the data (x_i, y_i) that are still ‘hard’ after combining $\{1, \dots, r - 1\}$
- **Gradient boosting:** The r^{th} model learns to fix the residual errors after combining $\{1, \dots, r - 1\}$

General idea of boosting

We can keep increasing R to aim for higher training accuracy, unlike bagging.

Single model



train one model $y(x)$

Bagging



train R models
independently



\dots



Can be done in parallel.

Models that are *unstable* benefit the most.

Models don't have to be 'weak', to benefit, *e.g.* helps stabilize decision trees even if deep.

Boosting



train R models
sequentially



training errors



\dots



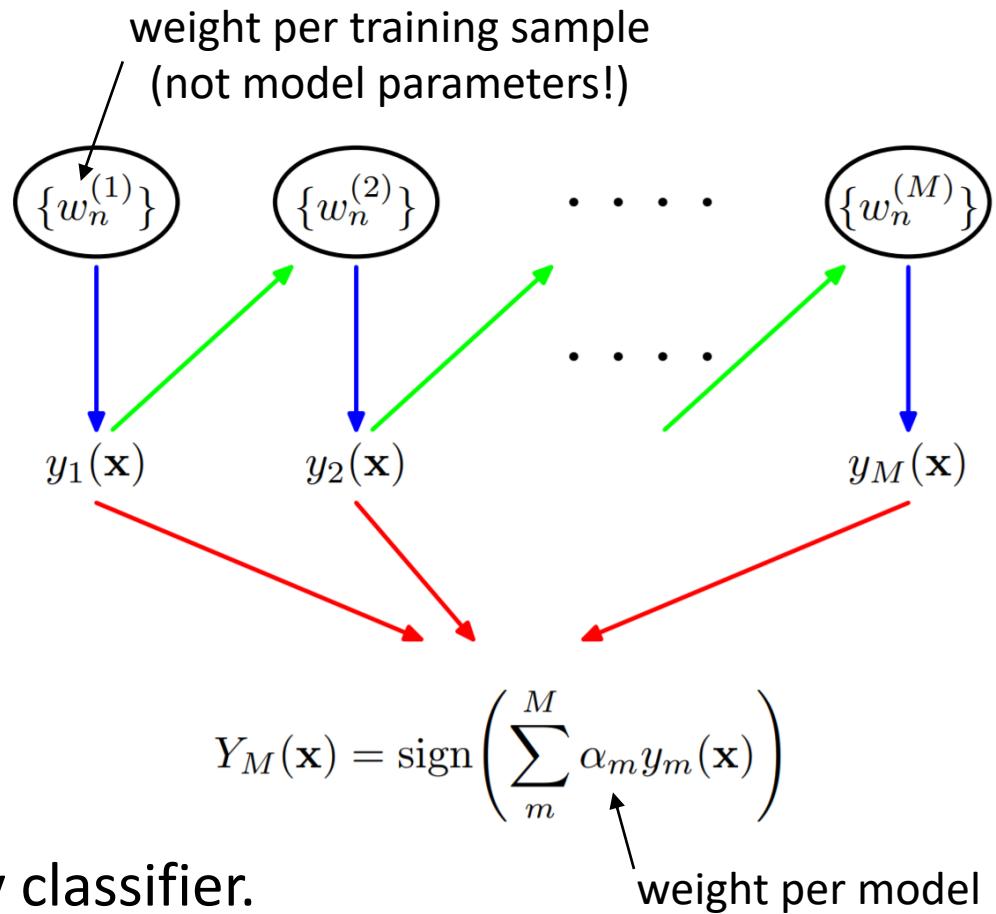
'Weak' models benefit the most; strong models already have low training error.

Subsequent models are trained to focus on *fixing errors* that remain after combining previous models.

Depiction of boosting from Bishop

Figure 14.1

Schematic illustration of the boosting framework. Each base classifier $y_m(\mathbf{x})$ is trained on a weighted form of the training set (blue arrows) in which the weights $w_n^{(m)}$ depend on the performance of the previous base classifier $y_{m-1}(\mathbf{x})$ (green arrows). Once all base classifiers have been trained, they are combined to give the final classifier $Y_M(\mathbf{x})$ (red arrows).



This depicts “boosting” a binary classifier.
(Boosting can be generalized to multi-class too.)

AdaBoost

- Adaptive boosting (AdaBoost) is a classic algorithm
- Tries to find weights $\{\alpha_1, \dots, \alpha_R\}$ that can combine ‘weak’ models into a ‘strong’ one:

$$y(\mathbf{x}) = \sum_{r=1}^R \underbrace{\alpha_r}_{\text{collectively strong}} \underbrace{y_r(\mathbf{x})}_{\text{individually weak}}$$

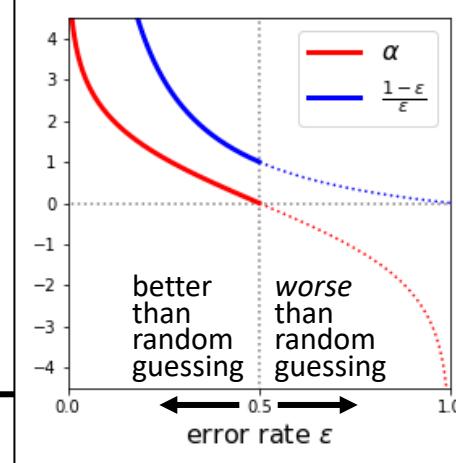


Notice that bagging assumes $\alpha_r = \frac{1}{R}$

- Works by ‘adaptively’ re-weighting the training cases so that later models focus on harder examples.
 - Maintains per-training sample weights $\{w_1, \dots, w_N\}$
 - At training iteration r , sample weight w_i captures how poorly the combination of models $\{1, \dots, r - 1\}$ was at predicting the target for training sample \mathbf{x}_i .
 - The r^{th} model then knows which samples to “focus” on.

AdaBoost algorithm (binary)

$\mathcal{D} = \{(\mathbf{x}_1, t_1), \dots, (\mathbf{x}_N, t_N)\}$ where $t_i \in \{-1, +1\}$



initialize $w_i = \frac{1}{N}$ for $i = 1, \dots, N$

for $r = 1, \dots, R$: weighted equally

train y_r to minimize $\epsilon = \sum_{i=1}^N w_i [y_r(\mathbf{x}_i) \neq t_i]$

compute $\alpha_r = \ln \left(\frac{1-\epsilon}{\epsilon} \right)$ weight of model r

compute $w'_i = w_i e^{\alpha_r [y_r(\mathbf{x}_i) \neq t_i]}$ scale w_i by $\frac{1-\epsilon}{\epsilon}$ if \mathbf{x}_i classified incorrectly, otherwise no change

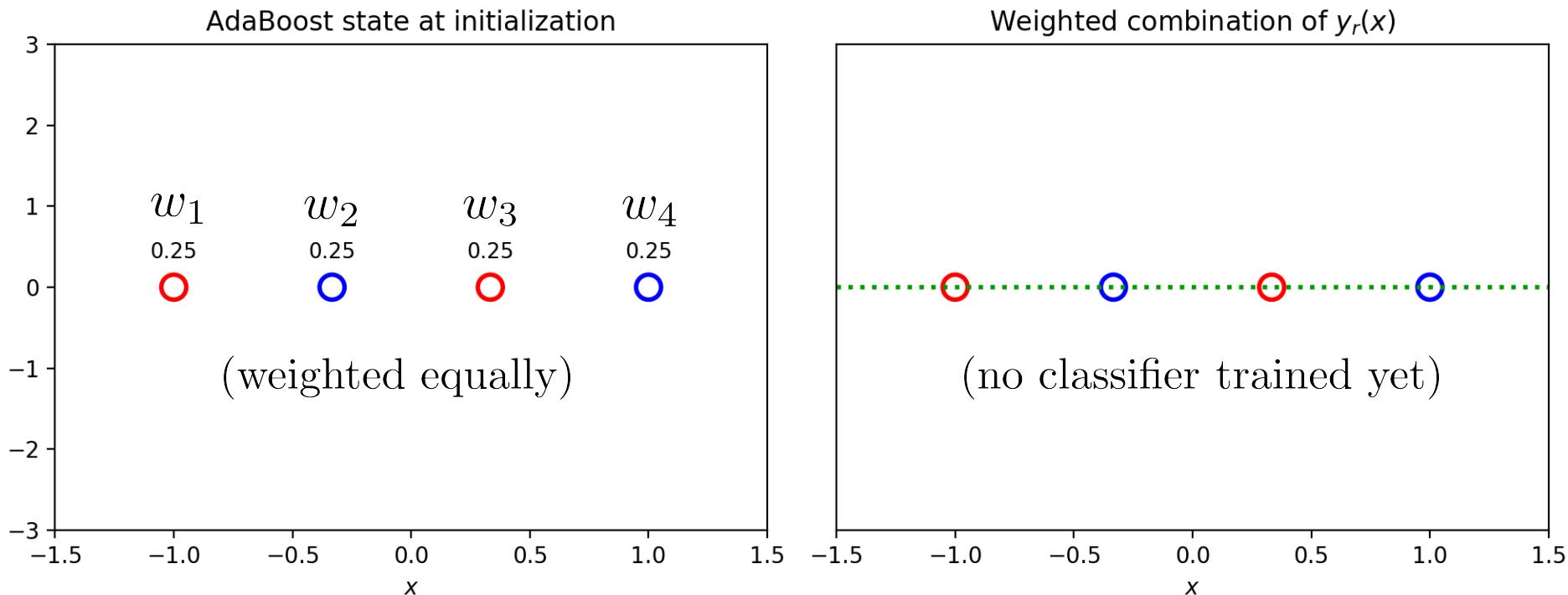
update $w_i \leftarrow \frac{w'_i}{\sum_i w'_i}$ re-normalize weights so that $\sum_i w_i = 1$

predict class by $y(\mathbf{x}) = \text{sign} \left(\sum_{r=1}^R \alpha_r y_r(\mathbf{x}) \right)$

sklearn normalizes its decision function, so slightly different (lab 6) ²¹

AdaBoost algorithm (binary)

Example using a decision tree with $\text{max_depth}=1$ (one split) as the “weak learner,” also called a “decision stump.” This is very popular, and the default in sklearn.



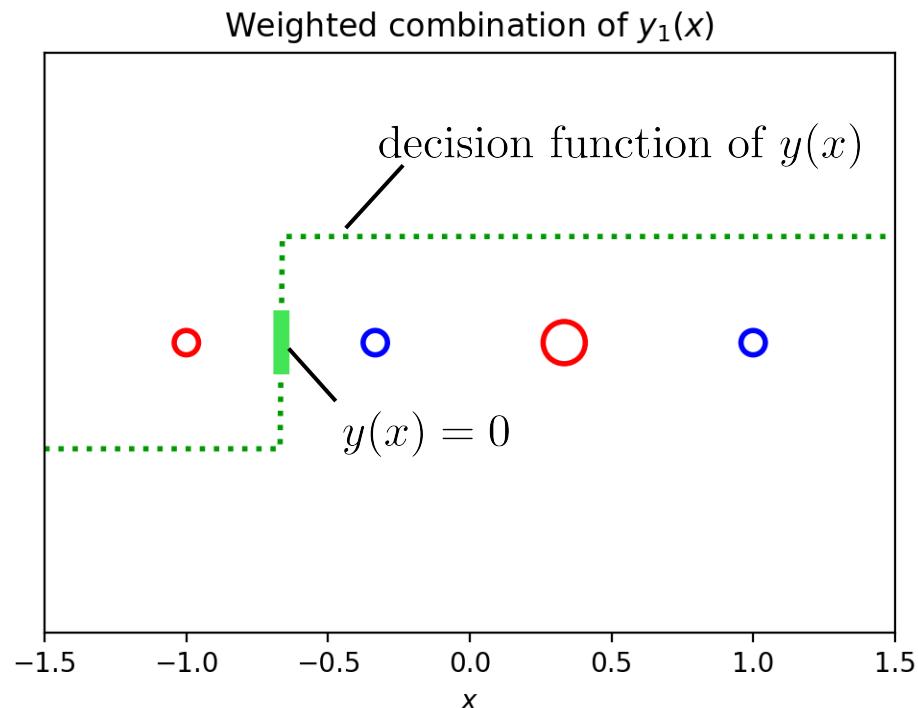
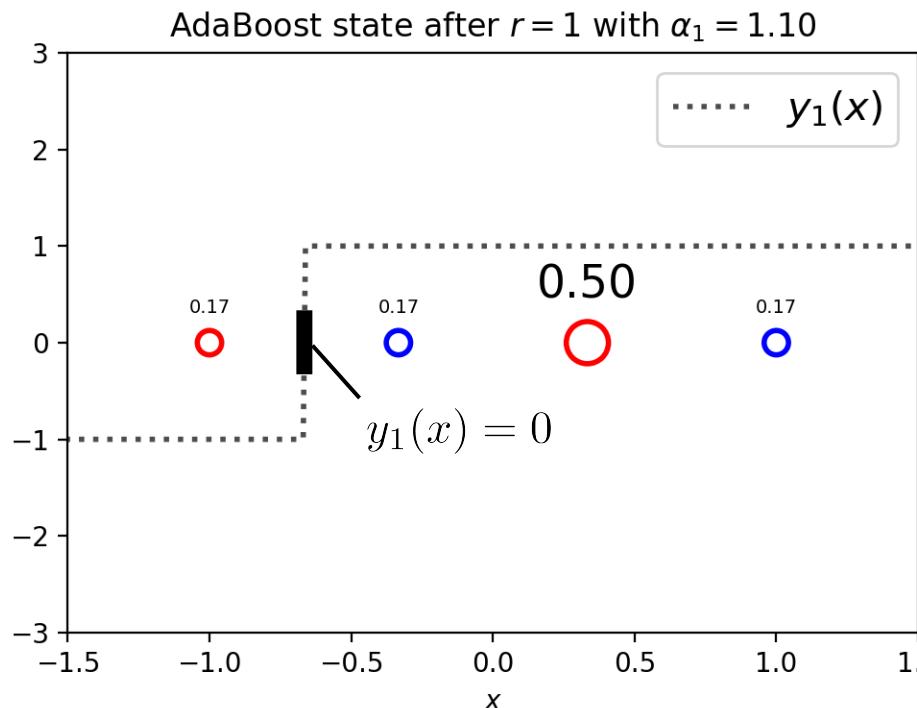
$$y(x) = \text{sign}(0)$$

(This example uses a 1-dimensional feature space.
The y -axis shows decision function value, not x_2 !)

$$\mathcal{D} = \{(-1, -1), (-\frac{1}{3}, +1), (\frac{1}{3}, -1), (1, +1)\}$$

AdaBoost algorithm (binary)

Example using a decision tree with $\text{max_depth}=1$ (one split) as the “weak learner,” also called a “decision stump.” This is very popular, and the default in sklearn.

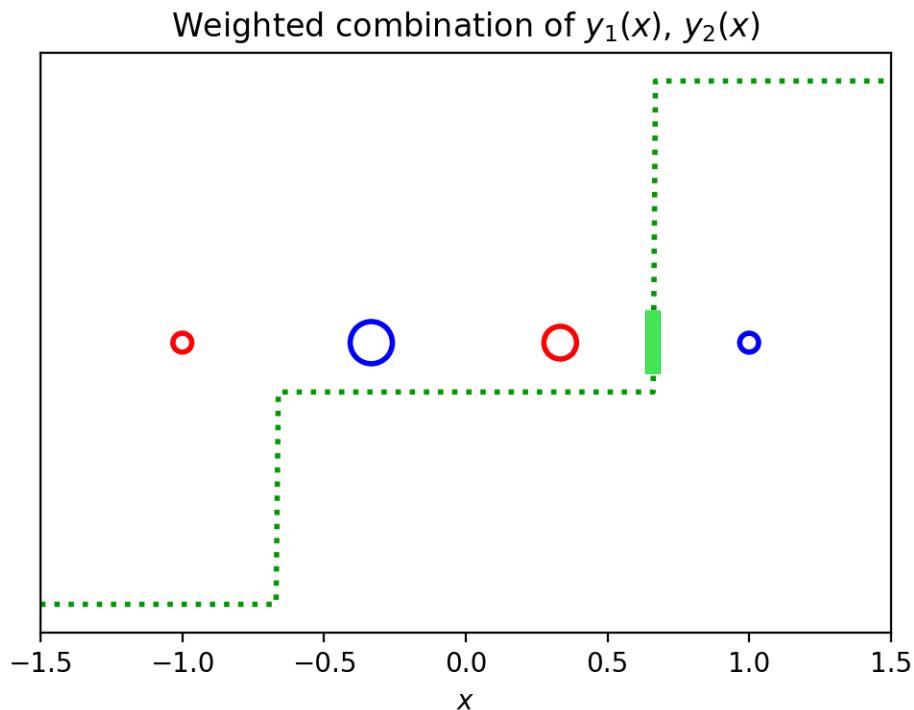
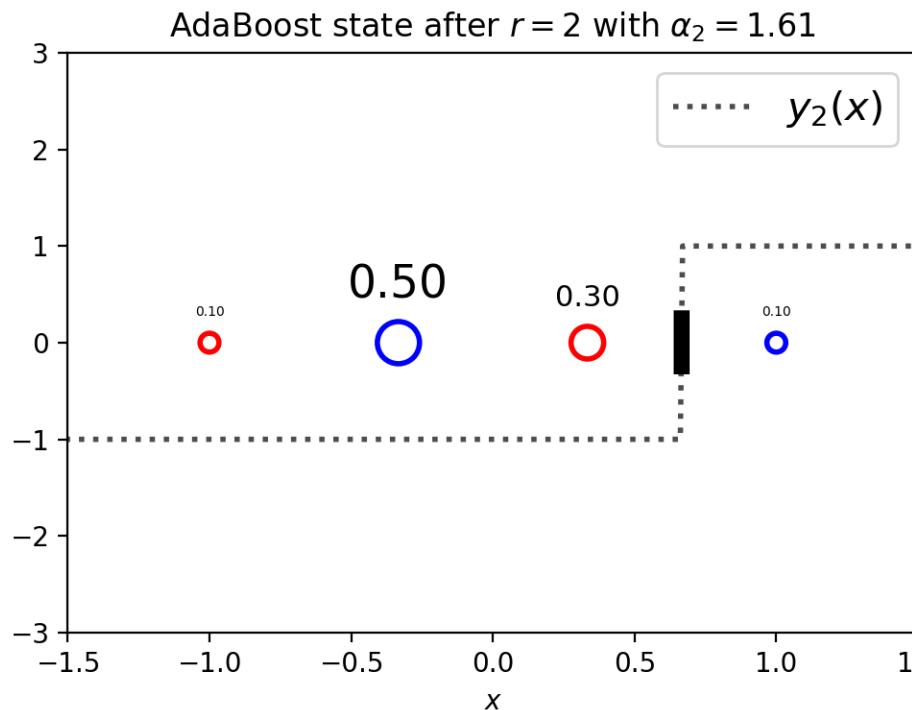


$$y(x) = \text{sign}(1.1y_1(x))$$

$$\mathcal{D} = \{(-1, -1), (-\frac{1}{3}, +1), (\frac{1}{3}, -1), (1, +1)\}$$

AdaBoost algorithm (binary)

Example using a decision tree with $\text{max_depth}=1$ (one split) as the “weak learner,” also called a “decision stump.” This is very popular, and the default in sklearn.

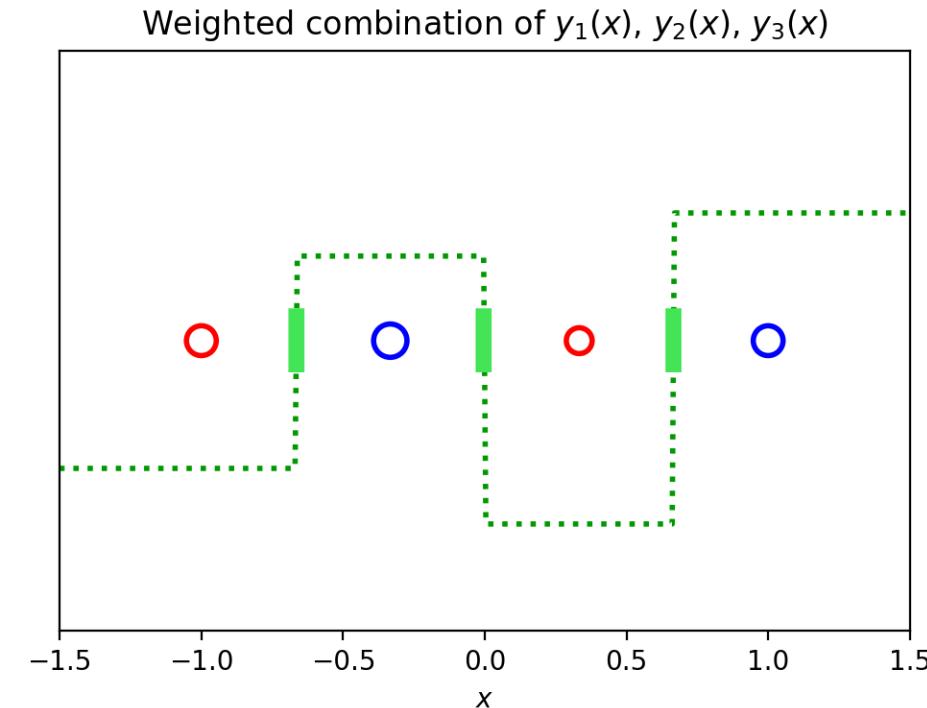
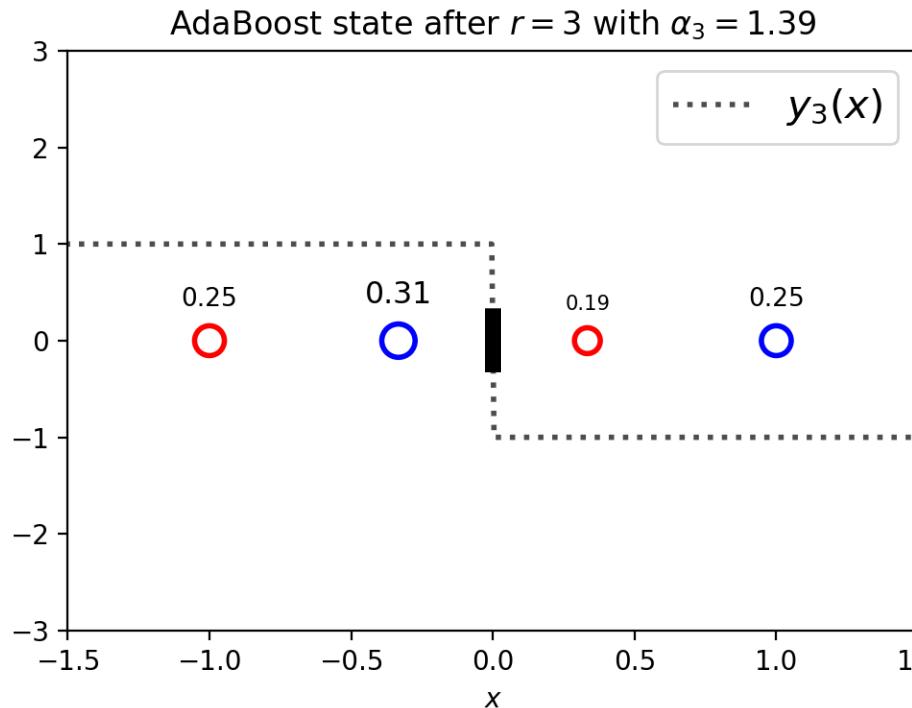


$$y(x) = \text{sign} (1.10y_1(x) + 1.61y_2(x))$$

$$\mathcal{D} = \{(-1, -1), \left(-\frac{1}{3}, +1\right), \left(\frac{1}{3}, -1\right), (1, +1)\}$$

AdaBoost algorithm (binary)

Example using a decision tree with $\text{max_depth}=1$ (one split) as the “weak learner,” also called a “decision stump.” This is very popular, and the default in sklearn.

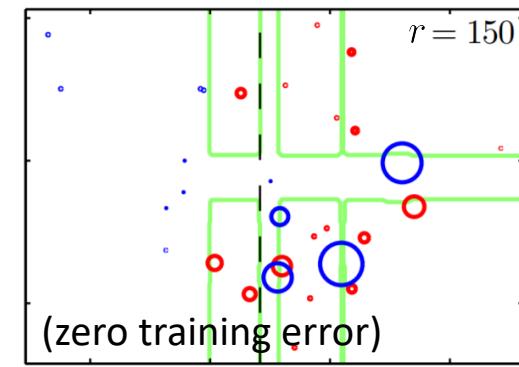
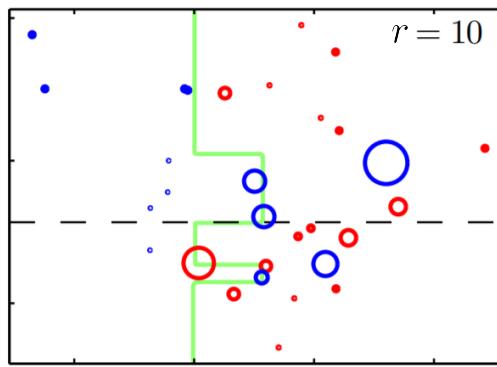
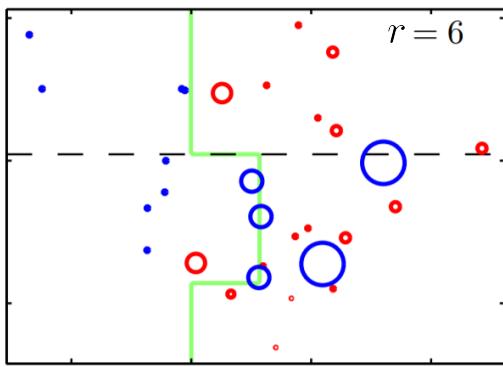
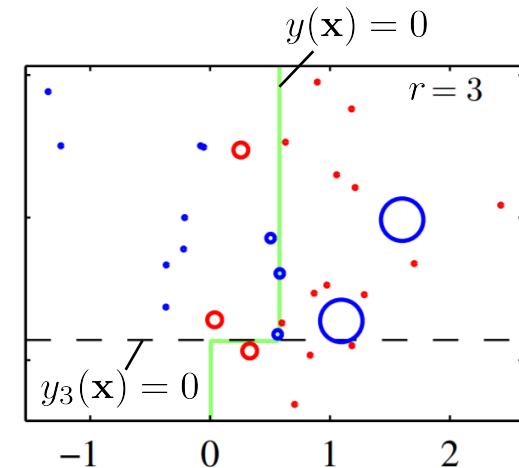
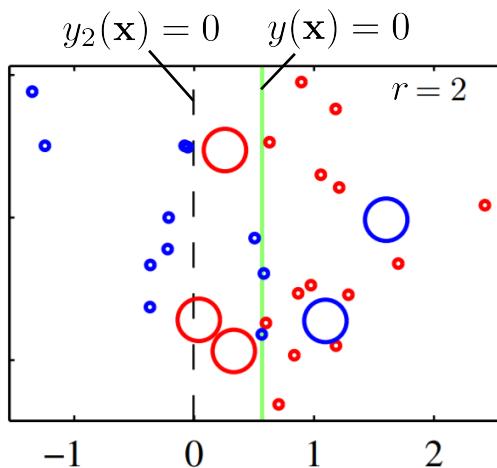
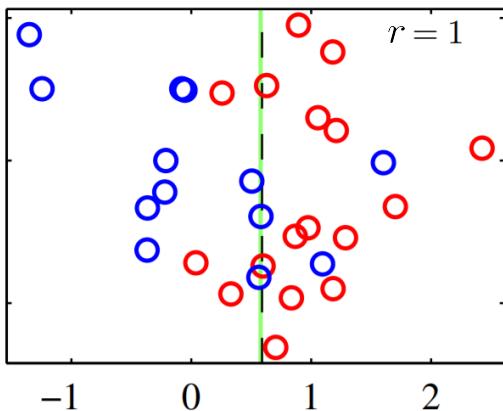


$$y(x) = \text{sign} (1.10y_1(x) + 1.61y_2(x) + 1.39y_3(x))$$

$$\mathcal{D} = \{(-1, -1), (-\frac{1}{3}, +1), (\frac{1}{3}, -1), (1, +1)\}$$

These weights are not identical to sklearn’s because there are slight variants of AdaBoost.

"not perfect separation, but better than random guessing"



i.e. "decision stumps"

Figure 14.2 Illustration of boosting in which the base learners consist of simple thresholds applied to one or other of the axes. Each figure shows the number r of base learners trained so far, along with the decision boundary of the most recent base learner (dashed black line) and the combined decision boundary of the ensemble (solid green line). Each data point is depicted by a circle whose radius indicates the weight assigned to that data point when training the most recently added base learner. Thus, for instance, we see that points that are misclassified by the $r = 1$ base learner are given greater weight when training the $r = 2$ base learner.

Gradient boosting

- Also searches over weights $\{\alpha_1, \dots, \alpha_R\}$ that could combine ‘weak’ models into a ‘strong’ one:

$$y(\mathbf{x}) = \sum_{r=1}^R \alpha_r y_r(\mathbf{x})$$

any number, not just -1 or +1

- A regression example on (\mathbf{x}_i, t_i) gives sense :

1. choose $y_0(\mathbf{x}) = \frac{1}{N} \sum_{i=1}^N t_i$

mean of targets (a constant)

try to predict this residual

2. train $y_1(\mathbf{x})$ on data $(\mathbf{x}_i, t_i - y_0(\mathbf{x}_i))$

3. predict $y(\mathbf{x}) = \underbrace{y_0(\mathbf{x})}_{\text{initial guess}} + \underbrace{y_1(\mathbf{x})}_{\text{correction}} + \dots$

corrections
on corrections
on corrections?

But wait, here all the $\alpha_r = 1$.
Where do the α_r come in??

Gradient boosting

- Suppose overall training goal is to minimize squared error $\ell(t, y) = \frac{1}{2} \sum_{i=1}^N (t_i - y(\mathbf{x}_i))^2$ of the *strong* model
 - vector of all strong predictions $y(\mathbf{x}_i)$, short for $\mathbf{y}(\mathbf{X})$
- Given current strong model $y(\mathbf{x})$, what change to that function would most rapidly reduce strong training loss $\ell(t, y)$?
- Idea: Change $y(\mathbf{x})$ in a ‘direction’ (in function space) that gives a *steepest descent* step w.r.t. $\ell(t, y)$!
 - Wait, we somehow compute $\nabla_y \ell(t, y)$? ... um, what?
 - It’s OK! For squared error loss: just negative of i^{th} residual

$$\frac{\partial \ell}{\partial y(\mathbf{x}_i)} \left[\frac{1}{2} \sum_{i=1}^N (t_i - y(\mathbf{x}_i))^2 \right] = \overbrace{y(\mathbf{x}_i) - t_i}^{\text{just negative of } i^{\text{th}} \text{ residual}}$$



Let's re-do our example:

1. let $y(\mathbf{x}) = y_0$ where $y_0 = \frac{1}{N} \sum_{i=1}^N t_i$

- 2a. compute $\nabla_{\mathbf{y}} \ell(\mathbf{t}, \mathbf{y})$ on trainint data

$$\begin{bmatrix} y(\mathbf{x}_1) - t_1 \\ \vdots \\ y(\mathbf{x}_N) - t_N \end{bmatrix}$$

- 2b. train $y_1(\mathbf{x})$ on data $(\mathbf{x}_i, t_i - y(\mathbf{x}_i))$
so that $y_1(\mathbf{x}) \approx -\frac{\partial \ell}{\partial y(\mathbf{x})}$

3. let $y^{\text{new}}(\mathbf{x}) = y(\mathbf{x}) - \eta \frac{\partial \ell}{\partial y(\mathbf{x})}$ for step size $\eta \geq 0$
or, equivalently, choose some step size $\alpha_1 \geq 0$ for

$$y^{\text{new}}(\mathbf{x}) = \underbrace{y_0}_{\text{initial guess}} + \underbrace{\alpha_1 y_1(\mathbf{x})}_{\text{correction}}$$

for squared error loss, this is just the predicted training residual at \mathbf{x} , i.e. residual \mathbf{x} would have had were it included in the original training set.

Key: this is only defined for the training points \mathbf{x}_i

... whereas y_1 is defined for *any* \mathbf{x} can use to predict!

Gradient boosting: why defined like this?

- Why must we define gradient boosting in terms of this fancy $\nabla_y \ell(t, y)$ thing? What's wrong with just training each "corrector" directly on residuals?
- Our overall training goal is to minimize loss $\ell(t, y)$ that we chose for the *strong model* $y(\mathbf{x})$.
 - Loss could be squared error $\ell(t, y) = \frac{1}{2} \sum_{i=1}^N (t_i - y(\mathbf{x}_i))^2$, or anything else that we want to train strong model on.
 - Key: Want ability to use *any* weak model $y_r(\mathbf{x})$, even if its usual training algorithm minimizes some other loss ℓ_{weak} .
 - Treat $y_r(\mathbf{x})$ as **black box**, can't assume it directly minimizes ℓ !
- By asking $y_r(\mathbf{x})$ to mimick $\frac{\partial \ell}{\partial y(\mathbf{x})}$ at each \mathbf{x}_i , it can use whatever loss function it wants. After y_r is trained, we just need to find a good step size α_r to "boost" with!

Gradient boosting algorithm

let $y(\mathbf{x}) = y_0$ where $y_0 = \arg \min_{y'_0 \in \mathbb{R}} \ell(\mathbf{t}, \mathbf{y}'_0)$

for $r = 1, \dots, R$:
find constant y_0 that minimizes ℓ

compute $\boldsymbol{\delta} = \nabla_{\mathbf{y}} \ell(\mathbf{t}, \mathbf{y})$ on the training set

$\frac{\partial \ell}{\partial y(\mathbf{x}_i)}$ i.e. rate of change in $\ell(\mathbf{t}, \mathbf{y})$
per unit change in $y(\mathbf{x}_i)$

train $y_r(\mathbf{x})$ on data $(\mathbf{x}_i, -\delta_i)$ using ℓ_{weak}

all current predictions

predicted gradient (predicted $\boldsymbol{\delta}$)

find $\alpha_r = \arg \min_{\alpha} \ell(\mathbf{t}, \mathbf{y} + \alpha \mathbf{y}_r)$ by line search

easy 1D optimization method
(don't need to know for course)

or just use a fixed
learning rate

update $y(\mathbf{x}) \leftarrow y(\mathbf{x}) + \alpha y_r(\mathbf{x})$ add 'best' correction term

predict $y(\mathbf{x}) = \sum_{r=1}^R \alpha_r y_r(\mathbf{x})$

Scikit-learn supports several loss functions but only *tree* learners

1.11.4.1. Classification

`GradientBoostingClassifier` supports both binary and multi-class classification. The following example shows how to fit a gradient boosting classifier with 100 decision stumps as weak learners:

```
>>> from sklearn.datasets import make_hastie_10_2
>>> from sklearn.ensemble import GradientBoostingClassifier

>>> X, y = make_hastie_10_2(random_state=0)
>>> X_train, X_test = X[:2000], X[2000:]
>>> y_train, y_test = y[:2000], y[2000:]

>>> clf = GradientBoostingClassifier(n_estimators=100, learning_rate=1.0,
...     max_depth=1, random_state=0).fit(X_train, y_train)
>>> clf.score(X_test, y_test)
0.913...
```

1.11.4.2. Regression

`GradientBoostingRegressor` supports a number of [different loss functions](#) for regression which can be specified via the argument `loss`; the default loss function for regression is least squares (`'ls'`).

```
>>> import numpy as np
>>> from sklearn.metrics import mean_squared_error
>>> from sklearn.datasets import make_friedman1
>>> from sklearn.ensemble import GradientBoostingRegressor

>>> X, y = make_friedman1(n_samples=1200, random_state=0, noise=1.0)
>>> X_train, X_test = X[:200], X[200:]
>>> y_train, y_test = y[:200], y[200:]
>>> est = GradientBoostingRegressor(n_estimators=100, learning_rate=0.1,
...     max_depth=1, random_state=0, loss='ls').fit(X_train, y_train)
>>> mean_squared_error(y_test, est.predict(X_test))
5.00...
```

Classic paper showing how to interpret AdaBoost and others as “gradient descent in function space”

Boosting Algorithms as Gradient Descent

Llew Mason

Research School of Information
Sciences and Engineering
Australian National University
Canberra, ACT, 0200, Australia
lmason@syseng.anu.edu.au

Jonathan Baxter

Research School of Information
Sciences and Engineering
Australian National University
Canberra, ACT, 0200, Australia
Jonathan.Baxter@anu.edu.au

Peter Bartlett

Research School of Information
Sciences and Engineering
Australian National University
Canberra, ACT, 0200, Australia
Peter.Bartlett@anu.edu.au

Marcus Frean

Department of Computer Science
and Electrical Engineering
The University of Queensland
Brisbane, QLD, 4072, Australia
marcusf@elec.uq.edu.au

Abstract

We provide an abstract characterization of boosting algorithms as gradient descent on cost-functionals in an inner-product function space. We prove convergence of these functional-gradient-descent algorithms under quite weak conditions.

(not on exam!)



Summary of bagging, boosting

- **Bagging** can be run in parallel; main goal is remove the errors caused by instability (sensitivity to training set, or “variance”); a *variance reduction* technique.
- **Boosting** is run sequentially; main goal is to reduce the systematic errors (prediction “bias”) made by “weak learners”; a *bias reduction* technique that also inherits some of the *variance reduction* of bagging.
 - We discuss “bias” and “variance” next.
- AdaBoost and Gradient Boosting are both **greedy** algorithms. Neither guarantees that the returned combination of $\{\alpha_1, \dots, \alpha_R\}$ and trained weak learners is the best possible for a budget of R .

Ensembling in general

Ensembles are consistently better than single models, in research and in almost all Kaggle competitions!



- When we take multiple models and combine their predictions to create a new model, that's called *ensembling* and the new model is called an *ensemble*.
- Bagging and boosting are “ensemble methods” because the model that they return is an *ensemble*.
- Ensembling is much broader and more flexible:
 - Any mixture of model types that are suitable for the training task (SVMs, decision trees, neural networks, etc).
- **Stacking:** train models $\{1, \dots, R\}$, then use their predictions as features to train a final model (still on original targets) that is “stacked on top” of the others.

*Bagging is stacking but where the final model has no parameters: just take average!
Sometimes training a model “on top” helps more than taking average.*

PRML Readings

- 3rd paragraph on p.23 (“One approach to determining frequentist error bars is the *bootstrap*”)
- §14.2.0 Committees
- §14.3.0 Boosting

OPTIONAL:

- §14.3.1 Minimizing exponential error
(Only if curious about how the algorithm was derived)