ColumbiaX: Machine Learning

Lecture 13

Prof. John Paisley

Department of Electrical Engineering
& Data Science Institute

Columbia University

BOOSTING

BAGGING CLASSIFIERS

Algorithm: Bagging binary classifiers

Given $(x_1, y_1), \ldots, (x_n, y_n)$, $x \in \mathcal{X}$, $y \in \{-1, +1\}$

- For $b = 1, \ldots, B$
  - Sample a bootstrap dataset $\mathcal{B}_b$ of size n. For each entry in $\mathcal{B}_b$, select (x_i, y_i) with probability $\frac{1}{n}$. Some (x_i, y_i) will repeat and some won't appear in $\mathcal{B}_b$.
  - Learn a classifier f_b using data in $\mathcal{B}_b$.
- Define the classification rule to be

$$f_{bag}(x_0) = \text{sign}\left(\sum_{b=1}^{B} f_b(x_0)\right).$$

- With bagging, we observe that a *committee* of classifiers votes on a label.
- Each classifier is learned on a *bootstrap sample* from the data set.
- Learning a collection of classifiers is referred to as an *ensemble method*.

BOOSTING

How is it that a committee of blockheads can somehow arrive at highly reasoned decisions, despite the weak judgment of the individual members?

- Schapire & Freund, "Boosting: Foundations and Algorithms"

Boosting is another powerful method for ensemble learning. It is similar to bagging in that a set of classifiers are combined to make a better one.

It works for any classifier, but a "weak" one that is easy to learn is usually chosen. (weak = accuracy a little better than random guessing)

Short history

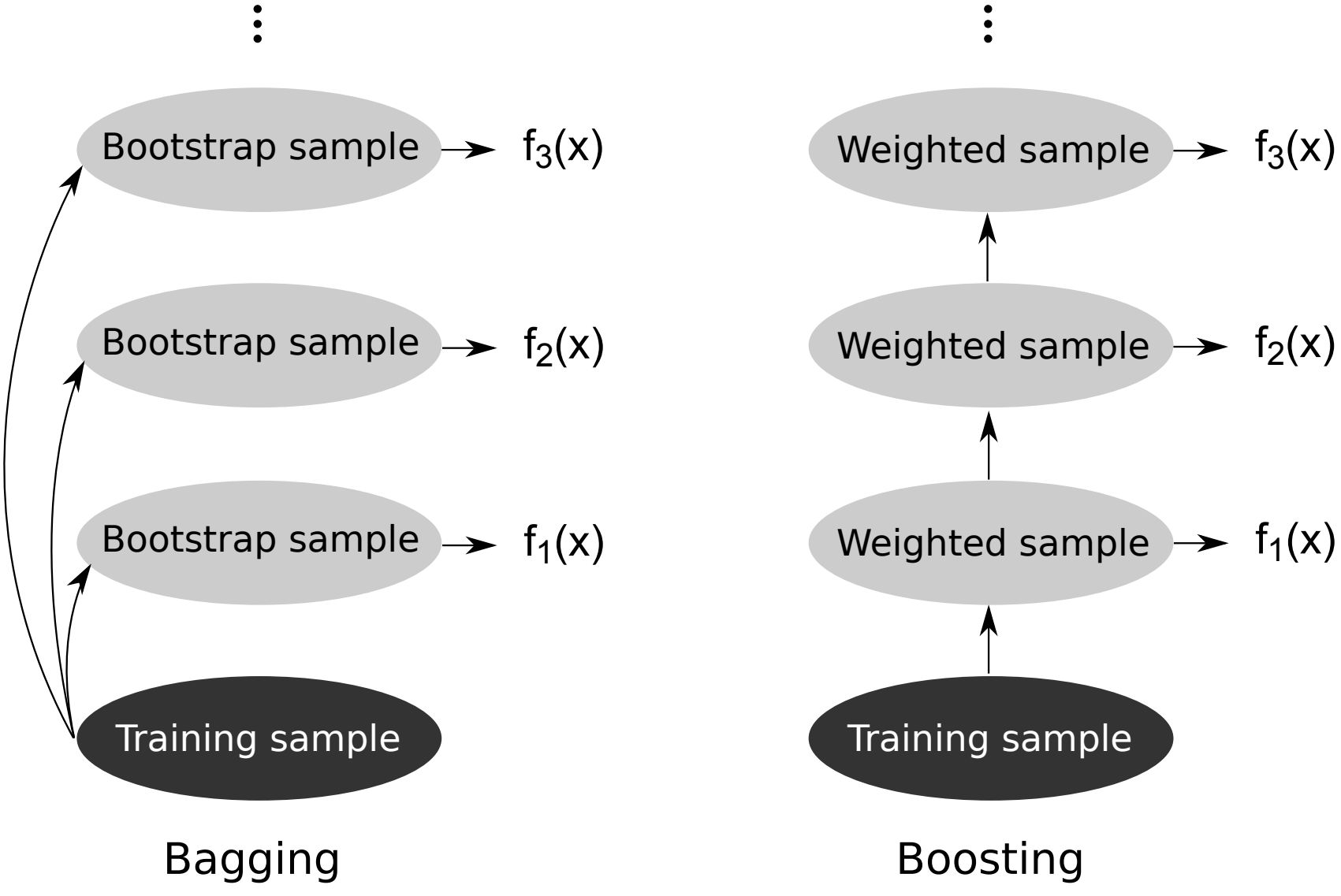
1984 : Leslie Valiant and Michael Kearns ask if "boosting" is possible.

1989 : Robert Schapire creates first boosting algorithm.

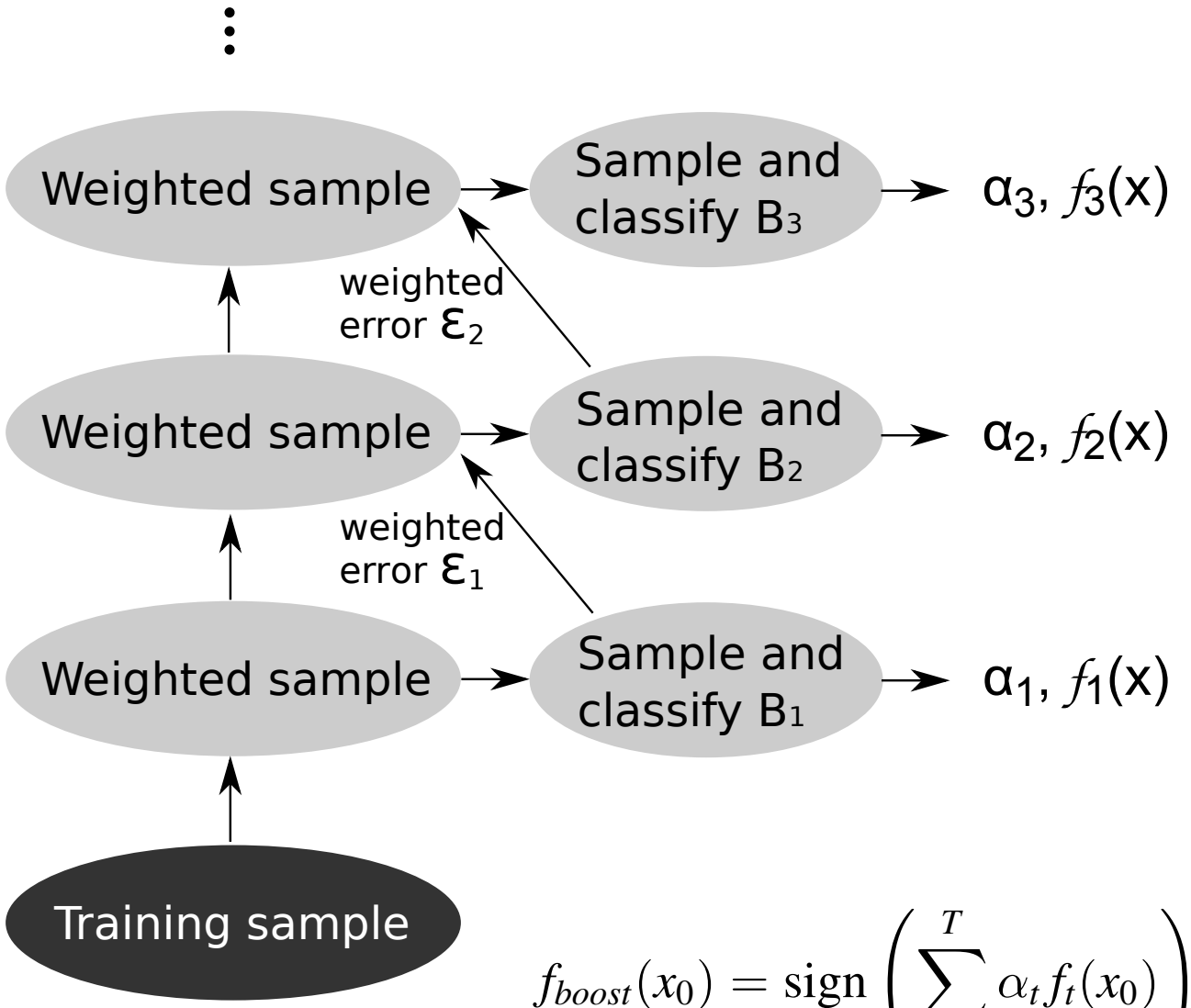
1990 : Yoav Freund creates an optimal boosting algorithm.

1995 : Freund and Schapire create AdaBoost (Adaptive Boosting), the major boosting algorithm.

Bagging vs Boosting (overview)



The AdaBoost Algorithm (sampling version)



$$f_{boost}(x_0) = \text{sign}\left(\sum_{t=1}^{T} \alpha_t f_t(x_0)\right)$$

The AdaBoost Algorithm (sampling version)

Algorithm: Boosting a binary classifier

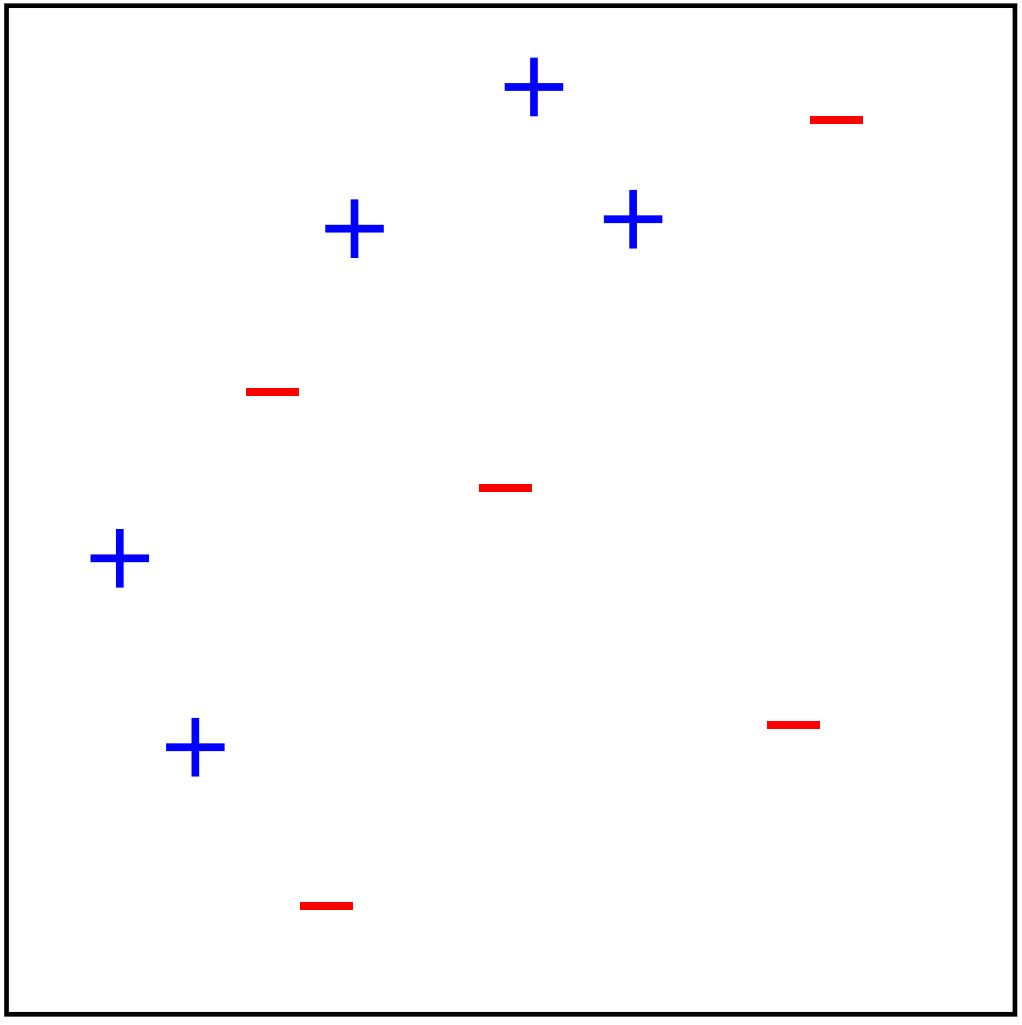
Given $(x_1, y_1), \ldots, (x_n, y_n)$, $x \in \mathcal{X}$, $y \in \{-1, +1\}$, set $w_1(i) = \frac{1}{n}$

- For $t = 1, \ldots, T$
  1. Sample a bootstrap dataset $\mathcal{B}_t$ of size n according to distribution w_t. Notice we pick (x_i, y_i) with probability $w_t(i)$ and not $\frac{1}{n}$.
  2. Learn a classifier f_t using data in $\mathcal{B}_t$.
  3. Set $\epsilon_t = \sum_{i=1}^{n} w_t(i) \mathbb{1}\{y_i \neq f_t(x_i)\}$ and $\alpha_t = \frac{1}{2} \ln\left(\frac{1-\epsilon_t}{\epsilon_t}\right)$.
  4. Scale $\hat{w}_{t+1}(i) = w_t(i) e^{-\alpha_t y_i f_t(x_i)}$ and set $w_{t+1}(i) = \dfrac{\hat{w}_{t+1}(i)}{\sum_j \hat{w}_{t+1}(j)}$.
- Set the classification rule to be

$$f_{boost}(x_0) = \text{sign}\left(\sum_{t=1}^{T} \alpha_t f_t(x_0)\right).$$

Comment: Description usually simplified to "learn classifier f_t using distribution w_t."

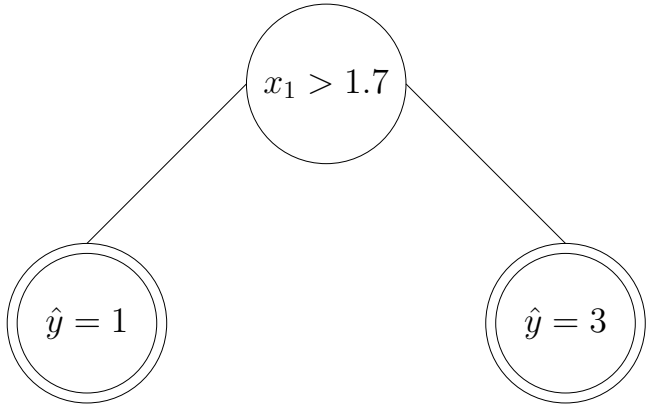
BOOSTING A DECISION STUMP (EXAMPLE 1)



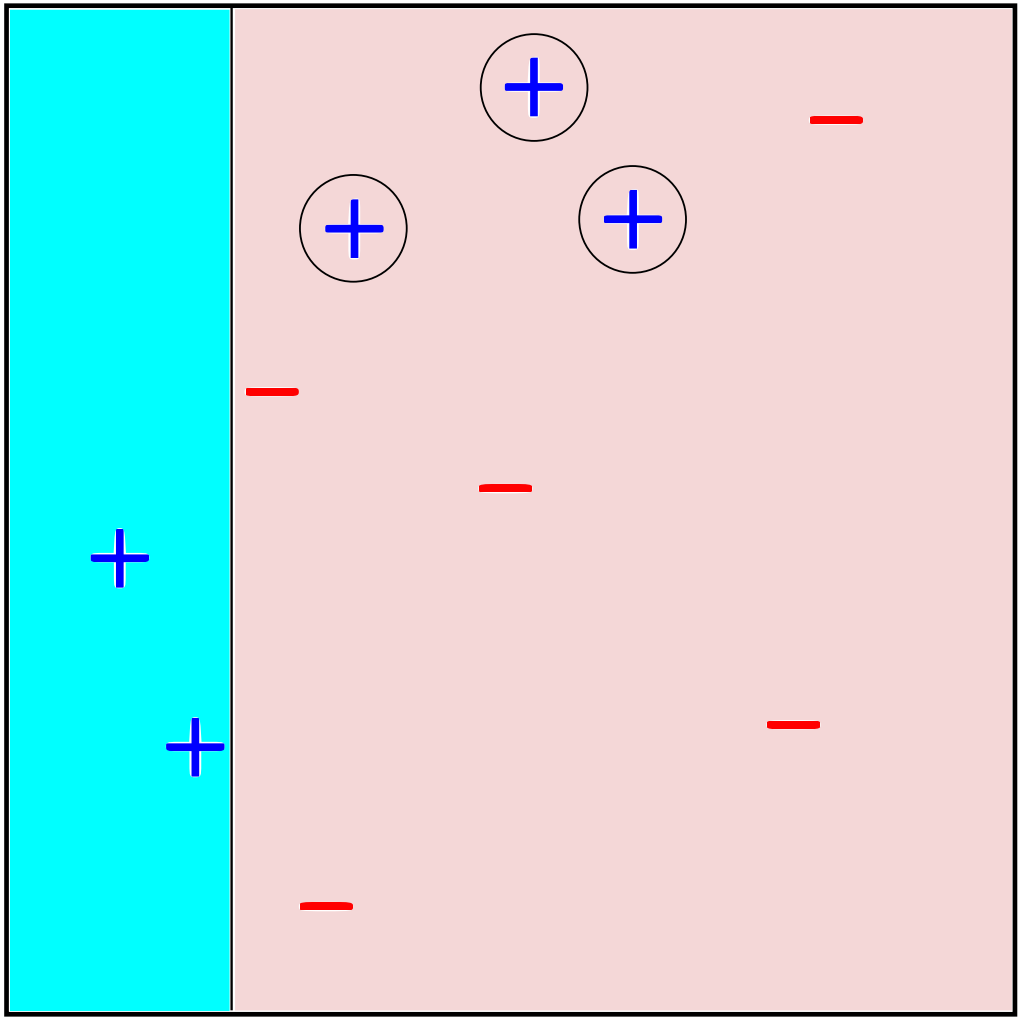
Original data

Uniform distribution, w_1
Learn *weak classifier*

Here: Use a decision stump



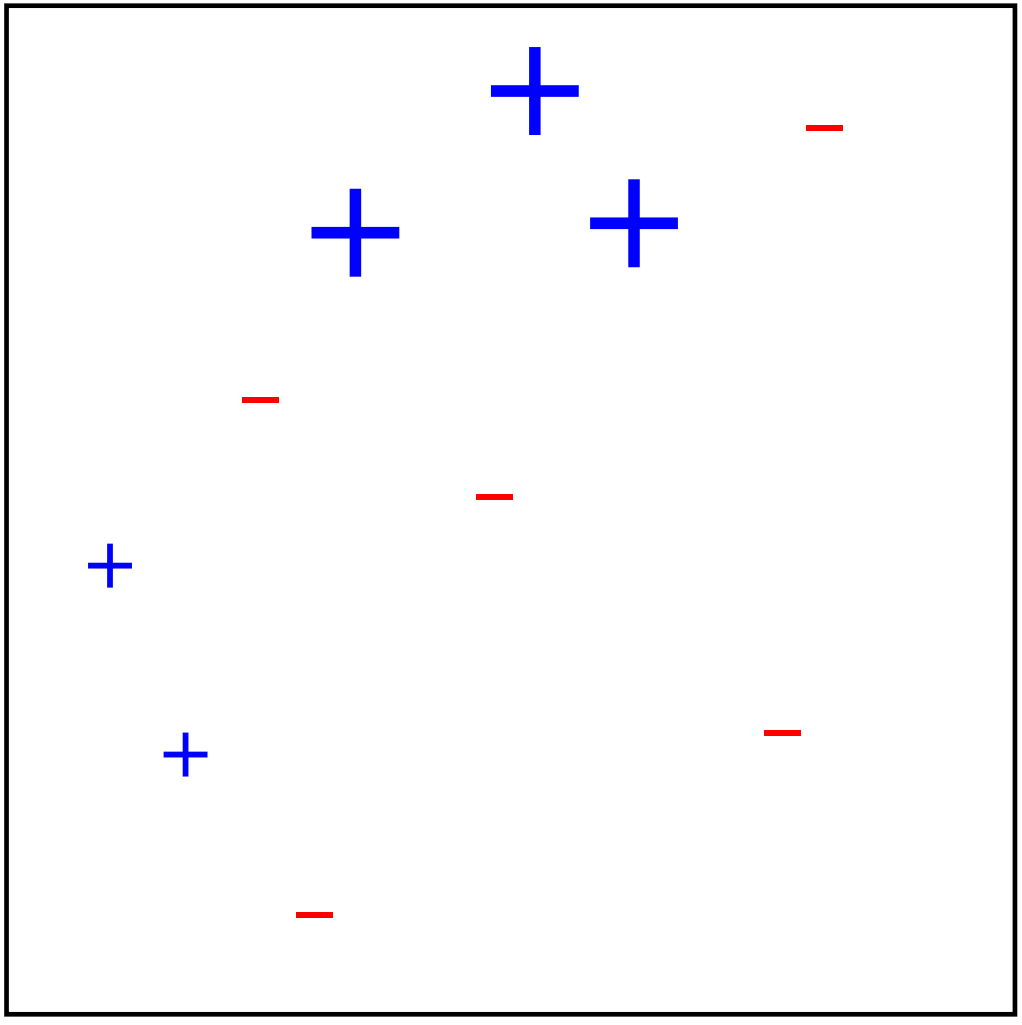
BOOSTING A DECISION STUMP (EXAMPLE 1)



Round 1 classifier

Weighted error: $\epsilon_1 = 0.3$
Weight update: $\alpha_1 = 0.42$

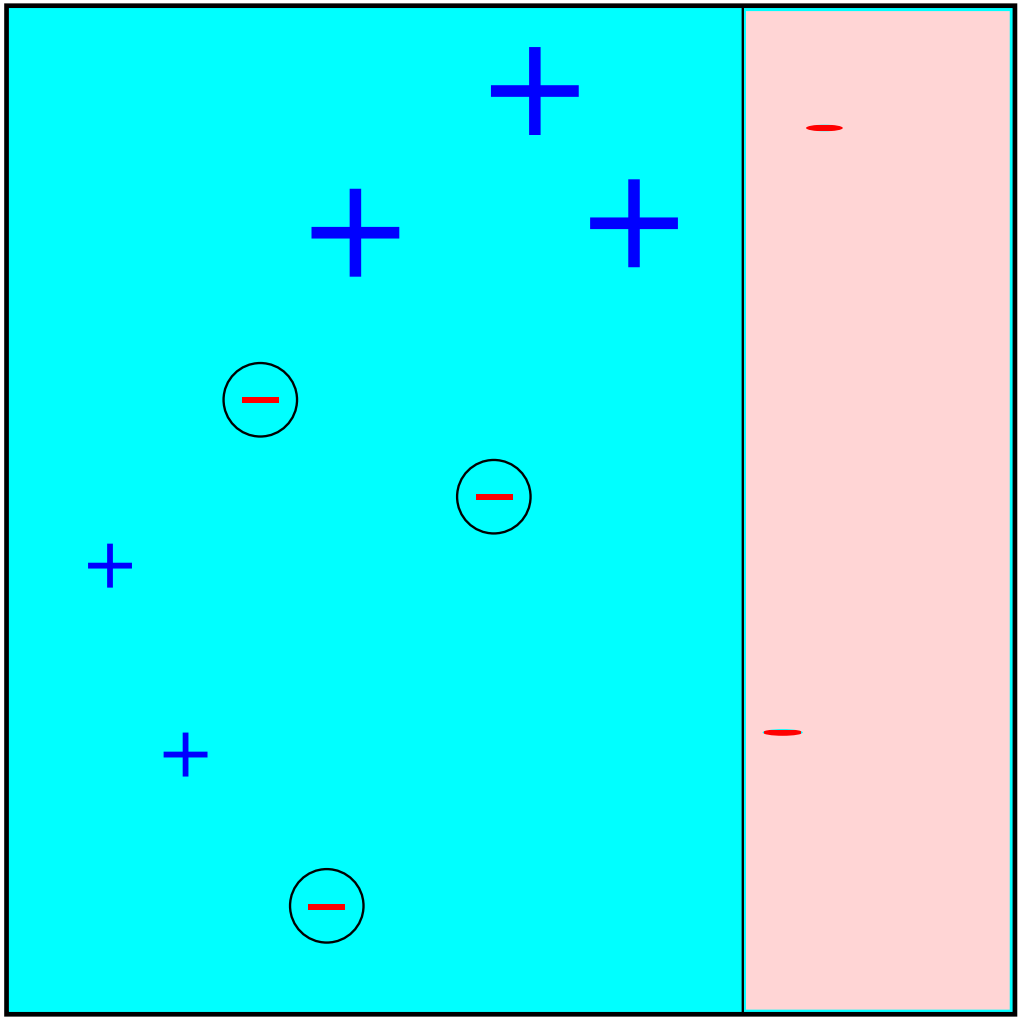
BOOSTING A DECISION STUMP (EXAMPLE 1)



Weighted data

After round 1

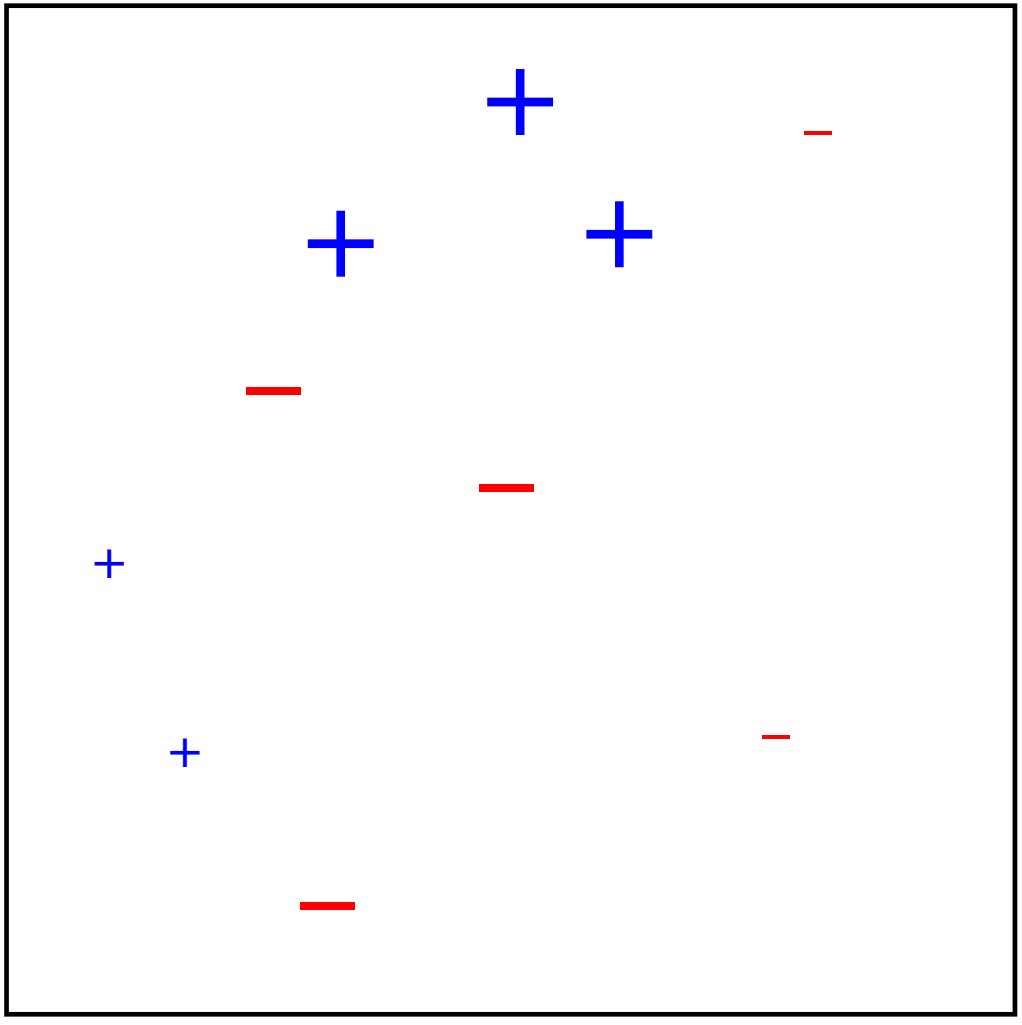
BOOSTING A DECISION STUMP (EXAMPLE 1)



Round 2 classifier

Weighted error: $\epsilon_2 = 0.21$
Weight update: $\alpha_2 = 0.65$

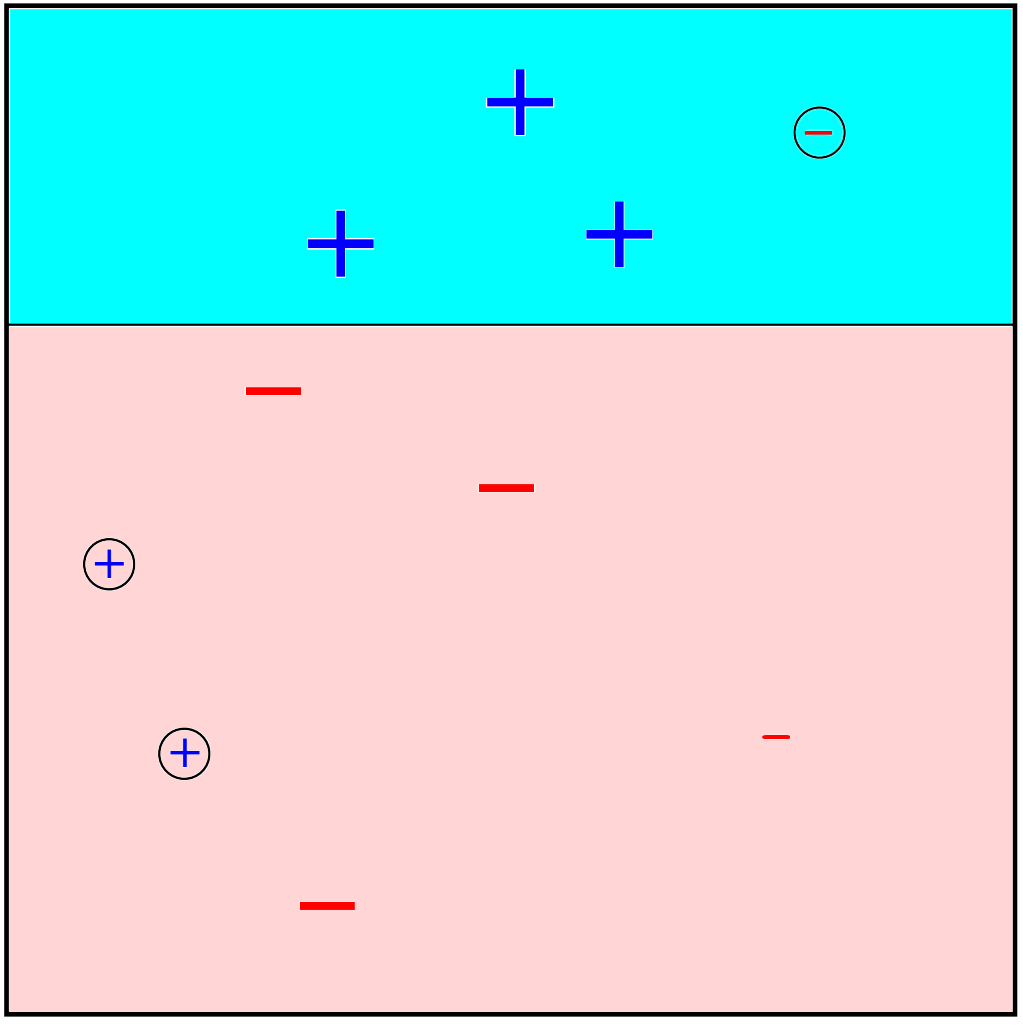
BOOSTING A DECISION STUMP (EXAMPLE 1)



Weighted data

After round 2

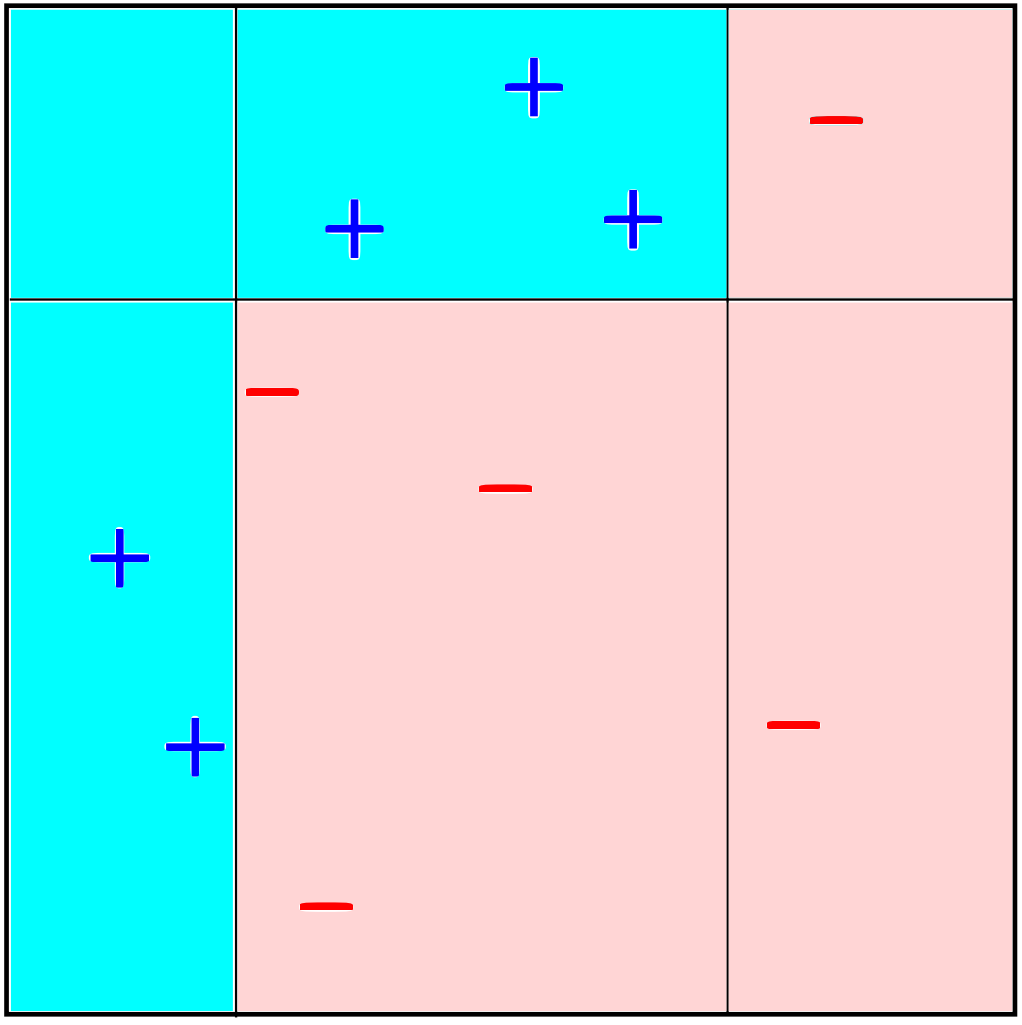
BOOSTING A DECISION STUMP (EXAMPLE 1)



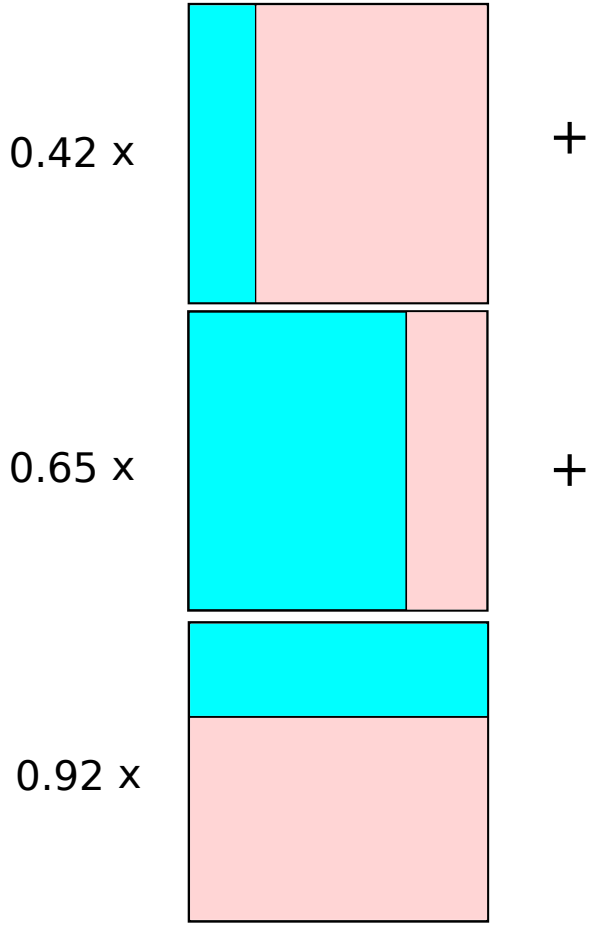
Round 2 classifier

Weighted error: $\epsilon_3 = 0.14$
Weight update: $\alpha_3 = 0.92$

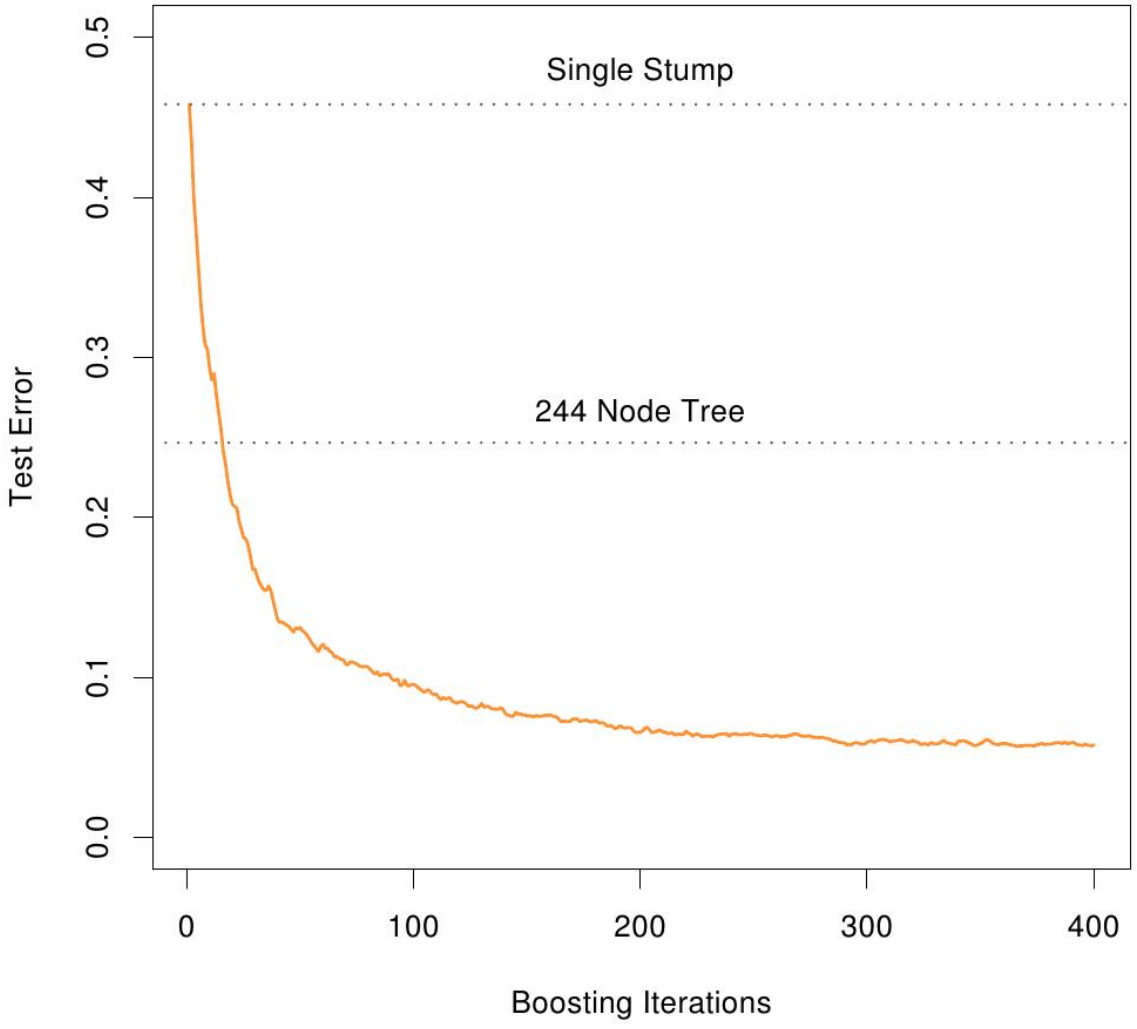
BOOSTING A DECISION STUMP (EXAMPLE 1)



Classifier after three rounds



BOOSTING A DECISION STUMP (EXAMPLE 2)



Example problem

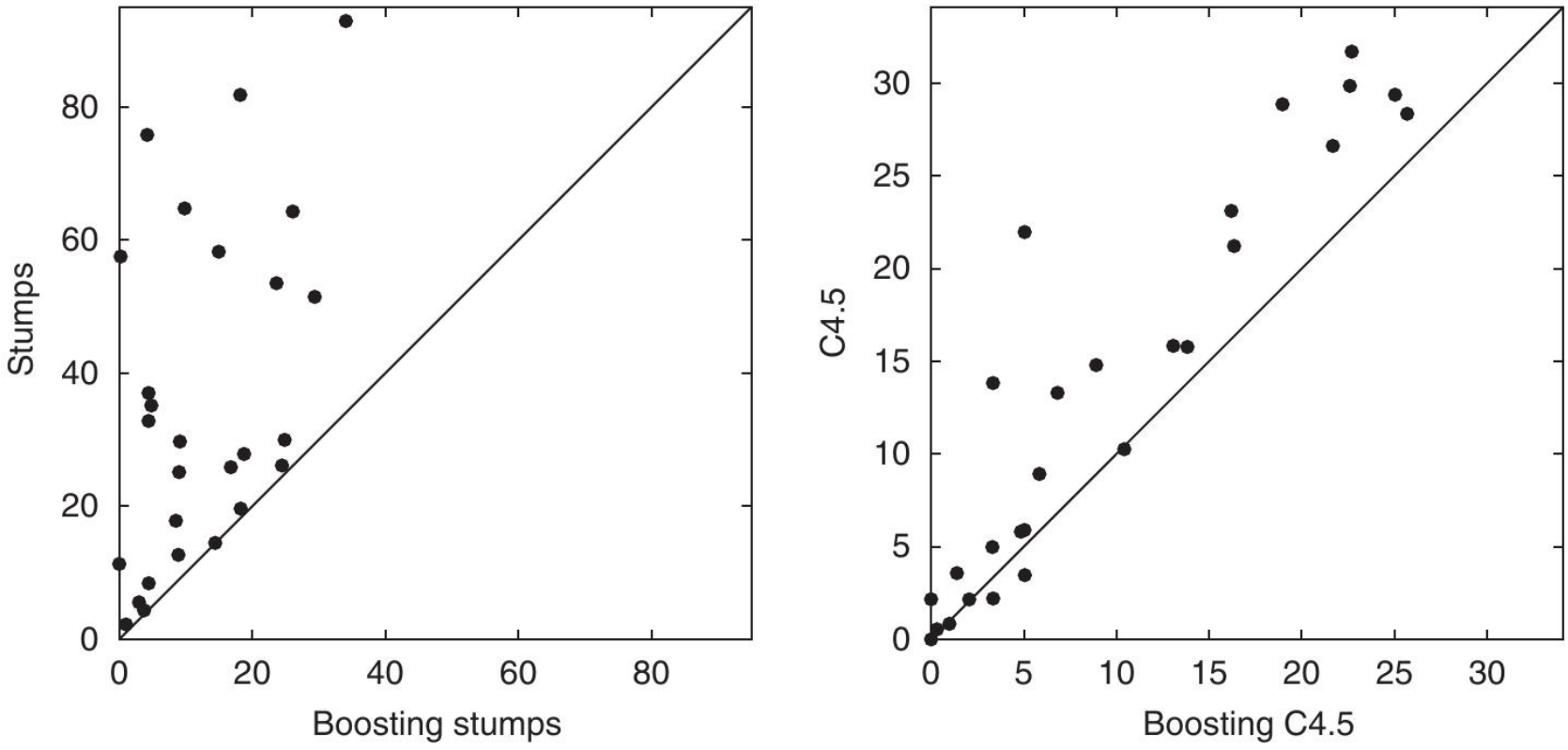
Random guessing
50% error

Decision stump
45.8% error

Full decision tree
24.7% error

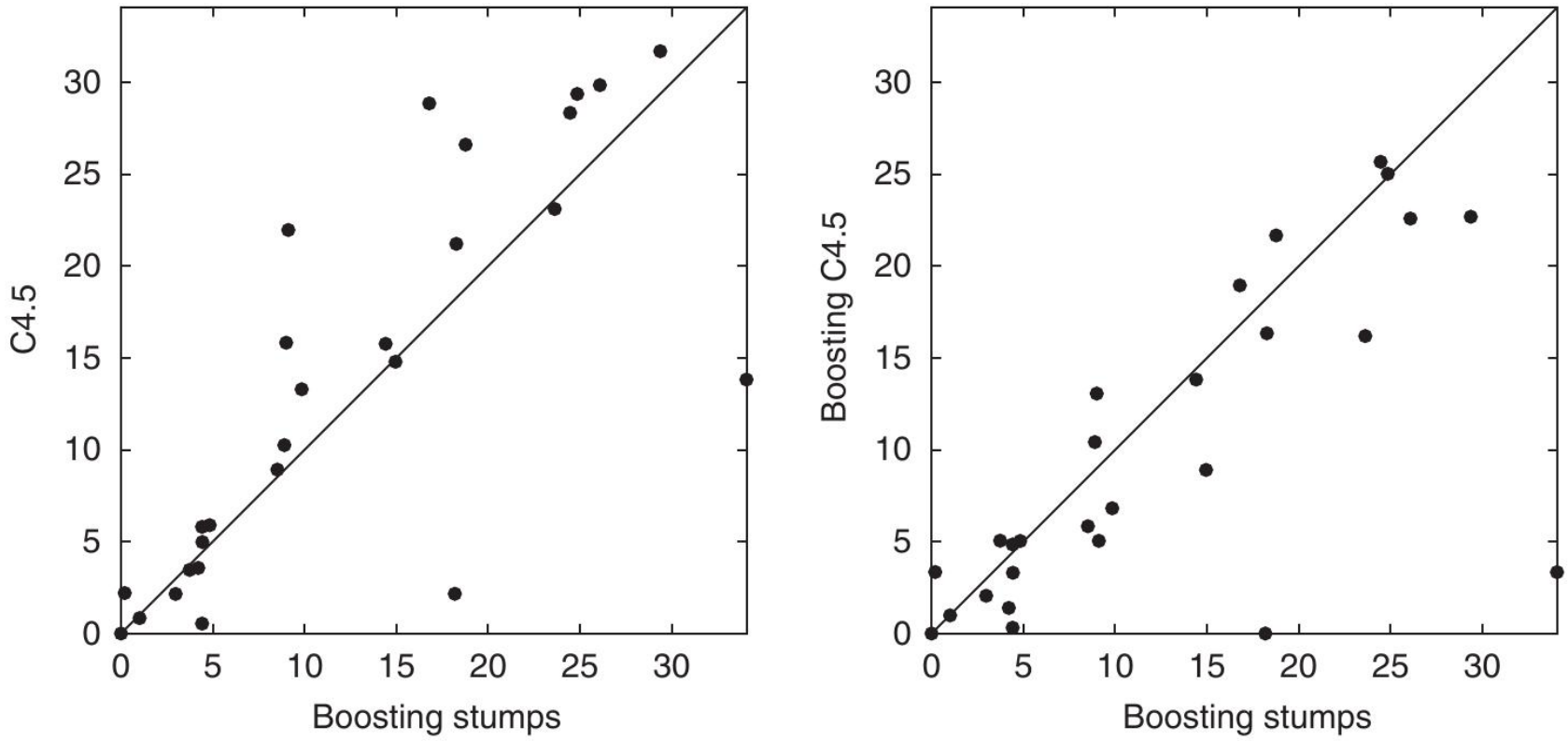
Boosted stump
5.8% error

BOOSTING



Point = one dataset. Location = error rate w/ and w/o boosting. The boosted version of the same classifier almost always produces better results.

BOOSTING



(left) Boosting a bad classifier is often better than not boosting a good one.
(right) Boosting a good classifier is often better, but can take more time.

BOOSTING AND FEATURE MAPS

Q: What makes boosting work so well?

A: This is a well-studied question. We will present one analysis later, but we can also give intuition by tying it in with what we've already learned.

The classification for a new x_0 from boosting is

$$f_{boost}(x_0) = \text{sign}\left(\sum_{t=1}^{T} \alpha_t f_t(x_0)\right).$$

Define $\phi(x) = [f_1(x), \ldots, f_T(x)]^\top$, where each $f_t(x) \in \{-1, +1\}$.

- We can think of $\phi(x)$ as a high dimensional feature map of x.
- The vector $\boldsymbol{\alpha} = [\alpha_1, \ldots, \alpha_T]^\top$ corresponds to a hyperplane.
- So the classifier can be written $f_{boost}(x_0) = \text{sign}(\phi(x_0)^\top \boldsymbol{\alpha})$.
- Boosting learns the feature mapping and hyperplane simultaneously.

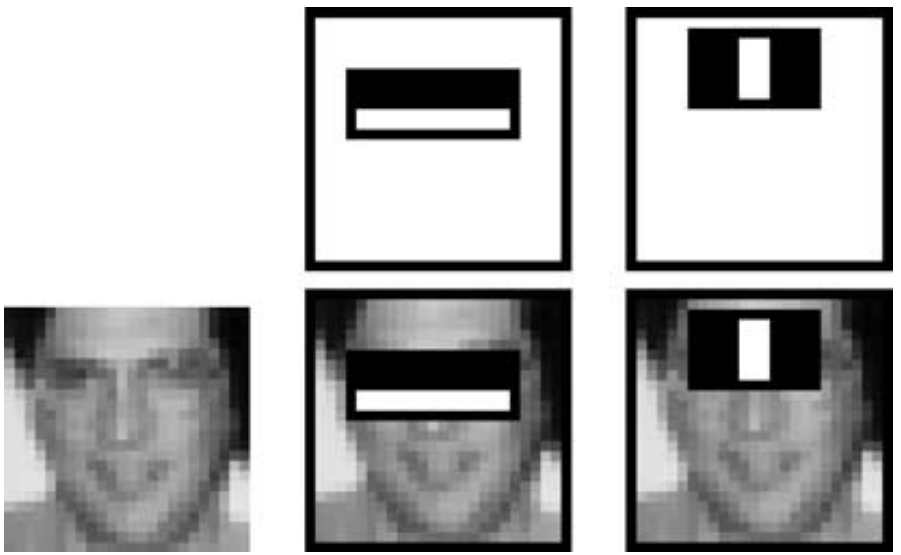
Application: Face detection

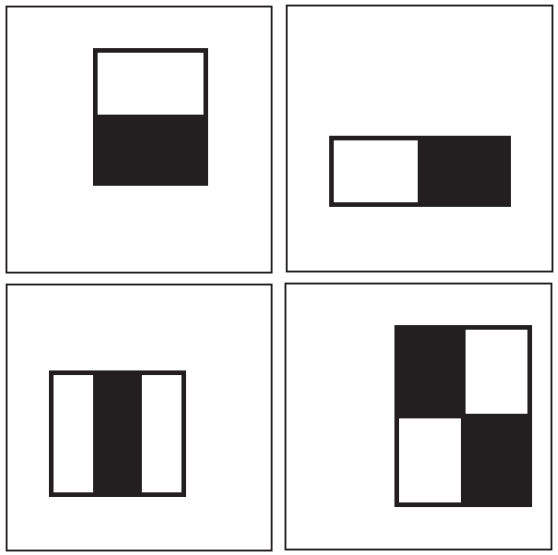
FACE DETECTION (VIOLA & JONES, 2001)

Problem: Locate the faces in an image or video.

Processing: Divide image into patches of different scales, e.g., 24×24, 48×48, etc. Extract *features* from each patch.

Classify each patch as face or no face using a *boosted decision stump*. This can be done in real-time, for example by your digital camara (at 15 fps).





- One patch from a larger image. Mask it with many "feature extractors."
- Each pattern gives one number, which is the sum of all pixels in black region minus sum of pixels in white region (total of 45,000+ features).

FACE DETECTION (EXAMPLE RESULTS)

Analysis of boosting

Training error theorem

We can use *analysis* to make a statement about the accuracy of boosting *on the training data*.

Theorem: Under the AdaBoost framework, if ϵ_t is the weighted error of classifier f_t, then for the classifier $f_{boost}(x_0) = \text{sign}(\sum_{t=1}^{T} \alpha_t f_t(x_0))$,

$$\text{training error } = \frac{1}{n}\sum_{i=1}^{n} \mathbb{1}\{y_i \neq f_{boost}(x_i)\} \leq \exp\Big(-2\sum_{t=1}^{T}(\tfrac{1}{2}-\epsilon_t)^2\Big).$$

Even if each ϵ_t is only a little better than random guessing, the sum over T classifiers can lead to a large negative value in the exponent when T is large.

For example, if we set:

$$\epsilon_t = 0.45,\ T = 1000 \rightarrow \text{ training error } \leq 0.0067.$$

PROOF OF THEOREM

Setup

We break the proof into three steps. It is an application of the fact that

$$\text{if} \quad \underbrace{a < b}_{\text{Step 2}} \quad \text{and} \quad \underbrace{b < c}_{\text{Step 3}} \quad \text{then} \quad \underbrace{a < c}_{\text{conclusion}}$$

- Step 1 calculates the value of b.
- Steps 2 and 3 prove the two inequalities.

Also recall the following step from AdaBoost:

- Update $\hat{w}_{t+1}(i) = w_t(i)e^{-\alpha_t y_i f_t(x_i)}$.
- Normalize $w_{t+1}(i) = \dfrac{\hat{w}_{t+1}(i)}{\sum_j \hat{w}_{t+1}(j)} \quad \longrightarrow \quad$ Define $Z_t = \sum_j \hat{w}_{t+1}(j)$.

PROOF OF THEOREM $(a \leq \boldsymbol{b} \leq c)$

Step 1

We first want to expand the equation of the weights to show that

$$w_{T+1}(i) = \frac{1}{n}\frac{e^{-y_i \sum_{t=1}^{T} \alpha_t f_t(x_i)}}{\prod_{t=1}^{T} Z_t} = \frac{1}{n}\frac{e^{-y_i f_{boost}^{(T)}(x_i)}}{\prod_{t=1}^{T} Z_t} \quad (f_{boost}^{(T)} \text{ is up to step } T)$$

Derivation of Step 1:

Notice the update rule: $w_{t+1}(i) = \frac{1}{Z_t} w_t(i) e^{-\alpha_t y_i f_t(x_i)}$

Do the same expansion for $w_t(i)$ and continue until reaching $w_1(i) = \frac{1}{n}$,

$$w_{T+1}(i) = w_1(i)\frac{e^{-\alpha_1 y_i f_1(x_i)}}{Z_1} \times \cdots \times \frac{e^{-\alpha_T y_i f_T(x_i)}}{Z_T}$$

The product $\prod_{t=1}^{T} Z_t$ is "b" above. We use this form of $w_{T+1}(i)$ in Step 2.

PROOF OF THEOREM $(a \leq b \leq c)$

Step 2

Next show that the training error of $f_{boost}^{(T)}$ after T steps is $\leq \prod_{t=1}^{T} Z_t$.

From Step 1: $w_{T+1}(i) = \frac{1}{n} \frac{e^{-y_i f_{boost}^{(T)}(x_i)}}{\prod_{t=1}^{T} Z_t} \implies w_{T+1}(i) \prod_{t=1}^{T} Z_t = \frac{1}{n} e^{-y_i f_{boost}^{(T)}(x_i)}$

Derivation of Step 2:

(Observe that $0 < e^{z_1}$ and $1 < e^{z_2}$ for any $z_1 < 0 < z_2$.)

$$\begin{aligned} \frac{1}{n} \sum_{i=1}^{n} \mathbb{1}\{y_i \neq f_{boost}^{(T)}(x_i)\} &\leq \frac{1}{n} \sum_{i=1}^{n} e^{-y_i f_{boost}^{(T)}(x_i)} \\ &= \sum_{i=1}^{n} w_{T+1}(i) \prod_{t=1}^{T} Z_t \\ &= \prod_{t=1}^{T} Z_t \end{aligned}$$

"a" is the training error – the quantity we care about.

PROOF OF THEOREM $(a \leq b \leq c)$

Step 3

The final step is to calculate an upper bound on Z_t, and by extension $\prod_{t=1}^{T} Z_t$.

Derivation of Step 3:

This step is slightly more involved. It also shows why $\alpha_t := \frac{1}{2} \ln \left(\frac{1-\epsilon_t}{\epsilon_t} \right)$.

$$\begin{aligned} Z_t &= \sum_{i=1}^{n} w_t(i) e^{-\alpha_t y_i f_t(x_i)} \\ &= \sum_{i\,:\,y_i = f_t(x_i)} e^{-\alpha_t} w_t(i) + \sum_{i\,:\,y_i \neq f_t(x_i)} e^{\alpha_t} w_t(i) \\ &= e^{-\alpha_t}(1-\epsilon_t) + e^{\alpha_t}\epsilon_t \end{aligned}$$

Remember we *defined* $\epsilon_t = \sum_{i\,:\,y_i \neq f_t(x_i)} w_t(i)$, the probability of error for w_t.

PROOF OF THEOREM $(a \leq \boldsymbol{b} \leq \boldsymbol{c})$

Derivation of Step 3 (continued):
Remember from Step 2 that

$$\text{training error} \;=\; \frac{1}{n}\sum_{i=1}^{n} \mathbb{1}\{y_i \neq f_{boost}(x_i)\} \;\leq\; \prod_{t=1}^{T} Z_t \,.$$

and we just showed that $Z_t = e^{-\alpha_t}(1-\epsilon_t) + e^{\alpha_t}\epsilon_t$.

We want the training error to be small, so we pick α_t to *minimize* Z_t.
Minimizing, we get the value of α_t used by AdaBoost:

$$\alpha_t = \frac{1}{2}\ln\left(\frac{1-\epsilon_t}{\epsilon_t}\right).$$

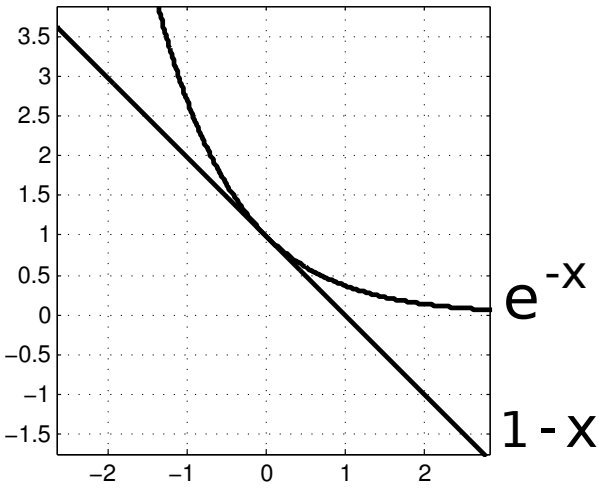
Plugging this value back in gives $Z_t = 2\sqrt{\epsilon_t(1-\epsilon_t)}$.

PROOF OF THEOREM $(a \leq b \leq c)$

Derivation of Step 3 (continued):

Next, re-write Z_t as

$$\begin{aligned} Z_t &= 2\sqrt{\epsilon_t(1-\epsilon_t)} \\ &= \sqrt{1-4(\frac{1}{2}-\epsilon_t)^2} \end{aligned}$$



Then, use the inequality $1 - x \leq e^{-x}$ to conclude that

$$Z_t = \left(1-4(\tfrac{1}{2}-\epsilon_t)^2\right)^{\frac{1}{2}} \leq \left(e^{-4(\frac{1}{2}-\epsilon_t)^2}\right)^{\frac{1}{2}} = e^{-2(\frac{1}{2}-\epsilon_t)^2}.$$

Concluding the right inequality $(a \leq \boldsymbol{b} \leq \boldsymbol{c})$

Because both sides of $Z_t \leq e^{-2(\frac{1}{2}-\epsilon_t)^2}$ are positive, we can say that

$$\prod_{t=1}^{T} Z_t \;\leq\; \prod_{t=1}^{T} e^{-2(\frac{1}{2}-\epsilon_t)^2} = e^{-2\sum_{t=1}^{T}(\frac{1}{2}-\epsilon_t)^2}.$$

This concludes the "$b \leq c$" portion of the proof.

Combining everything

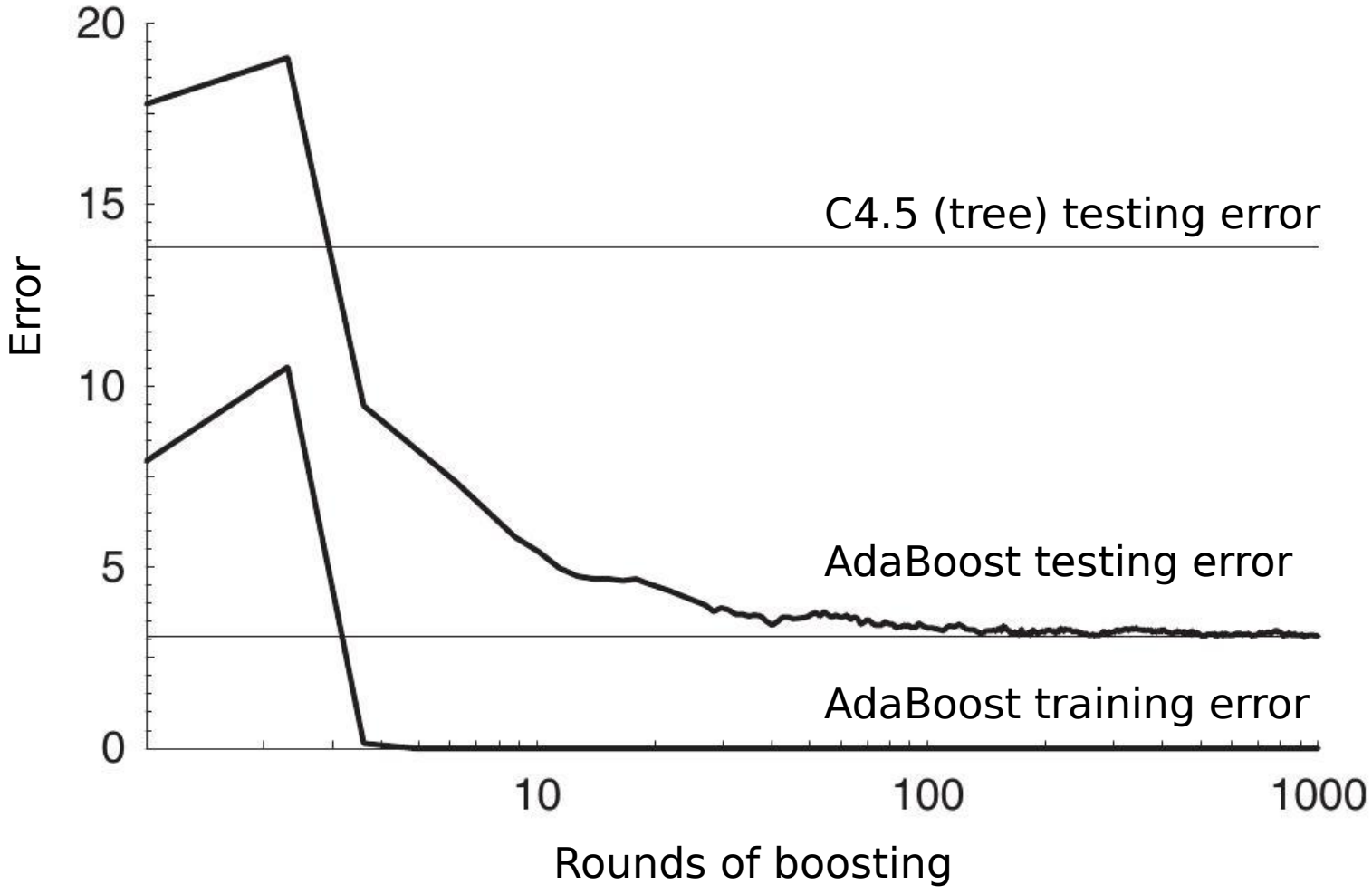
$$\text{training error} \;=\; \overbrace{\frac{1}{n}\sum_{i=1}^{n} \mathbb{1}\{y_i \neq f_{boost}(x_i)\}}^{a} \;\leq\; \overbrace{\prod_{t=1}^{T} Z_t}^{b} \;\leq\; \overbrace{e^{-2\sum_{t=1}^{T}(\frac{1}{2}-\epsilon_t)^2}}^{c}.$$

We set out to prove "$a < c$" and we did so by using "b" as a stepping-stone.

TRAINING VS TESTING ERROR

Q: Driving the training error to zero leads one to ask, does boosting overfit?

A: Sometimes, but very often it doesn't!



ColumbiaX: Machine Learning

Lecture 14

Prof. John Paisley

Department of Electrical Engineering
& Data Science Institute

Columbia University

Unsupervised Learning

Framework of supervised learning

Given: Pairs $(x_1, y_1), \ldots, (x_n, y_n)$. Think of x as input and y as output.

Learn: A function $f(x)$ that accurately predicts $y_i \approx f(x_i)$ on this data.

Goal: Use the function $f(x)$ to predict new y_0 given x_0.

Probabilistic motivation

If we think of (x, y) as a random variable with joint distribution $p(x, y)$, then supervised learning seeks to learn the conditional distribution $p(y|x)$.

This can be done either directly or indirectly:

Directly: e.g., with logistic regression where $p(y|x) =$ sigmoid function

Indirectly: e.g., with a Bayes classifier

$$y = \arg\max_k \; p(y = k|x) = \arg\max_k \; p(x|y = k)p(y = k)$$

UNSUPERVISED LEARNING

Some motivation

- The Bayes classifier factorizes the joint density as $p(x, y) = p(x|y)p(y)$.
- The joint density can also be written as $p(x, y) = p(y|x)p(x)$.
- *Unsupervised learning* focuses on the term $p(x)$ — learning $p(x|y)$ on a class-specific subset has the same "feel." What should this be?
- This implies an underlying classification task, but often there isn't one.

Unsupervised learning

Given: A data set $x_1, \ldots, x_n$, where $x_i \in \mathcal{X}$, e.g., $\mathcal{X} = \mathbb{R}^d$

Define: Some model of the data (probabilistic or non-probabilistic).

Goal: Learn structure within the data set *as defined by the model*.

- Supervised learning has a clear performance metric: accuracy
- Unsupervised learning is often (but not always) more subjective

SOME TYPES OF UNSUPERVISED LEARNING

Overview of second half of course

We will discuss a few types of unsupervised learning approaches in the second half of the course.

Clustering models: Learn a partition of data $x_1, \ldots, x_n$ into groups.

- Image segmentation, data quantization, preprocessing for other models

Matrix factorization: Learn an underlying dot-product representation.

- User preference modeling, topic modeling

Sequential models: Learn a model based on sequential information.

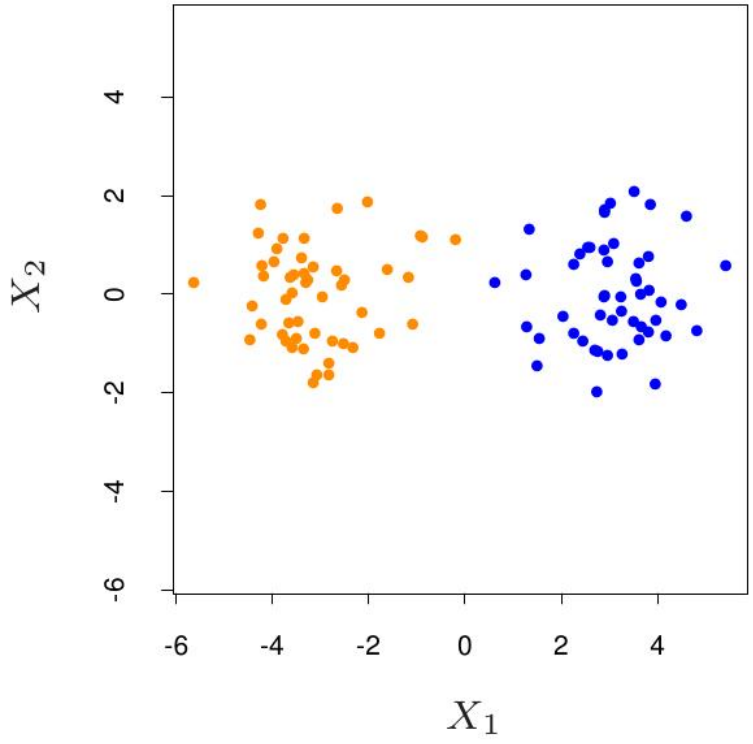
- Learn how to rank objects, target tracking

As will become evident, an unsupervised model can often be interpreted as a supervised model, or very easily turned into one.

CLUSTERING

Problem

- Given data $x_1, \ldots, x_n$, partition it into groups called *clusters*.
- Find the clusters, given only the data.
- Observations in same group $\Rightarrow$ "similar," different groups $\Rightarrow$ "different."
- We will set how many clusters we learn.



Cluster assignment representation

For K clusters, encode cluster assignments as an indicator $c \in \{1, \ldots, K\}$,

$$c_i = k \quad \Longleftrightarrow \quad x_i \text{ is assigned to cluster } k$$

Clustering feels similar to classification in that we "label" an observation by its cluster assignment. The difference is that there is no ground truth.

THE K-MEANS ALGORITHM

CLUSTERING AND K-MEANS

K-means is the simplest and most fundamental clustering algorithm.

Input: $x_1, \ldots, x_n$, where $x \in \mathbb{R}^d$.

Output: Vector $\boldsymbol{c}$ of cluster assignments, and K mean vectors $\boldsymbol{\mu}$

- $\boldsymbol{c} = (c_1, \ldots, c_n), \quad c_i \in \{1, \ldots, K\}$
  - If $c_i = c_j = k$, then x_i and x_j are *clustered together* in cluster k.
- $\boldsymbol{\mu} = (\mu_1, \ldots, \mu_K), \quad \mu_k \in \mathbb{R}^d$ (same space as x_i)
  - Each μ_k (called a *centroid*) defines a cluster.

As usual, we need to define an *objective function*. We pick one that:

1. Tells us what are good $\boldsymbol{c}$ and $\boldsymbol{\mu}$, and
2. That is easy to optimize.

K-MEANS OBJECTIVE FUNCTION

The K-means objective function can be written as

$$\boldsymbol{\mu}^*, \boldsymbol{c}^* = \arg\min_{\boldsymbol{\mu}, \boldsymbol{c}} \sum_{i=1}^{n} \sum_{k=1}^{K} \mathbb{1}\{c_i = k\} \|x_i - \mu_k\|^2$$

Some observations:

- K-means uses the squared Euclidean distance of x_i to the centroid μ_k.
- It only penalizes the distance of x_i to the centroid it's assigned to by c_i.

$$\mathcal{L} = \sum_{i=1}^{n} \sum_{k=1}^{K} \mathbb{1}\{c_i = k\} \|x_i - \mu_k\|^2 = \sum_{k=1}^{K} \sum_{i:c_i=k} \|x_i - \mu_k\|^2$$

- The objective function is "non-convex"
  - This means that we can't actually find the *optimal* $\boldsymbol{\mu}^*$ and $\boldsymbol{c}^*$.
  - We can only derive an *algorithm* for finding a *local optimum* (more later).

Optimizing the K-means objective

Gradient-based optimization

We can't optimize the K-means objective function exactly by taking derivatives and setting to zero, so we use an iterative algorithm.

However, the algorithm we will use is different from gradient methods:

$$w \leftarrow w - \eta \nabla_w \mathcal{L} \qquad \text{(gradient descent)}$$

Recall: With gradient descent, when we update a parameter "w" we move in the direction that decreases the objective function, but

- It will almost certainly not move to the *best* value for that parameter.
- It may not even move to a better value if the step size η is too big.
- We also need the parameter w to be continuous-valued.

Coordinate descent

We will discuss a new and widely used optimization procedure in the context of K-means clustering. We want to minimize the objective function

$$\mathcal{L} = \sum_{i=1}^{n} \sum_{k=1}^{K} \mathbb{1}\{c_i = k\} \|x_i - \mu_k\|^2.$$

We split the variables into two unknown sets $\boldsymbol{\mu}$ and $\boldsymbol{c}$. We can't find their best values *at the same time* to minimize $\mathcal{L}$. However, we will see that

- Fixing $\boldsymbol{\mu}$ we can find the best $\boldsymbol{c}$ exactly.
- Fixing $\boldsymbol{c}$ we can find the best $\boldsymbol{\mu}$ exactly.

This optimization approach is called *coordinate descent*: Hold one set of parameters fixed, and optimize the other set. Then switch which set is fixed.

Coordinate descent (in the context of K-means)

Input: $x_1, \ldots, x_n$ where $x_i \in \mathbb{R}^d$. Randomly initialize $\boldsymbol{\mu} = (\mu_1, \ldots, \mu_K)$.

- Iterate back-and-forth between the following two steps:
  1. Given $\boldsymbol{\mu}$, find the best value $c_i \in \{1, \ldots, K\}$ for $i = 1, \ldots, n$.
  2. Given $\boldsymbol{c}$, find the best vector $\mu_k \in \mathbb{R}^d$ for $k = 1, \ldots, K$.

There's a circular way of thinking about why we need to iterate:

1. Given a particular $\boldsymbol{\mu}$, we may be able to find *the best* $\boldsymbol{c}$, but once we *change* $\boldsymbol{c}$ we can probably find a better $\boldsymbol{\mu}$.
2. Then find *the best* $\boldsymbol{\mu}$ for the new-and-improved $\boldsymbol{c}$ found in #1, but now that we've changed $\boldsymbol{\mu}$, there is probably a better $\boldsymbol{c}$.

We have to iterate because the values of $\boldsymbol{\mu}$ and $\boldsymbol{c}$ *depend on each other*. This happens very frequently in unsupervised models.

K-MEANS ALGORITHM: UPDATING $\boldsymbol{c}$

Assignment step

Given $\boldsymbol{\mu} = (\mu_1, \ldots, \mu_K)$, update $\boldsymbol{c} = (c_1, \ldots, c_n)$. By rewriting $\mathcal{L}$, we notice the independence of each c_i given $\boldsymbol{\mu}$,

$$\mathcal{L} = \underbrace{\left(\sum_{k=1}^{K} \mathbb{1}\{c_1 = k\} \|x_1 - \mu_k\|^2 \right)}_{\text{distance of } x_1 \text{ to its assigned centroid}} + \cdots + \underbrace{\left(\sum_{k=1}^{K} \mathbb{1}\{c_n = k\} \|x_n - \mu_k\|^2 \right)}_{\text{distance of } x_n \text{ to its assigned centroid}}.$$

We can minimize $\mathcal{L}$ with respect to each c_i by minimizing each term above separately. The solution is to assign x_i to the closest centroid

$$c_i = \arg\min_k \ \|x_i - \mu_k\|^2.$$

Because there are only K options for each c_i, there are no derivatives. Simply calculate all the possible values for c_i and pick the best (smallest) one.

K-means algorithm: Updating $\boldsymbol{\mu}$

Update step

Given $\boldsymbol{c} = (c_1, \ldots, c_n)$, update $\boldsymbol{\mu} = (\mu_1, \ldots, \mu_K)$. For a given $\boldsymbol{c}$, we can break $\mathcal{L}$ into K clusters defined by $\boldsymbol{c}$ so that each μ_i is independent.

$$\mathcal{L} = \underbrace{\left(\sum_{i=1}^{N} \mathbb{1}\{c_i = 1\} \|x_i - \mu_1\|^2 \right)}_{\text{sum squared distance of data in cluster \#1}} + \cdots + \underbrace{\left(\sum_{i=1}^{N} \mathbb{1}\{c_i = K\} \|x_i - \mu_K\|^2 \right)}_{\text{sum squared distance of data in cluster \#}K}.$$

For each k, we then optimize. Let $n_k = \sum_{i=1}^{n} \mathbb{1}\{c_i = k\}$. Then

$$\mu_k = \arg\min_{\mu} \sum_{i=1}^{n} \mathbb{1}\{c_i = k\} \|x_i - \mu\|^2 \quad \longrightarrow \quad \mu_k = \frac{1}{n_k} \sum_{i=1}^{n} x_i \mathbb{1}\{c_i = k\}.$$

That is, μ_k is the *mean* of the data assigned to cluster k.

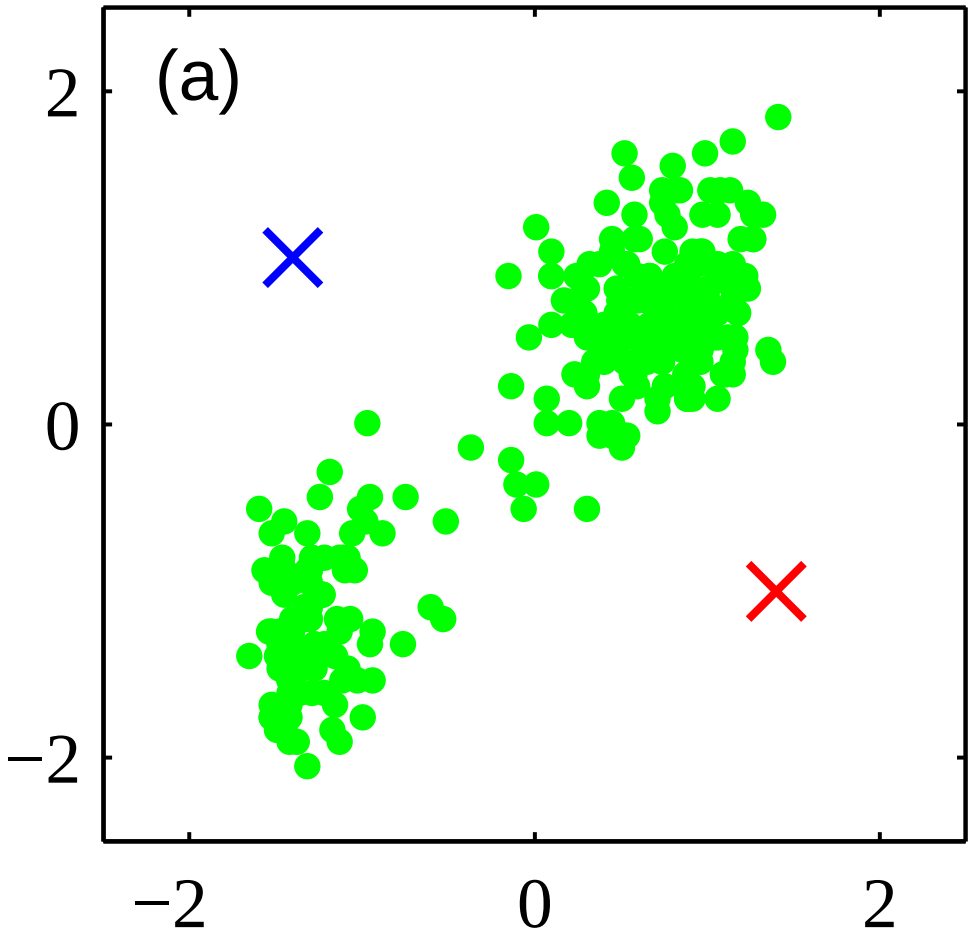
Algorithm: K-means clustering

Given: $x_1, \ldots, x_n$ where each $x \in \mathbb{R}^d$

Goal: Minimize $\mathcal{L} = \sum_{i=1}^{n} \sum_{k=1}^{K} \mathbb{1}\{c_i = k\} \|x_i - \mu_k\|^2$.

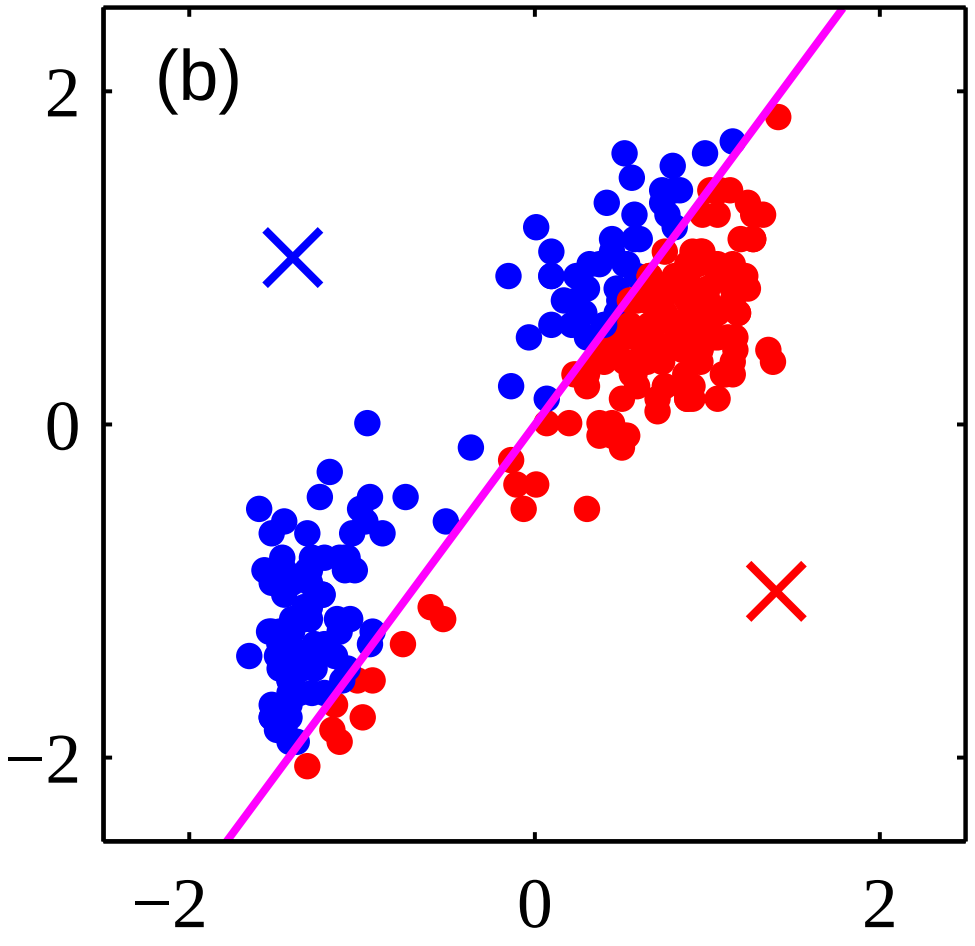
- Randomly initialize $\boldsymbol{\mu} = (\mu_1, \ldots, \mu_K)$.
- Iterate until $\boldsymbol{c}$ and $\boldsymbol{\mu}$ stop changing
  1. Update each c_i :
  $$c_i = \arg\min_k \|x_i - \mu_k\|^2$$
  2. Update each μ_k : Set
  $$n_k = \sum_{i=1}^{n} \mathbb{1}\{c_i = k\} \quad \text{and} \quad \mu_k = \frac{1}{n_k} \sum_{i=1}^{n} x_i \mathbb{1}\{c_i = k\}$$

K-means algorithm: Example run



A random initialization

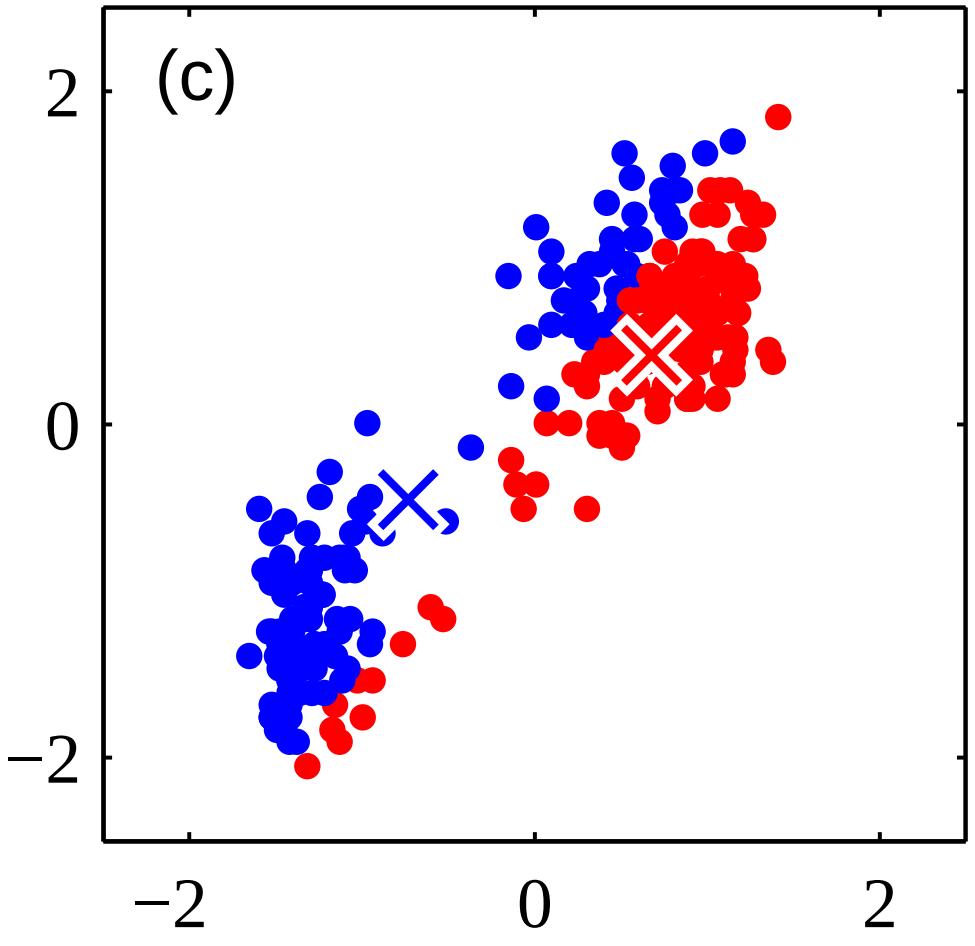
K-means algorithm: Example run



Iteration 1

Assign data to clusters

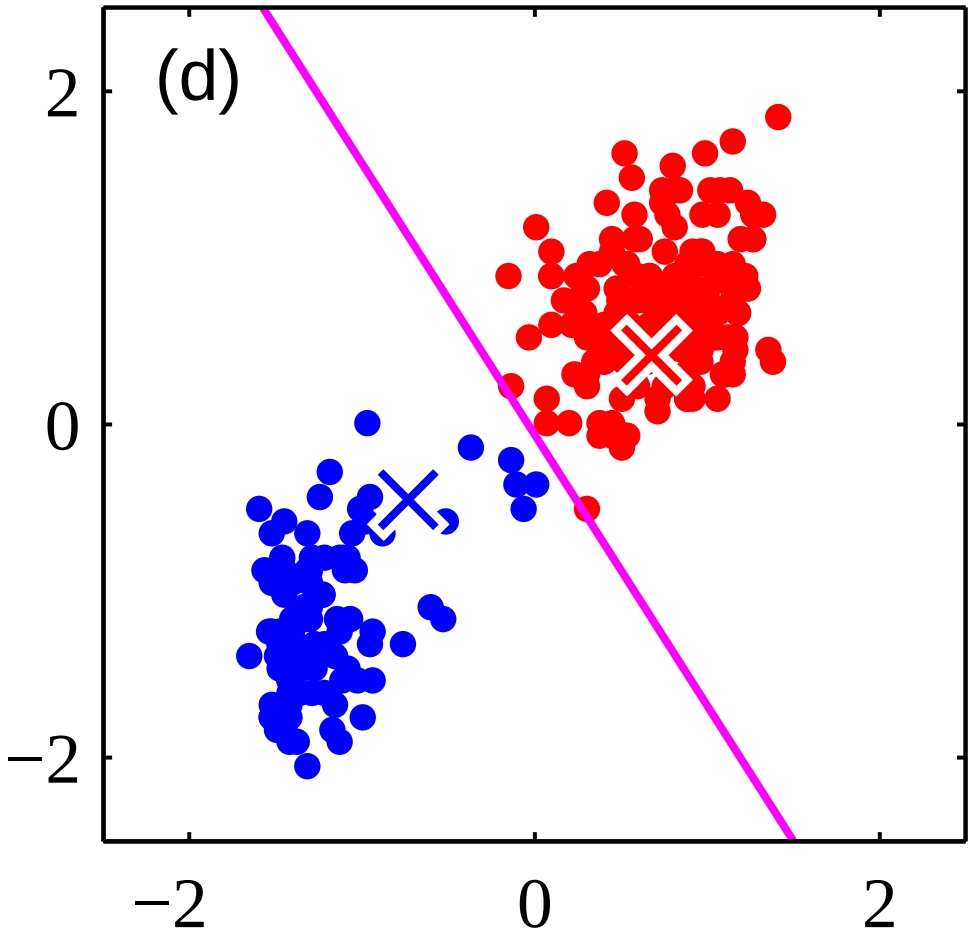
K-means algorithm: Example run



Iteration 1

Update the centroids

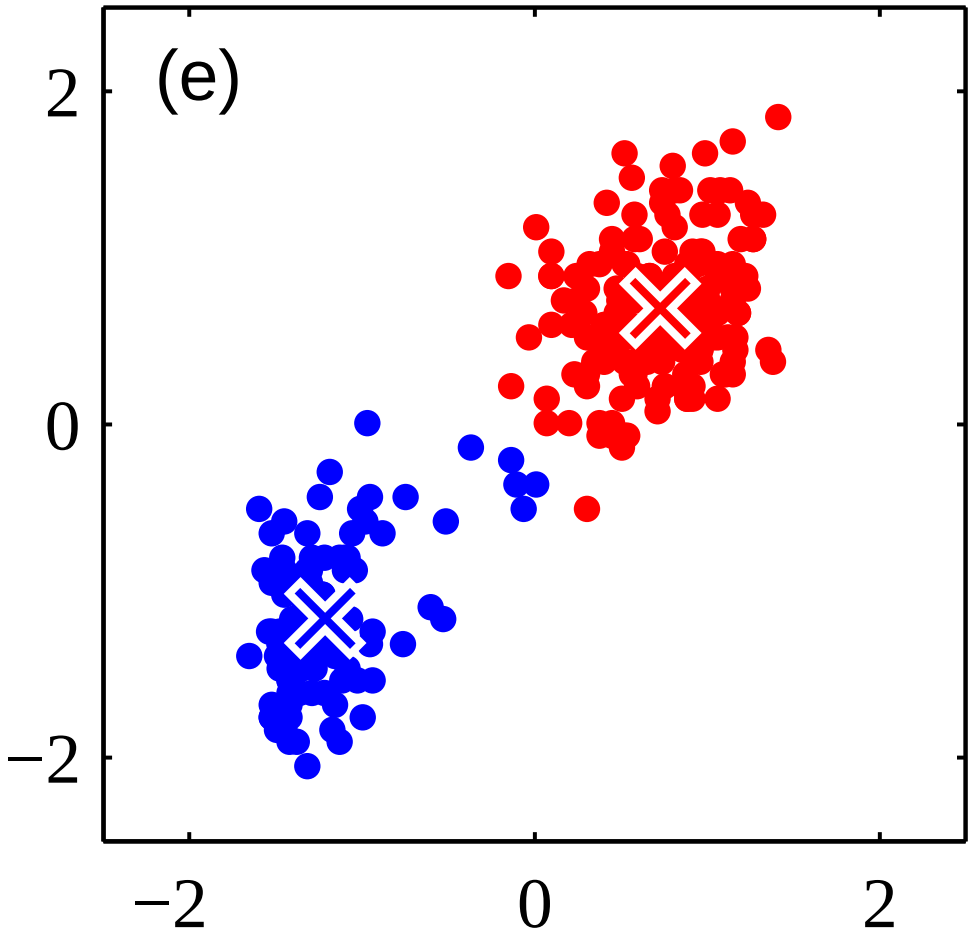
K-MEANS ALGORITHM: EXAMPLE RUN



Iteration 2

Assign data to clusters

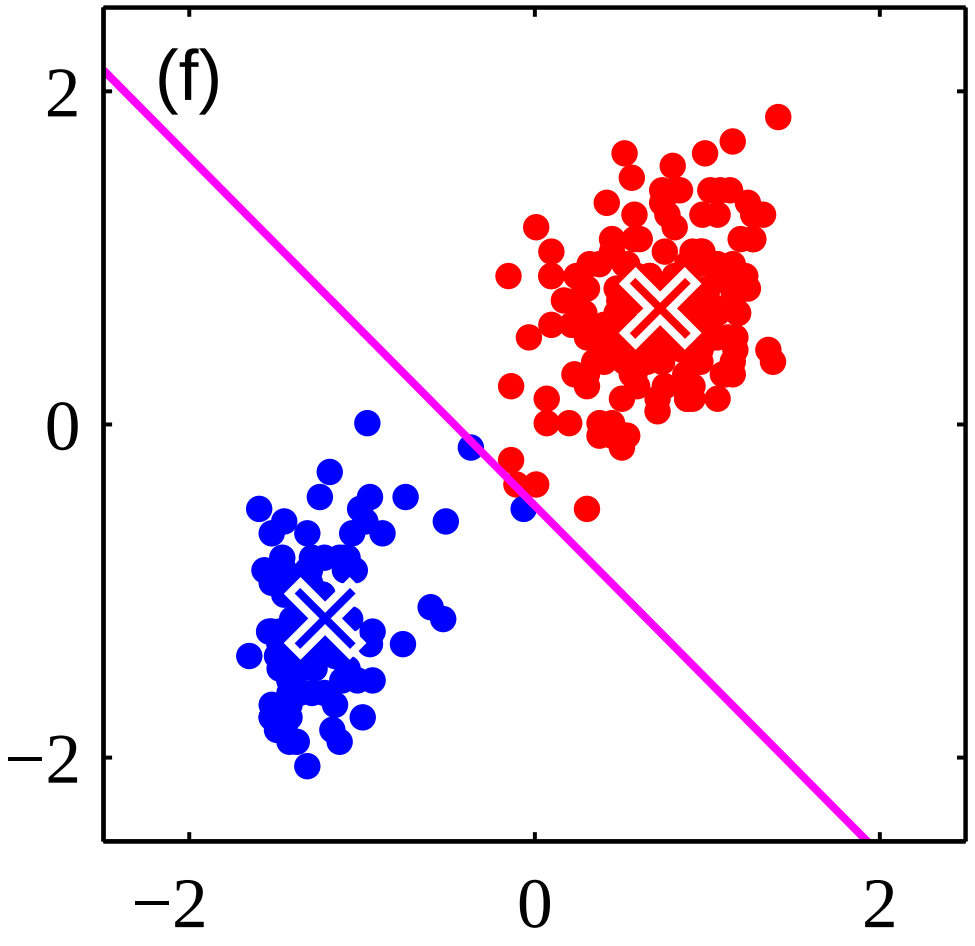
K-MEANS ALGORITHM: EXAMPLE RUN



Iteration 2

Update the centroids

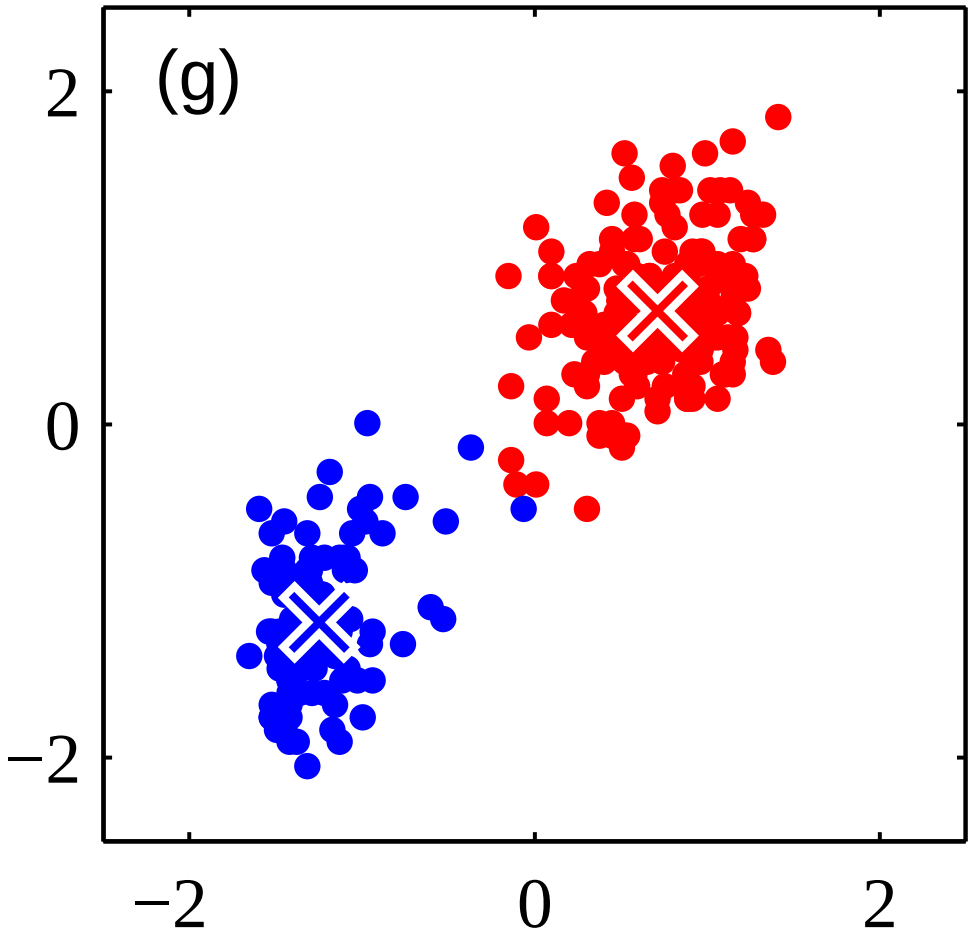
K-means algorithm: Example run



Iteration 3

Assign data to clusters

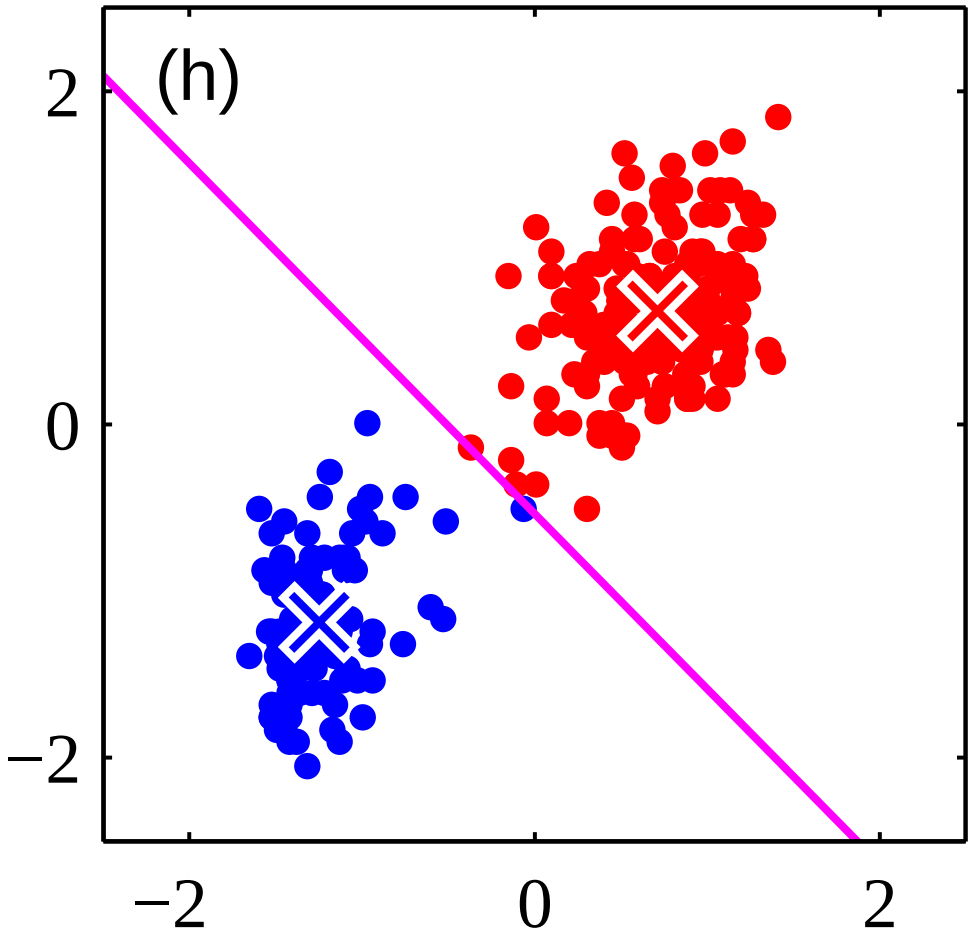
K-means algorithm: Example run



Iteration 3

Update the centroids

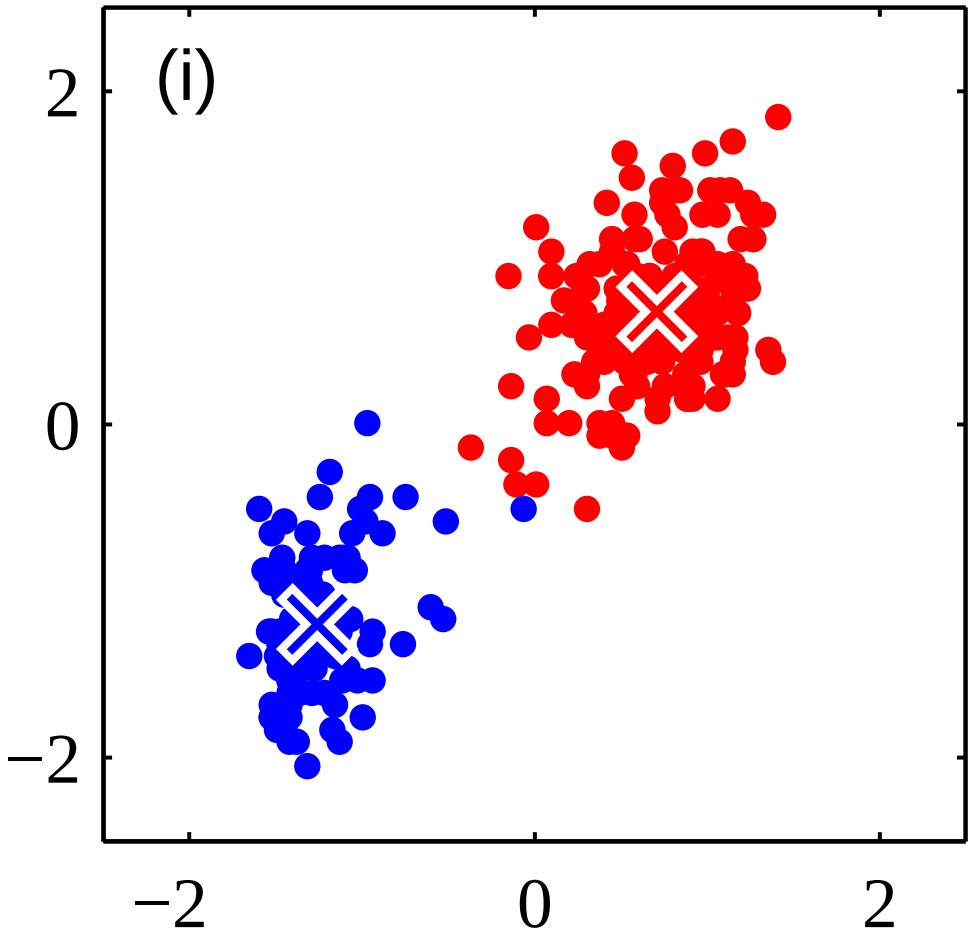
K-means algorithm: Example run



Iteration 4

Assign data to clusters

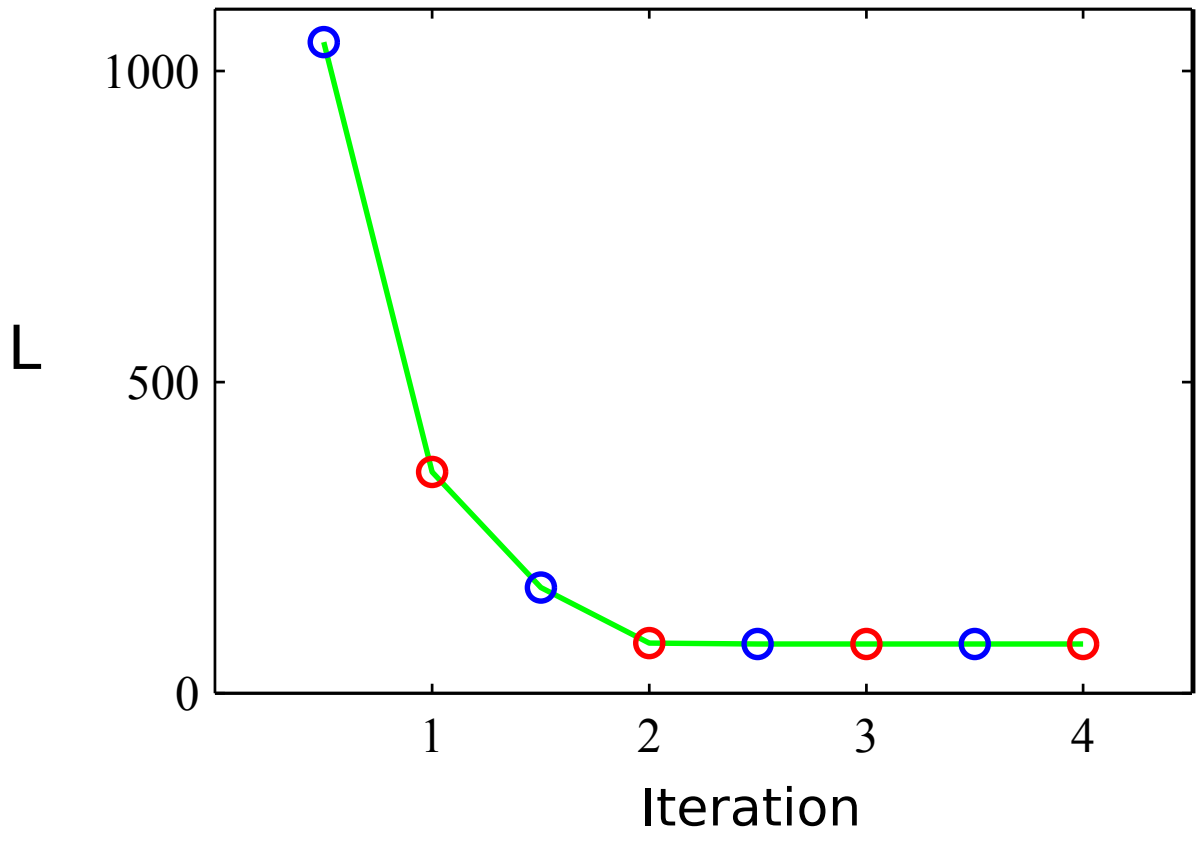
K-MEANS ALGORITHM: EXAMPLE RUN



Iteration 4

Update the centroids

CONVERGENCE OF K-MEANS



Objective function after

- the "assignment" step (blue: corresponding to $\boldsymbol{c}$), and
- the "update" step (red: corresponding to $\boldsymbol{\mu}$).

Convergence of K-means

The outline of why this convergences is straightforward:

1. Every update to c_i or μ_k decreases $\mathcal{L}$ compared to the previous value.
2. Therefore, $\mathcal{L}$ is *monotonically decreasing*.
3. $\mathcal{L} \geq 0$, so Step 1 converges to some point (but probably not to 0).

When $\boldsymbol{c}$ stops changing, the algorithm has converged to a *local* optimal solution. This is a result of $\mathcal{L}$ not being convex.

Non-convexity means that different initializations will give different results:

- Often the results will be similar in quality, but no guarantees.
- In practice, the algorithm can be run multiple times with different initializations. Then use the result with the lowest $\mathcal{L}$.

SELECTING K

We don't know how many clusters there are, but selecting K is tricky.
The K-means objective function decreases as K increases,

$$\mathcal{L} = \sum_{i=1}^{n} \sum_{k=1}^{K} \mathbb{1}\{c_i = k\} \|x_i - \mu_k\|^2.$$

For example, if $K = n$ then let $\mu_k = x_k$ and as a result $\mathcal{L} = 0$.

Methods for choosing K include:

- Using advanced knowledge. e.g., if you want to split a set of tasks among K people, then you already know K.
- Looking at the *relative* decrease in $\mathcal{L}$. If K^* is best, then increasing K when $K \leq K^*$ should decrease $\mathcal{L}$ much more than when $K > K^*$.
- Often the K-means result is part of a larger application. The main application may start to perform worse even though $\mathcal{L}$ is decreasing.
- More advanced modeling techniques exist that address this issue.

Two applications of K-means

Lossy data compression

Approach: Vectorize 2×2 patches from an image (so data is $x \in \mathbb{R}^4$) and cluster them with K-means. Replace each patch with its assigned centroid.

(left) Original 1024×1024 image requiring 8 bits/pixel (1MB total)
(middle) Approximation using 200 clusters (requires 239KB storage)
(right) Approximation using 4 clusters (requires 62KB storage)

Data preprocessing (side comment)

K-means is also very useful for *discretizing* data as a preprocessing step. This allows us to recast a continuous-valued problem as a discrete one.

EXTENSIONS: K-MEDOIDS

Algorithm: K-medoids clustering

Input: Data $x_1, \ldots, x_n$ and distance measure $D(x, \mu)$. Randomly initialize $\boldsymbol{\mu}$.

- Iterate until $\boldsymbol{c}$ is no longer changing

  1. For each c_i : Set
  $$c_i = \arg\min_k \; D(x_i, \mu_k)$$
  2. For each μ_k : Set
  $$\mu_k = \arg\min_\mu \sum_{i:c_i=k} D(x_i, \mu)$$

  Comment: Step #2 may require an algorithm.

K-medoids is a straightforward extension of K-means where the distance measure isn't the squared error. That is,

- K-means uses $D(x, \mu) = \|x - \mu\|^2$.
- Could set $D(x, \mu) = \|x - \mu\|_1$, which would be more robust to outliers.
- If $x \notin \mathbb{R}^d$, we could define $D(x, \mu)$ to be more complex.