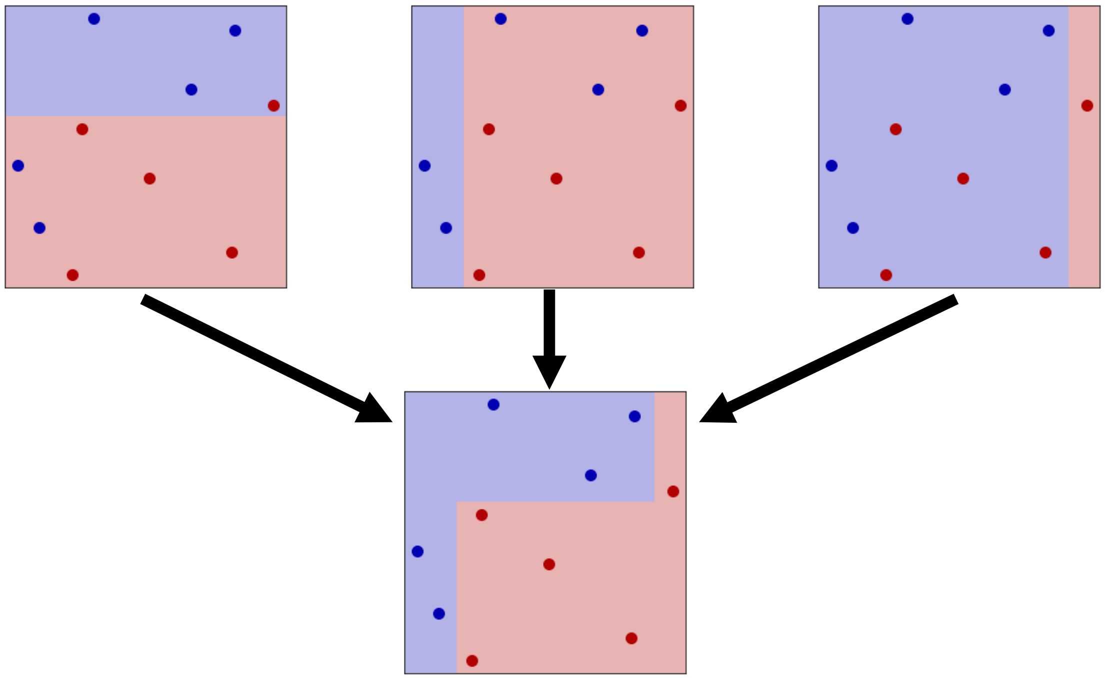
CS273A:
Ensemble Methods



Prof. Alexander Ihler

Fall 2023

Ensemble methods

- Why learn one classifier when you can learn many?

- Ensemble: combine many predictors
  - (Weighted) combinations of predictors
  - May be same type of learner or different

"Who wants to be a millionaire?"

Various options for getting help:

Ensemble Methods

Basic Ensembles

Committees | Stacking | Mix of Experts

Bagging

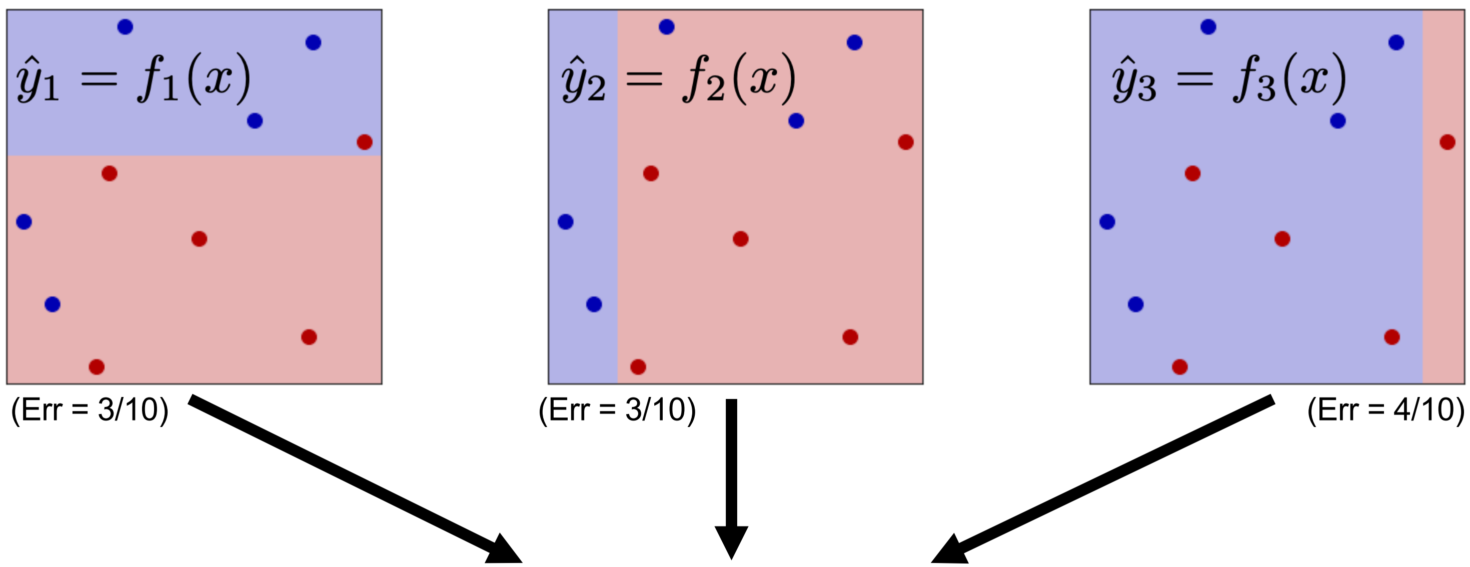
Gradient Boosting

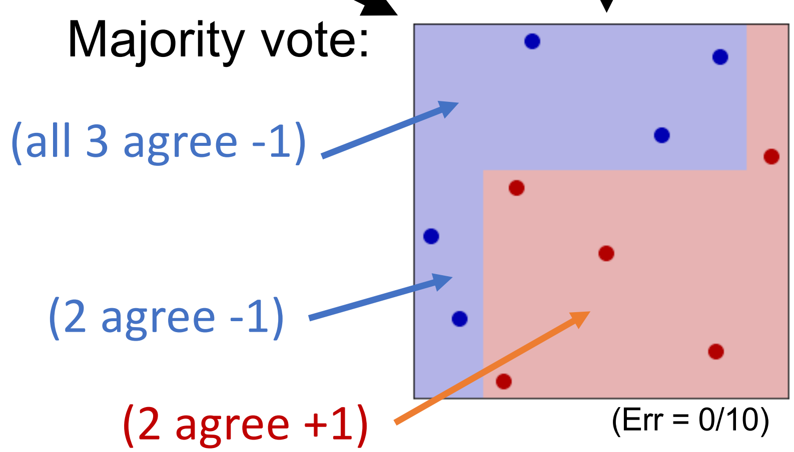
AdaBoost

Simple ensembles

- "Committees"
  - Average / majority vote of several predictors

$y \in \{-1, +1\}$





or, weighted average:

$$\hat{y} = \text{sign}\Big(\sum_k \alpha_k f_k(x)\Big)$$

e.g., up-weight better predictors
(needs more than 3…)

"Stacked" ensembles

- Train a "predictor of predictors"
  - Treat individual predictors as features

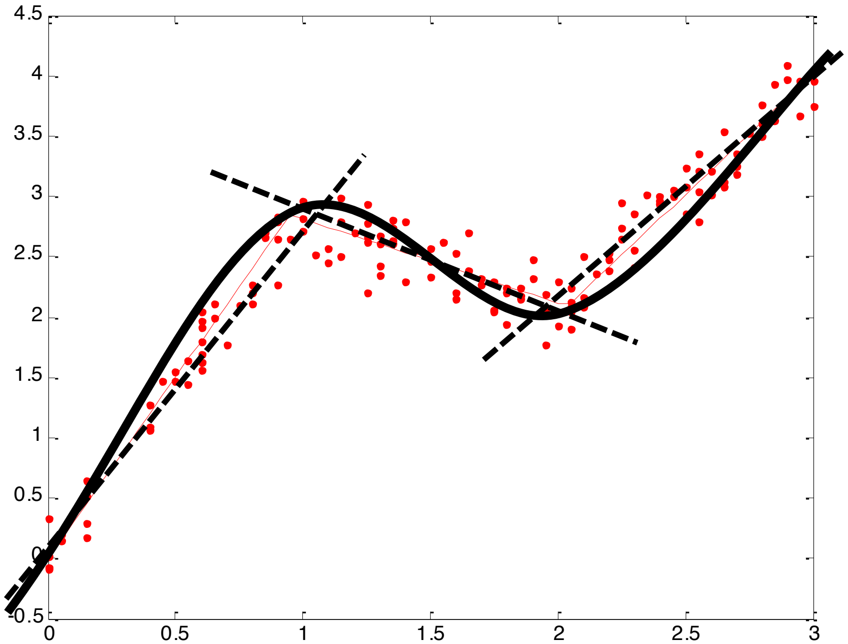
$$\hat{y}_1 = f_1(x_1,x_2,\dots)$$
$$\hat{y}_2 = f_2(x_1,x_2,\dots) \quad\Rightarrow\quad \hat{y}_e = f_e(\hat{y}_1, \hat{y}_2, \dots)$$
$$\dots$$

  - Similar to multi-layer perceptron idea
  - Special case: binary, f_e linear => weighted vote
  - Can train stacked learner f_e on validation data
    - Avoids giving high weight to overfit models

Mixtures of experts

- Can make weights depend on x
  - Weight α_z(x) indicates "expertise"
  - Combine using weighted average   (or even just pick largest)

Example:



Mixture of three linear predictor experts

Weighted average:

$$f(x;\omega,\theta) = \sum_z \alpha_z(x;\omega)\ f_z(x;\theta_z)$$

Weights: (multi) logistic regression

$$\alpha_z(x;\omega) = \frac{\exp(x\cdot\omega^z)}{\sum_c \exp(x\cdot\omega^c)}$$

If loss, learners, weights are all differentiable, can train jointly…

Ensemble Methods

Basic Ensembles

Committees | Stacking | Mix of Experts

Bagging

Gradient Boosting

AdaBoost

Ensemble methods

- Where can we get a diverse collection of learners?
  - Maybe create one artificially?
- "Bagging" = bootstrap aggregation
  - Learn many classifiers, each with only part of the data
  - Combine through model averaging
- Remember overfitting: "memorize" the data
  - Used test data to see if we had gone too far
  - Cross-validation
    - Make many splits of the data for train & test
    - Each of these defines a classifier
    - Typically, we use these to check for overfitting
    - Could we instead combine them to produce a better classifier?

Bagging

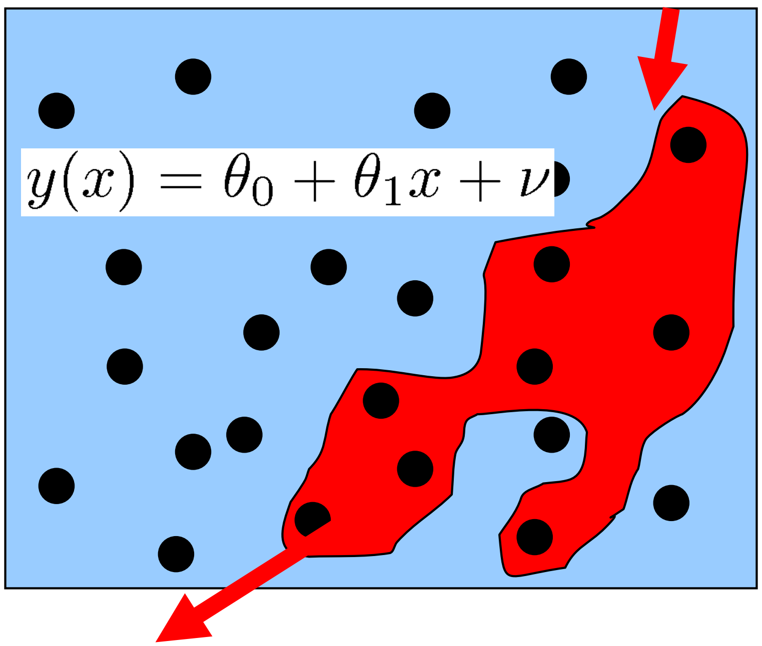
- Bootstrap
  - Create a random subset of data by sampling
  - Draw m’ of the m samples, with replacement (some variants w/o)
    - Some data left out; some data repeated several times

- Bagging
  - Repeat K times
    - Create a training set of m’ ≤ m examples
    - Train a classifier on the random training set
  - To test, run each trained classifier
    - Each classifier votes on the output, take majority
    - For regression: each regressor predicts, take average

- Notes:
  - Some complexity control: harder for each to memorize data
  - Doesn’t work for linear models (average of linear functions is linear function), but perceptrons OK (linear + threshold = nonlinear)

Bias / variance

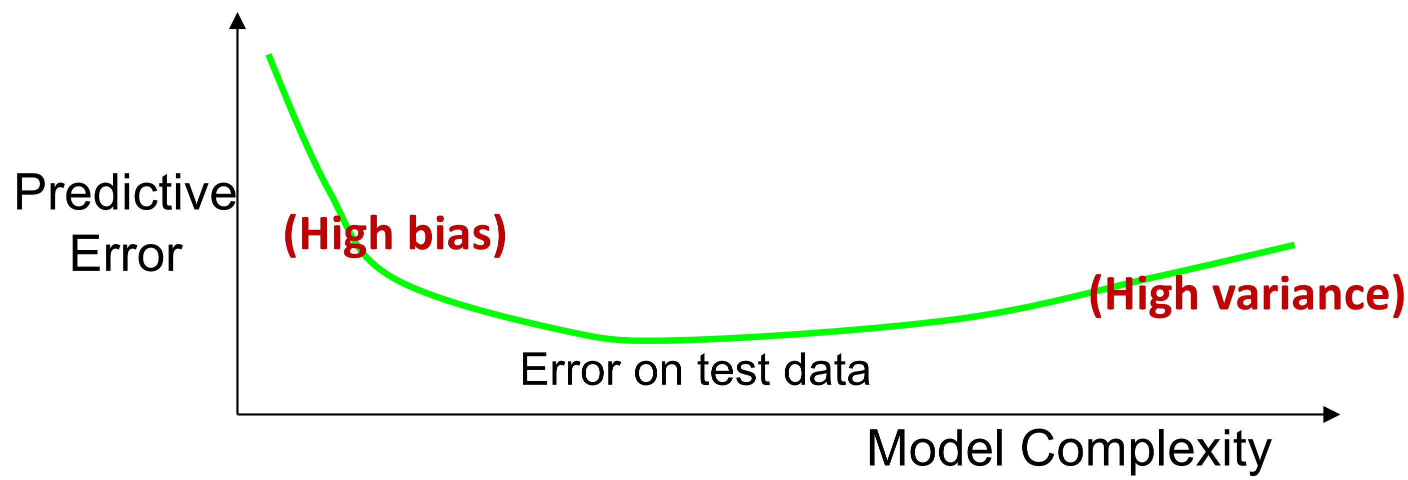
“The world” **Data we observe**



$$y(x) = \theta_0 + \theta_1 x + \nu$$

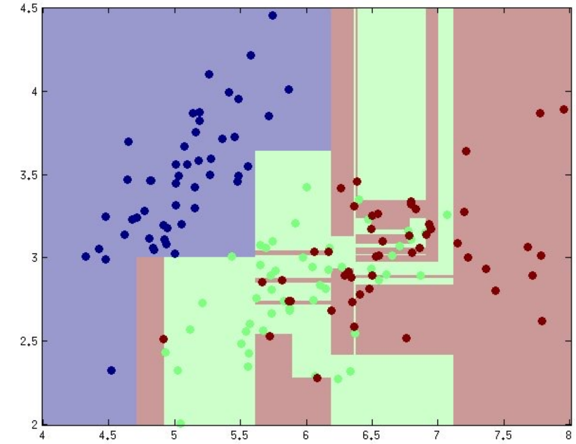
$$\hat{y}(x) = \hat{\theta}_0 + \hat{\theta}_1 x$$

- We only see a little bit of data
- Can decompose error into two parts
  - Bias – error due to model choice
    - Can our model represent the true best predictor?
    - Gets better with more complexity
  - Variance – randomness due to data size
    - Better w/ more data, worse w/ complexity



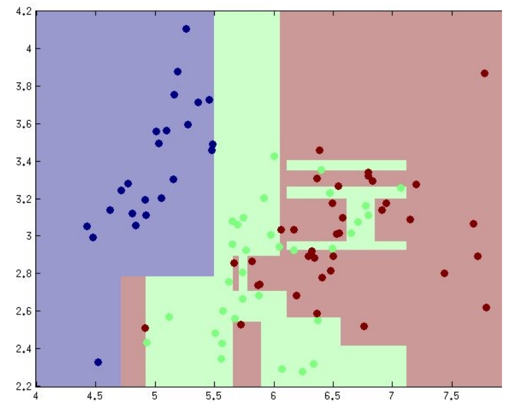
Bagged decision trees

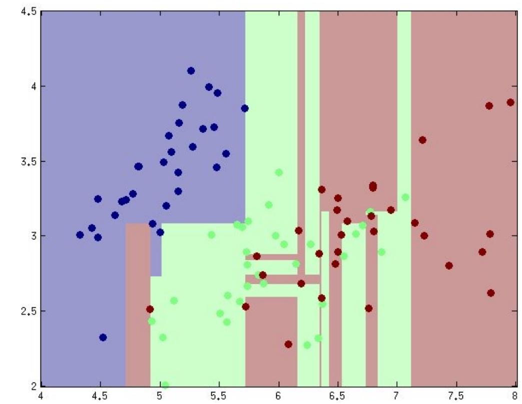
- Randomly resample data
- Learn a decision tree for each
  - No max depth = very flexible class of functions
  - Learner is low bias, but high variance

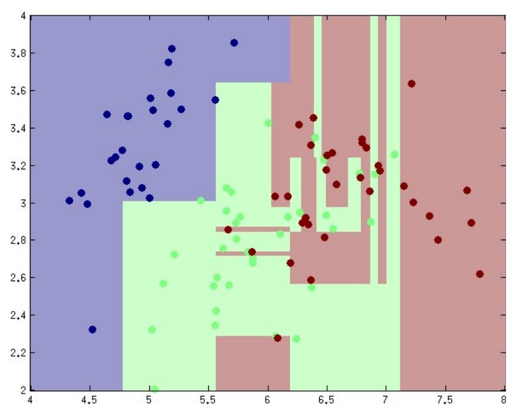


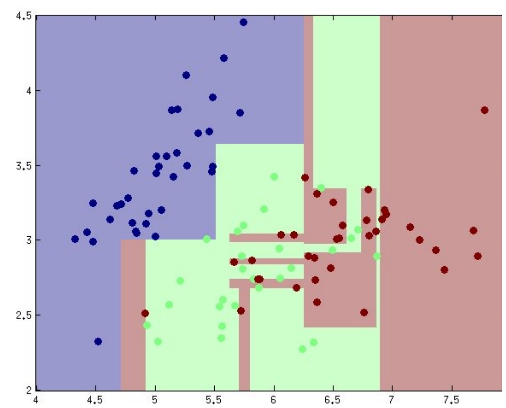
Full data set

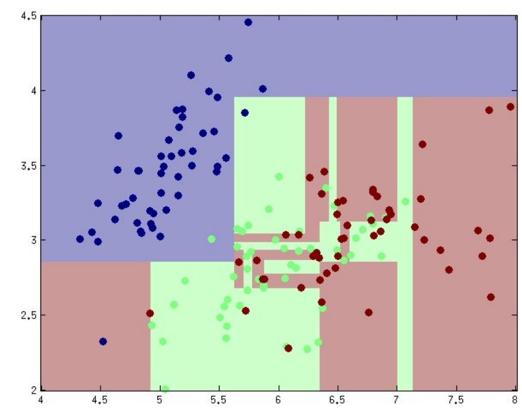
Sampling:
simulates "equally likely" data sets we could have observed instead, & their classifiers



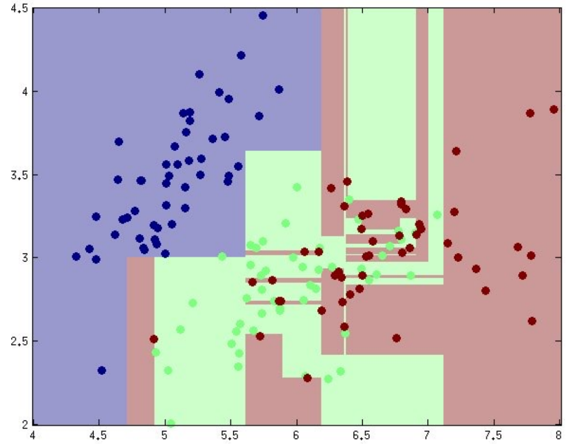








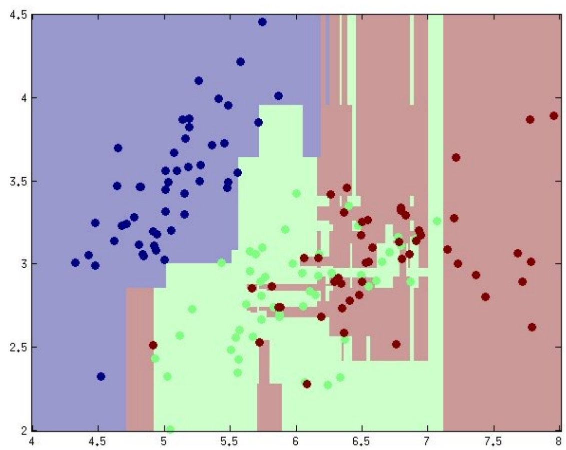
Bagged decision trees



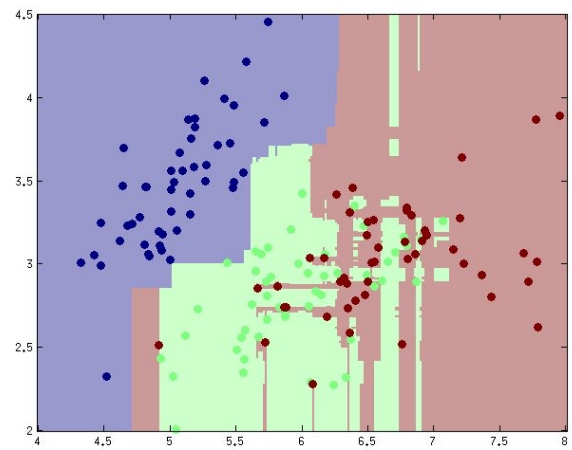
Full data set

- Average over collection
  - Classification: majority vote
- Reduces memorization effect
  - Not every predictor sees each data point
  - Lowers effective “complexity” of the overall average
  - Usually, better generalization performance
  - Intuition: reduces variance while keeping bias low

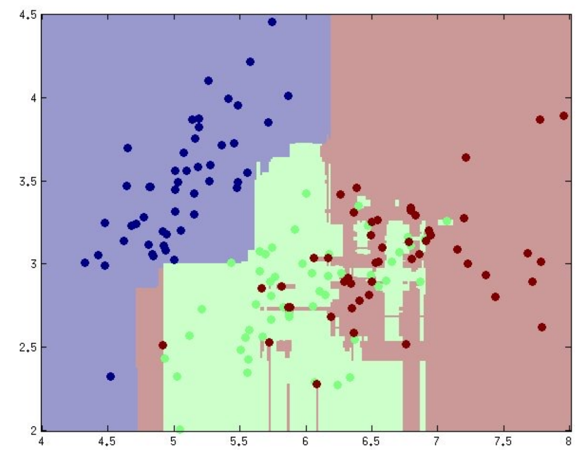
Avg of 5 trees



Avg of 25 trees



Avg of 100 trees



Bagging in Python

```
# Load data set X, Y for training the ensemble…
m,n = X.shape
classifiers = [ None ] * num_bags                # Allocate space for learners
for b in range(num_bags):
    # Bootstrap sample a dataset of size "mBag"; typically just pick mBag = m
    ind = np.floor( mBag * np.random.rand( m ) ).astype(int)
    Xb, Yb  =  X[ind,:] , Y[ind]                 #    select the data at those indices
    classifiers[i] = MyClassifier(Xb, Yb)        # Train a model on data Xi, Yi
```

```
# test on data Xtest
mTest = Xtest.shape[0]
predict = np.zeros( (mTest, num_bags) ).   # Allocate space for predictions from each model
for i in range(num_bags):
    predict[:,i] = classifiers[i].predict(Xtest)     # Apply each classifier

# Make overall prediction by majority vote
predict = np.mean(predict, axis=1) > 0   # if +1 vs -1
```

Random forests

- Bagging applied to decision trees
- Problem
  - With lots of data, we usually learn the same classifier
  - Averaging over these doesn’t help!
- Introduce extra variation in learner
  - At each step of training, only allow a (random) subset of features
  - Enforces diversity (“best” feature not available)
  - Keeps bias low (every feature available eventually)
  - Average over these learners (majority vote)

```
# in FindBestSplit(X,Y):
  for each of a subset of features
    for each possible split
      Score the split  (e.g. information gain)
  Pick the feature & split with the best score
  Recurse on left & right splits
```

Ensemble Methods

Basic Ensembles

Committees | Stacking | Mix of Experts

Bagging

Gradient Boosting

AdaBoost

Ensembles

- Weighted combinations of predictors
- “Committee” decisions
  - Trivial example
  - Equal weights (majority vote / unweighted average)
  - Might want to weight unevenly – up-weight better predictors

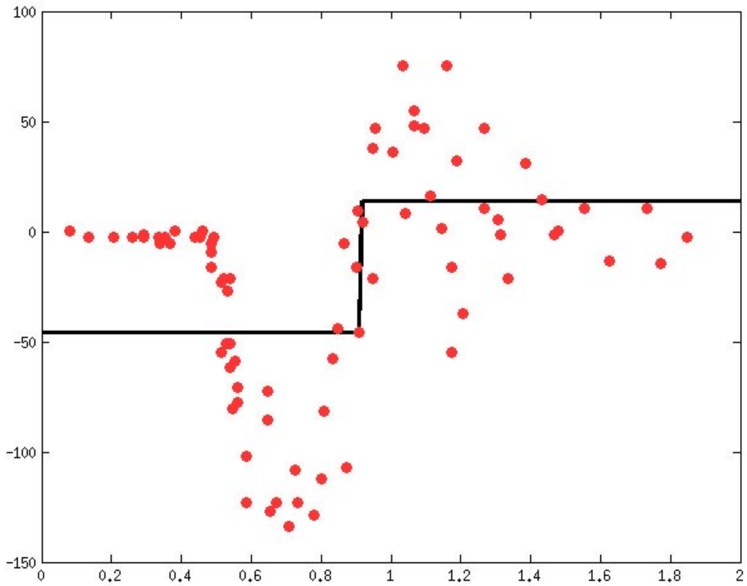
- Boosting
  - Focus new learners on examples that others get wrong
  - Train learners sequentially
  - Errors of early predictions indicate the “hard” examples
  - Focus later predictions on getting these examples right
  - Combine the whole set in the end
  - Convert many “weak” learners into a complex predictor

Gradient boosting

- Learn a regression predictor
- Compute the error residual
- Learn to predict the residual

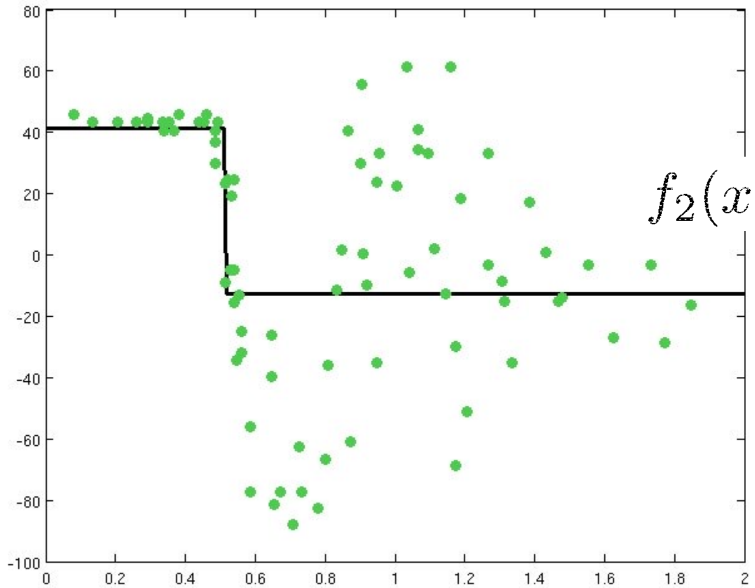
Learn a simple predictor…

$f_1(x^{(i)}) \approx y^{(i)}$



Then try to correct its errors

$\epsilon^{(i)} = y^{(i)} - f_1(x^{(i)})$



$f_2(x^{(i)}) \approx \epsilon^{(i)}$

Gradient boosting

- Learn a regression predictor
- Compute the error residual
- Learn to predict the residual

$$f_1(x^{(i)}) \approx y^{(i)}$$

$$\epsilon^{(i)} = y^{(i)} - f_1(x^{(i)})$$

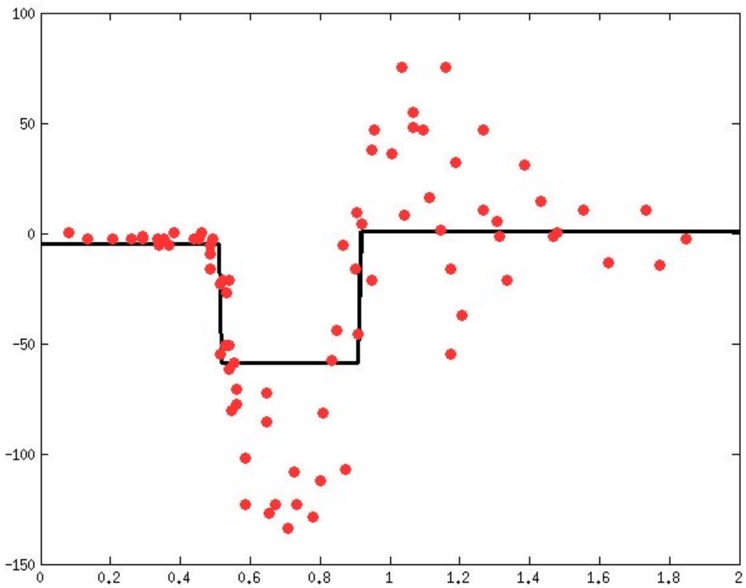
$$f_2(x^{(i)}) \approx \epsilon^{(i)}$$

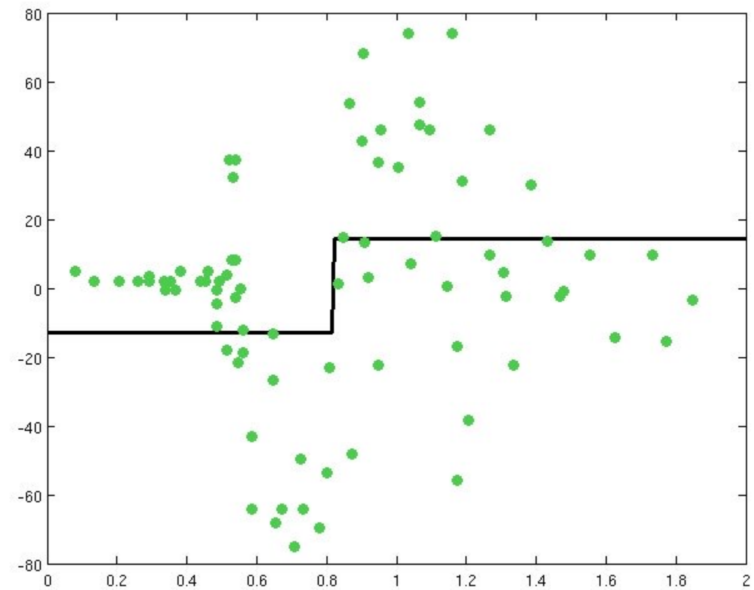
Combining gives a better predictor…

$$\Rightarrow \quad f_1(x^{(i)}) + f_2(x^{(i)}) \approx y^{(i)}$$

Can try to correct its errors also, & repeat

$$\epsilon_2^{(i)} = y^{(i)} - f_1(x^{(i)} - f_2(x^{(i)}) \quad \ldots$$



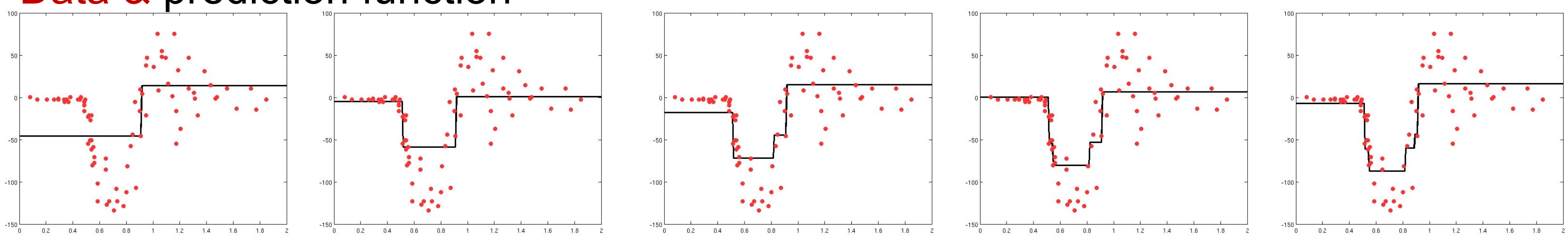


Gradient boosting

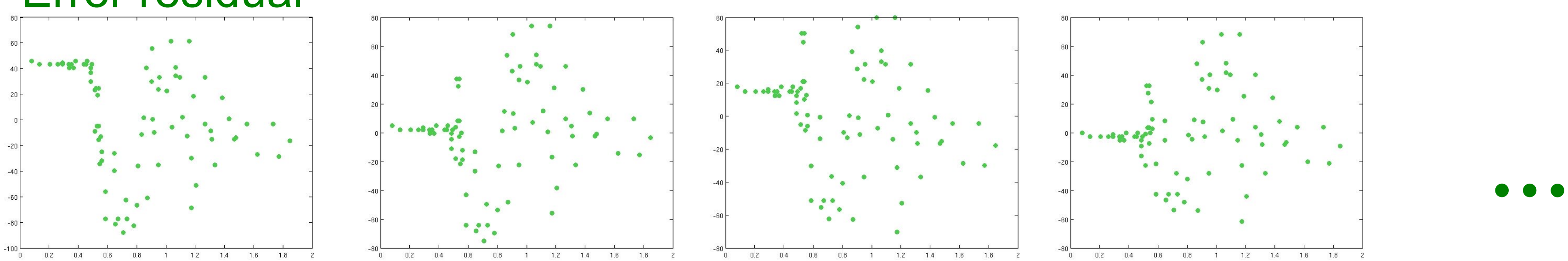
- Learn sequence of predictors
- Sum of predictions is increasingly accurate
- Predictive function is increasingly complex

$$y^{(i)} \approx \sum_z f_z(x^{(i)})$$

Data & prediction function



Error residual



Gradient boosting

- Make a set of predictions $\hat{y}[i]$
- The "error" in our predictions is $J(y,\hat{y})$
- For MSE: $J(.) = \sum (y[i] - \hat{y}[i])^2$
- We can "adjust" $\hat{y}$ to try to reduce the error
- $\hat{y}[i] = \hat{y}[i] + \text{alpha } f[i]$
- $f[i] \approx \nabla J(y, \hat{y})$ $= (y[i]-\hat{y}[i])$ for MSE
- Each learner is estimating the gradient of the loss function
- Gradient descent: take sequence of steps to reduce J
- Sum of predictors, weighted by step size alpha

Gradient boosting (classification)

- Ex: “logistic regression” using boosting
  - “Response” r(x)   (standard LR: a linear f’n of x)
  - Probability σ(r) \in [0,1]
  - Loss J(r) = \sum_i y$^{(i)}$ log r(x$^{(i)}$) + (1-y$^{(i)}$) log (1-r(x$^{(i)}$))

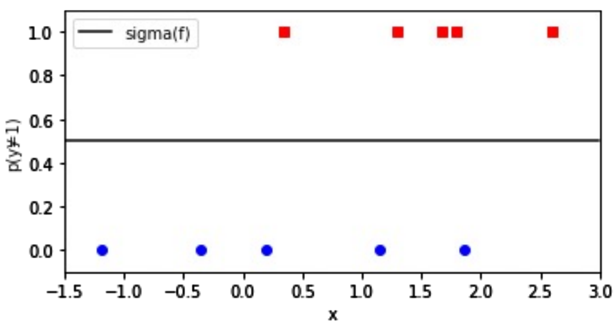
Learn a simple predictor…

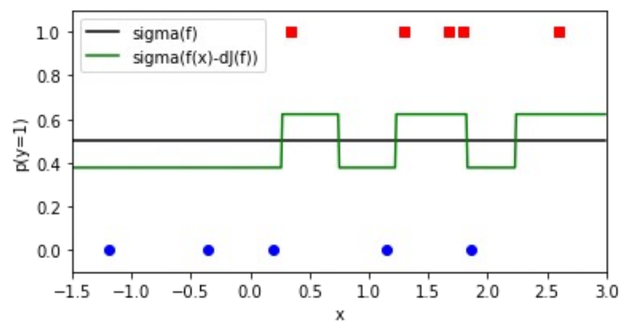
$$r_0(x^{(i)}) = 0.5$$

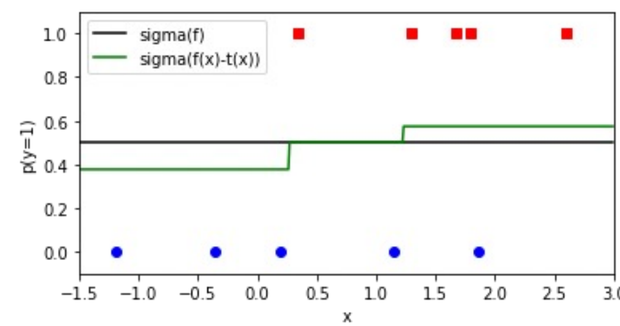
Find the loss gradient:

$$\frac{\partial J}{\partial r^{(i)}} = y^{(i)} - \sigma(r^{(i)})$$

Learn to approximate it:







Gradient boosting in Python

```
# Load data set X, Y …
learner = [None] * num_boost        # storage for ensemble of models
alpha = [1.0] * num_boost           # and weights of each learner

mu = Y.mean()        # often start with constant "mean" predictor
dJ = Y – mu          # subtract this prediction away (assumes MSE)
for b in range( num_boost ):
    learner[b] = MyRegressor( X, dJ )     # regress to predict gradient direction dJ using X
    alpha[b] = 1.0                        # alpha: "learning rate" or "step size"
    # smaller alphas need to use more classifiers, but may predict better given enough of them
    # compute the residual given our new prediction:
    dJ = dJ – alpha[b] * learner[b].predict(X).     # update gradient (assumes MSE loss)
```

```
# test on data Xtest
mTest = Xtest.shape[0]
predict = np.zeros( (mTest,) ) + mu     # Allocate space for predictions & add 1st (mean)
for b in range(num_boost):
   predict += alpha[b] * learner[b].predict(Xtest)  # Apply predictor of next residual & accum
```

Ensemble Methods

Basic Ensembles

Committees | Stacking | Mix of Experts

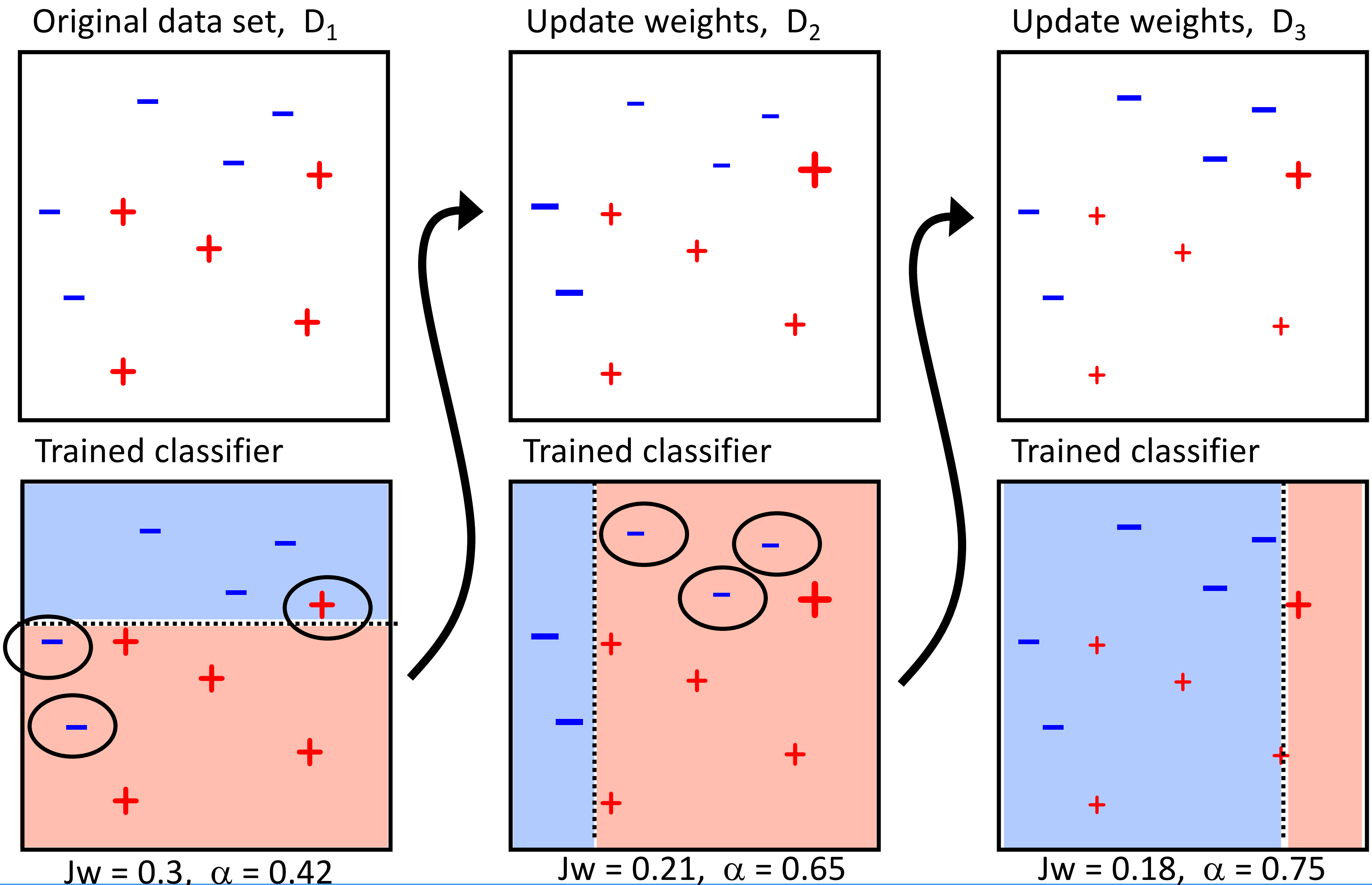
Bagging

Gradient Boosting

AdaBoost

Boosting example

Classes {+1 , -1}



Minimizing weighted error

- So far we've mostly minimized unweighted error
- Minimizing weighted error is no harder:

Unweighted average loss:

$$J(\theta) = \frac{1}{m} \sum_i J_i(\theta, x^{(i)})$$

Weighted average loss:

$$J(\theta) = \sum_i w_i J_i(\theta, x^{(i)})$$

For any loss (logistic MSE, hinge, …)

$$J(\theta, x^{(i)}) = \left(\sigma(\theta x^{(i)}) - y^{(i)}\right)^2$$

$$J(\theta, x^{(i)}) = \max\left[0\,,\ 1 - y^{(i)}\,\theta x^{(i)}\right]$$

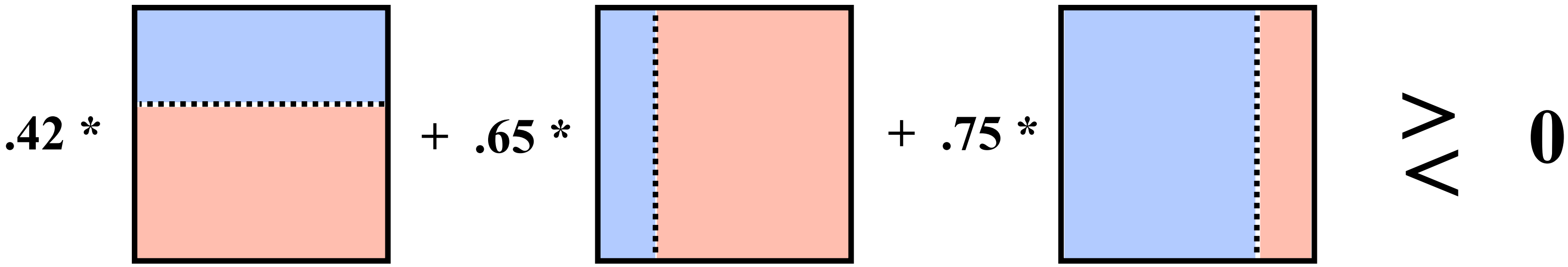
To learn decision trees, find splits to optimize *weighted* impurity scores:

p(+1) = total weight of data with class +1

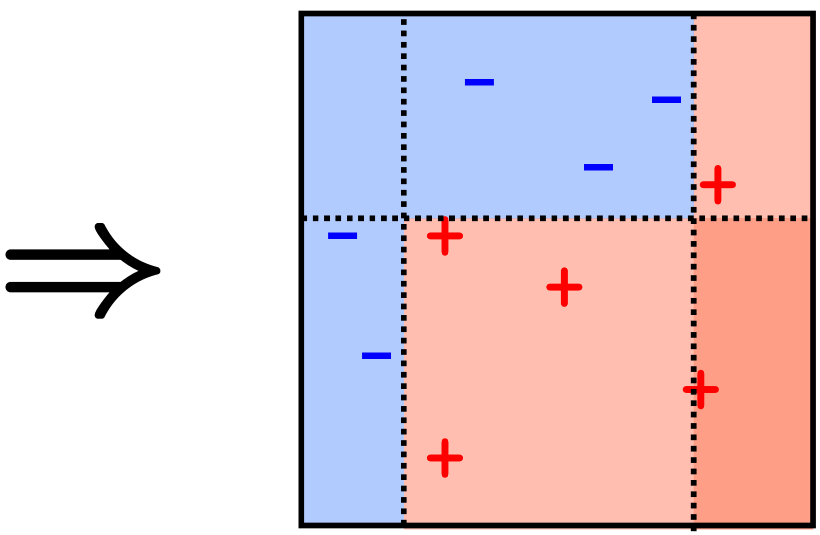
p(-1) = total weight of data with class -1 => H(p) = impurity

Boosting example

Weight each classifier and combine them:



Combined classifier



1-node decision trees
“decision stumps”
very simple classifiers

AdaBoost = "adaptive boosting"

Classes {+1 , -1}

- Pseudocode for AdaBoost

```
# Load data set X, Y … ; Y assumed +1 / -1
for b in range(num_boost):
    learner[b] = MyClassifier( X, Y, weights=wts )   # train a weighted classifier
    Yhat = learner[b].predict(X)
    e = wts.dot( Y != Yhat )                          # compute weighted error rate
    alpha[b] = 0.5 * np.log( (1-e)/e )
    wts *= np.exp( -alpha[b] * Y * Yhat )             # update weights
    wts /= wts.sum()                                  #    and normalize them
```

```
# Final classifier:
predict = np.zeros( (mTest,) )
for b in range(num_boost):
    predict += alpha[b] * learner[b].predict(Xtest)   # compute contribution of each model
predict = np.sign(predict)                            # and convert to +1 / -1 decision
```

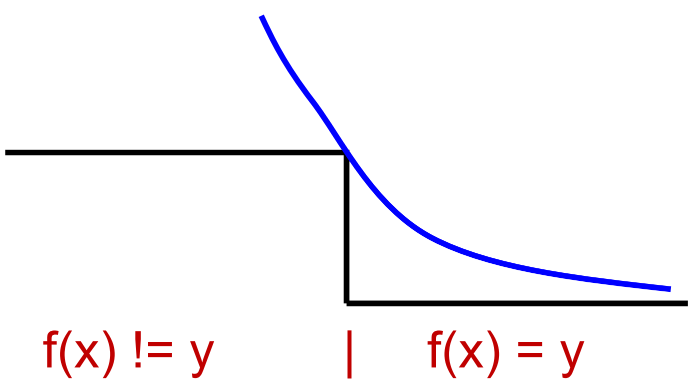
- Notes
    - e > .5 means classifier is not better than random guessing
    - Y * Yhat > 0 if Y == Yhat, and weights decrease
    - Otherwise, they increase

AdaBoost theory

- Minimizing classification error was difficult
  - For logistic regression, we minimized MSE or NLL instead
  - Idea: low MSE => low classification error
- Example of a surrogate loss function
- AdaBoost also corresponds to a surrogate loss function

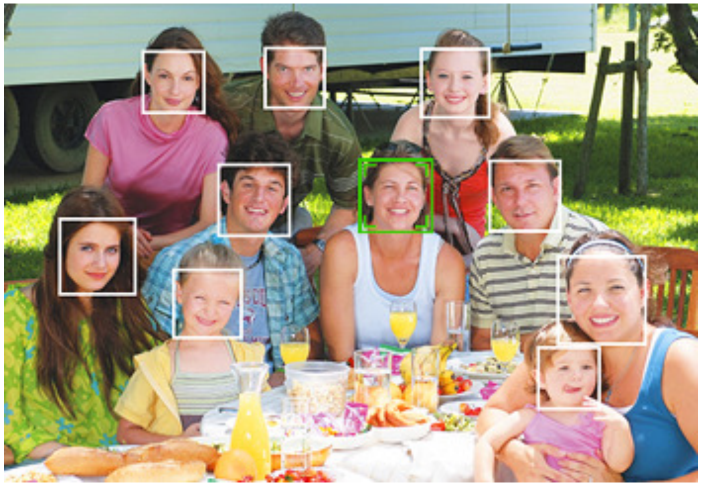
$$C_{ada} = \sum_i \exp[-y^{(i)} f(x^i)]$$

- Prediction is yhat = sign(f(x))
  - If same as y, loss < 1; if different, loss > 1; at boundary, loss=1
- This loss function is smooth & convex (easier to optimize)



AdaBoost example: Viola-Jones

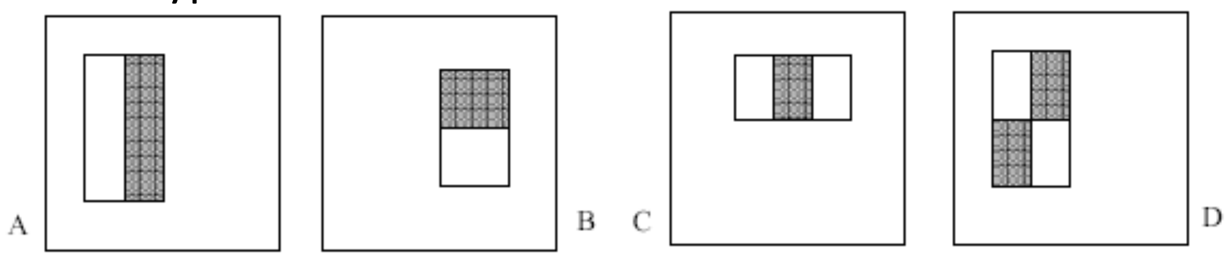
- Viola-Jones face detection algorithm
- Combine lots of very weak classifiers
  - Decision stumps = threshold on a single feature
- Define lots and lots of features
- Use AdaBoost to find good features
  - And weights for combining as well



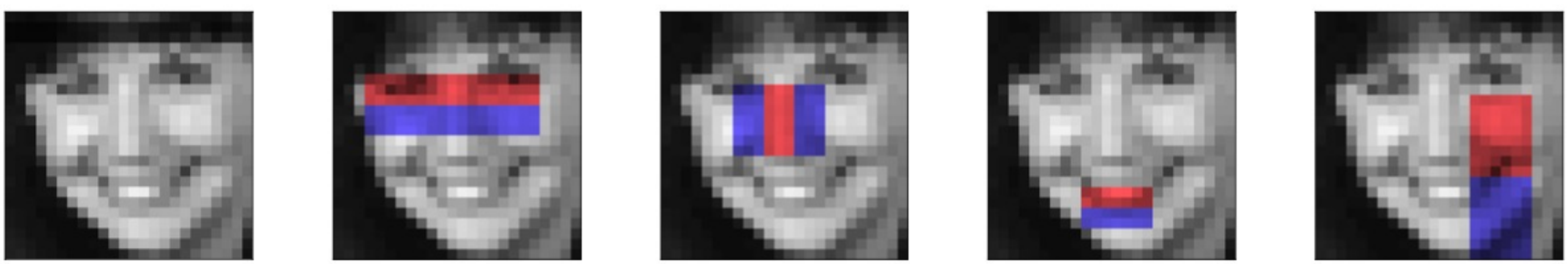
Haar wavelet features

- Four basic types.
  - They are easy to calculate.
  - The white areas are subtracted from the black ones.
  - A special representation of the sample called the **integral image** makes feature extraction faster.

Four types:



24x24 data : type, location, size => 162,336 features:



$x^{(i)}$ $\Phi_{18280}(\cdot)$ $\Phi_{126740}(\cdot)$ $\Phi_{9816}(\cdot)$ $\Phi_{36834}(\cdot)$

Training a face detector

- Wavelets give ~100k features
- Each feature is one possible classifier
- To train: iterate from 1:T
  - Train a classifier on each feature using weights
  - Choose the best one, find errors and re-weight

- This can take a long time… (lots of classifiers)
  - One way to speed up is to not train very well…
  - Rely on adaboost to fix "even weaker" classifier

- Lots of other tricks in "real" Viola-Jones
  - Cascade of decisions instead of weighted combo
  - Apply at multiple image scales
  - Work to make computationally efficient

Summary: Ensembles

- combine multiple trained models to make prediction

- Types
  - Committees: vote / average
  - Stacking: learn to combine
  - Mixtures of Experts: different "areas of expertise"
- Bagging
  - Randomly re-sample data to build diverse predictors
  - Averaging process reduces overfit of individual models
- Boosting
  - Train model to correct "remaining" mistakes
  - Combined model is more complex than individual models