# Introduction to Machine Learning

Thomas G. Dietterich
tgd@eecs.oregonstate.edu

# Outline

- What is Machine Learning?
- Introduction to Supervised Learning: Linear Methods
- Overfitting, Regularization, and the Bias-Variance Tradeoff
- Review and Summary

# Machine Learning: The Original Motivation

**Traditional Software Process**

- Interview the experts
- Create an algorithm that automates their process

**Machine Learning Process**

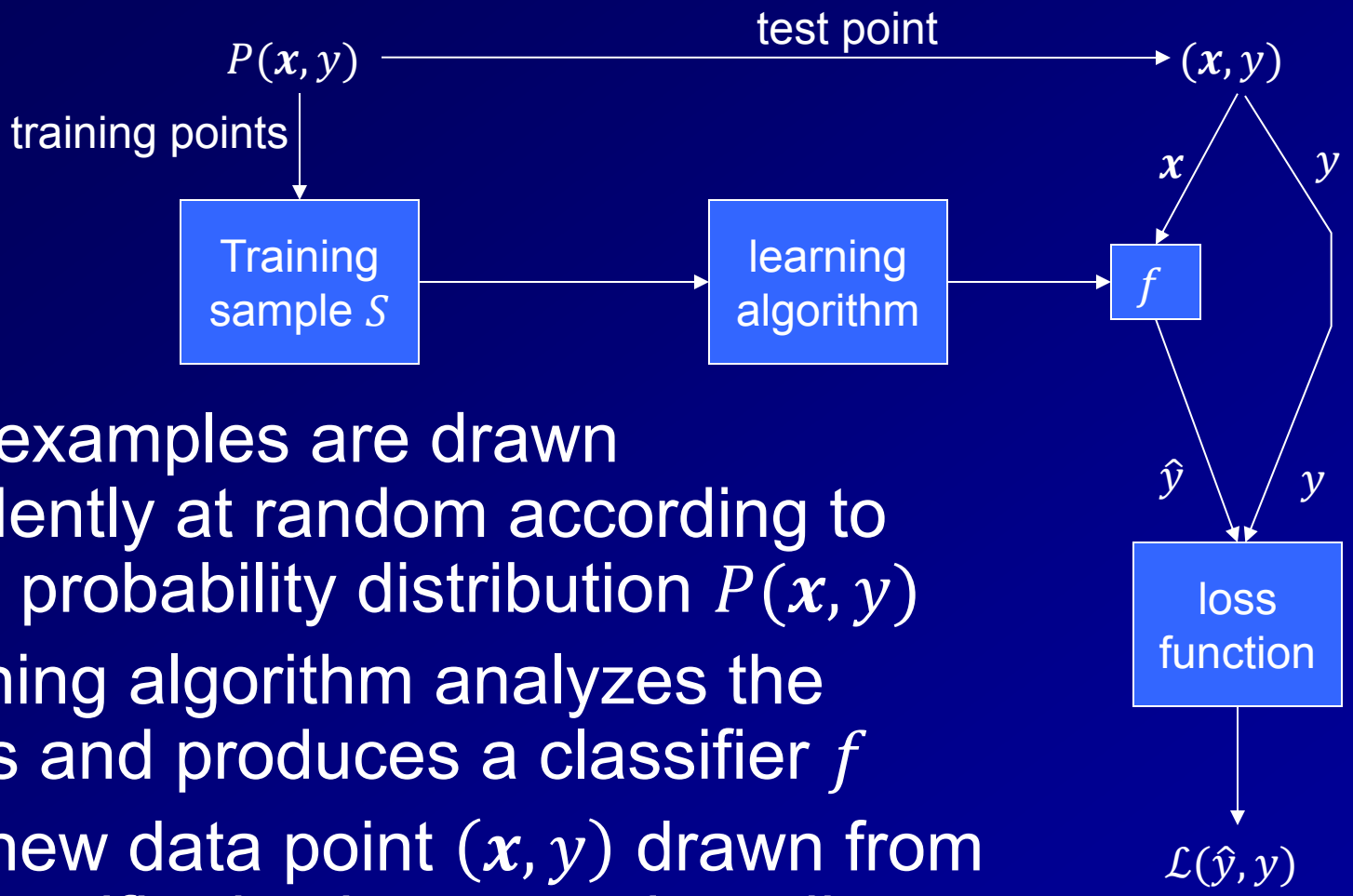
- Collect input-output examples from the experts
- Learn a function to map from the input to the output

# Supervised Learning

- Given: Training examples $(\boldsymbol{x}_i, f(\boldsymbol{x}_i))$ for some unknown function *f*.
- Find: A good approximation to *f*.

- Example Applications
  - Handwriting recognition
    - $x$: data from pen motion
    - $f(x)$: letter of the alphabet
  - Disease Diagnosis
    - $x$: properties of patient (symptoms, lab tests)
    - $f(x)$: disease (or maybe, recommended therapy)
  - Face Recognition
    - $x$: bitmap picture of person's face
    - $f(x)$: name of person
  - Spam Detection
    - $x$: email message
    - $f(x)$: spam or not spam

# Formal Setting




- Training examples are drawn independently at random according to unknown probability distribution $P(\boldsymbol{x}, y)$
- The learning algorithm analyzes the examples and produces a classifier $f$
- Given a new data point $(\boldsymbol{x}, y)$ drawn from $P$, the classifier is given $\boldsymbol{x}$ and predicts $\hat{y} = f(\boldsymbol{x})$
- The loss $\mathcal{L}(\hat{y}, y)$ is then measured
- Goal of the learning algorithm: Find the $f$ that minimizes the *expected loss*

# Formal Version of Spam Detection

- $P(\boldsymbol{x}, y)$: distribution of email messages **x** and their true labels $y$ ("spam" or "not spam")
- training sample: a set of email messages that have been labeled by the user
- learning algorithm: what we study in MLSS!
- $f$: the classifier output by the learning algorithm
- test point: A new email message $\boldsymbol{x}$ (with its true, but hidden, label $y$)
- loss function $\mathcal{L}(\hat{y}, y)$ :

| predicted label $\hat{y}$ | true label $y$ | |
|---|---|---|
| | spam | not spam |
| spam | 0 | 10 |
| not spam | 1 | 0 |

# Three Main Approaches to Machine Learning

- Learn a classifier: a function $f$.
- Learn a conditional distribution: a conditional distribution $P(y \mid \boldsymbol{x})$
- Learn the joint probability distribution: $P(\boldsymbol{x}, y)$
- We will study one example of each method:
  - Learn a classifier: The Perceptron algorithm
  - Learn a conditional distribution: Logistic regression
  - Learn the joint distribution: Linear discriminant analysis

# Linear Threshold Units

$$h(\boldsymbol{x}) = \begin{cases} +1 & \text{if } w_1 x_1 + \cdots + w_n x_n \geq w_0 \\ -1 & \text{otherwise} \end{cases}$$

- We assume that each feature $x_j$ and each weight $w_j$ is a real
- We will study three different algorithms for learning linear threshold units:
  - Perceptron: function
  - Logistic Regression: conditional distribution
  - Linear Discriminant Analysis: joint distribution

# A canonical representation

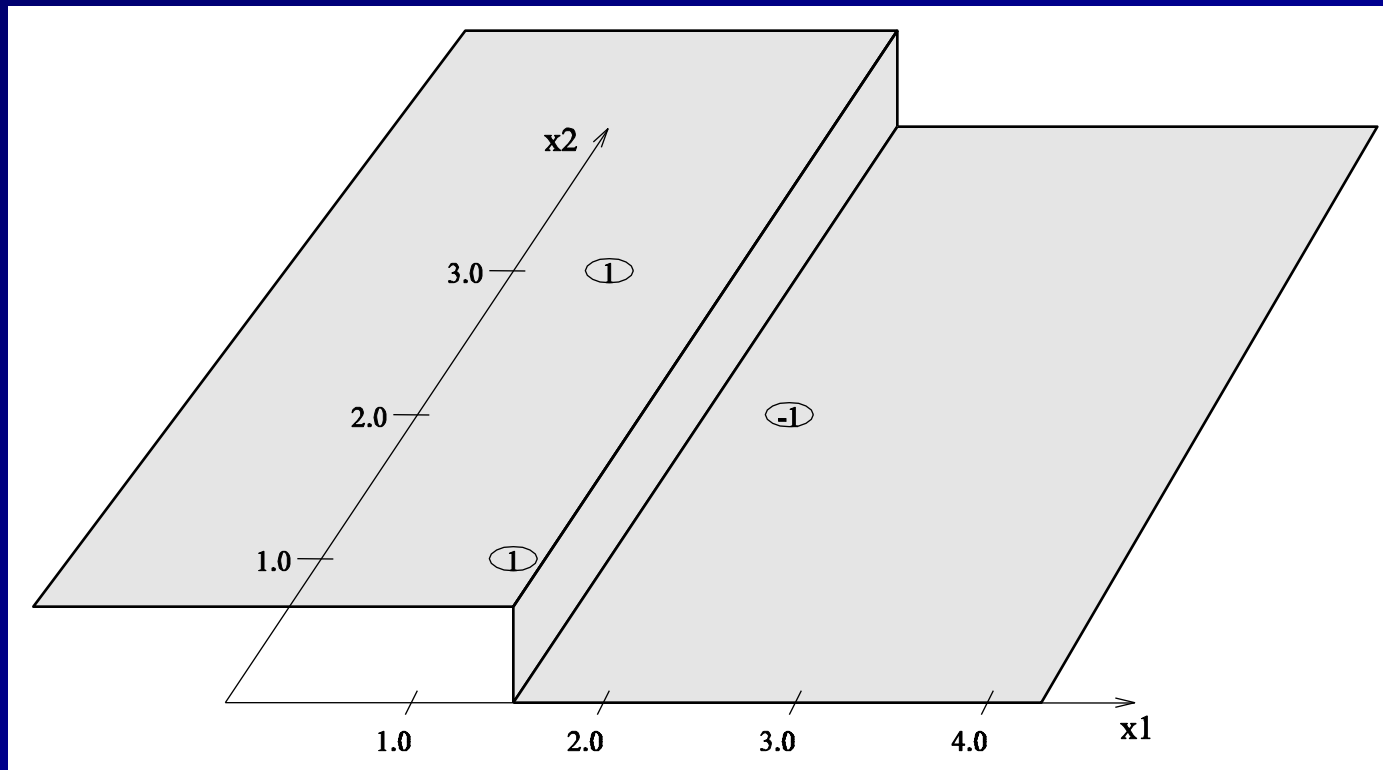
- Given a training example of the form
  $(\langle x_1, x_2, x_3, x_4 \rangle, y)$
- transform it to
  $(\langle -1, x_1, x_2, x_3, x_4 \rangle, y)$
- The parameter vector will then be
  $(w_0, w_1, w_2, w_3, w_4)$
- We will call the *unthresholded* hypothesis $u(\boldsymbol{x}, \boldsymbol{w})$
  $u(\boldsymbol{x}, \boldsymbol{w}) = \boldsymbol{w} \cdot \boldsymbol{x} = \boldsymbol{w}^\top \boldsymbol{x}$
- Each hypothesis can be written
  $h(\boldsymbol{x}) = sgn\big(u(\boldsymbol{x}, \boldsymbol{w})\big)$
- Our goal is to find $\boldsymbol{w}$

# Geometrical View

- Consider three training examples:

$$(\langle 1.0,1.0\rangle, +1)$$
$$(\langle 0.5,3.0\rangle, +1)$$
$$(\langle 2.0,2.0\rangle, -1)$$

- We want a classifier that looks like the following:



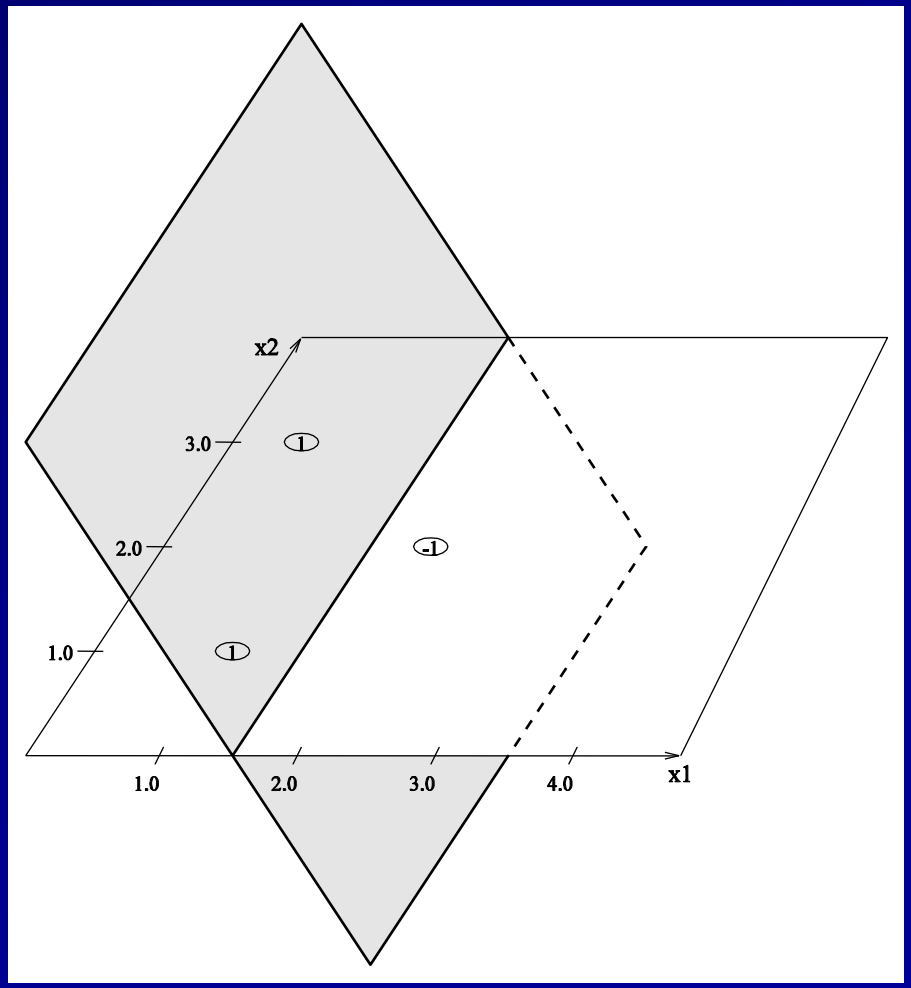
# The Unthresholded Discriminant Function is a Hyperplane

- The equation

$$u(\boldsymbol{x}) = \boldsymbol{w}^{\top}\boldsymbol{x}$$

is a plane

$$\hat{y} = \begin{cases} +1 & \text{if } u(\boldsymbol{x}) \geq 0 \\ -1 & \text{otherwise} \end{cases}$$



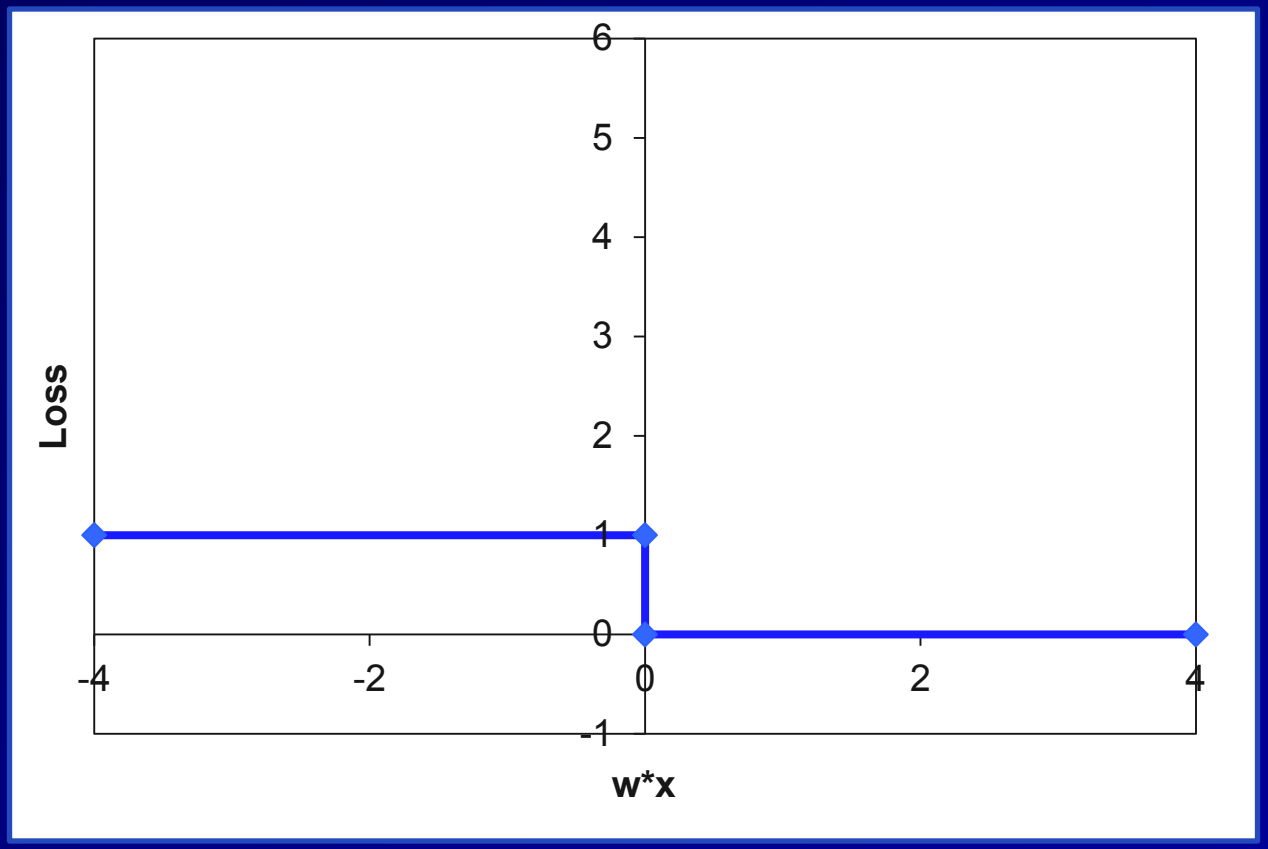
# Machine Learning == Optimization

- Given:
  - A set of *N* training examples
    $\{(\boldsymbol{x}_1, y_1), (\boldsymbol{x}_2, y_2), \ldots, (\boldsymbol{x}_N, yN)\}$
  - A loss function $\mathcal{L}$

- Find:
  - The weight vector $\boldsymbol{w}$ that minimizes the expected loss on the training data

$$J(\boldsymbol{w}) = \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}(sgn(\boldsymbol{w}^{\mathsf{T}} \boldsymbol{x}_i), y_i)$$

# Problem: Step-wise Constant Loss Function



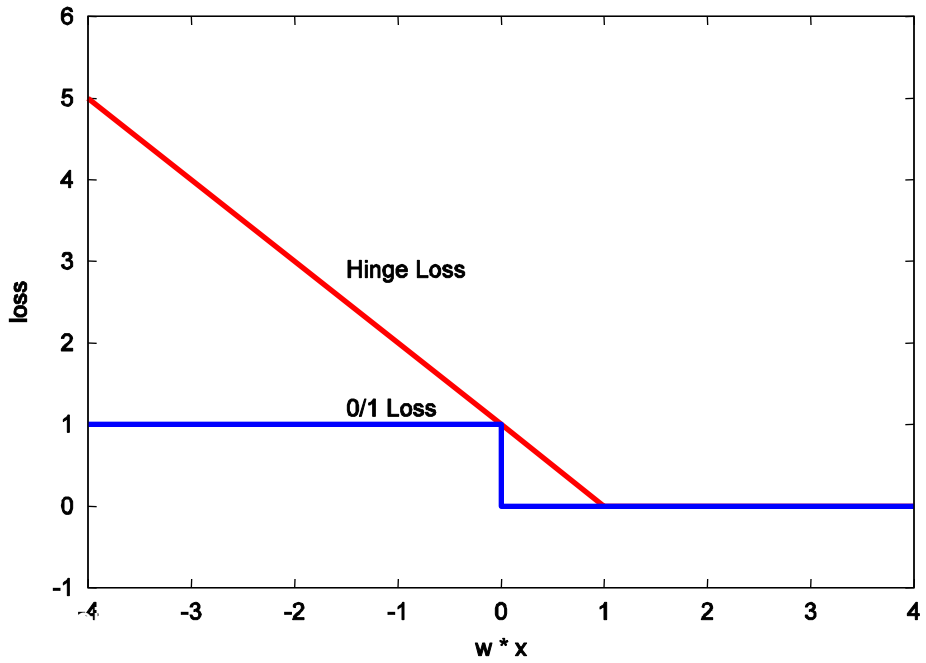

Derivative is either 0 or ∞

# Approximating the expected loss by a smooth function

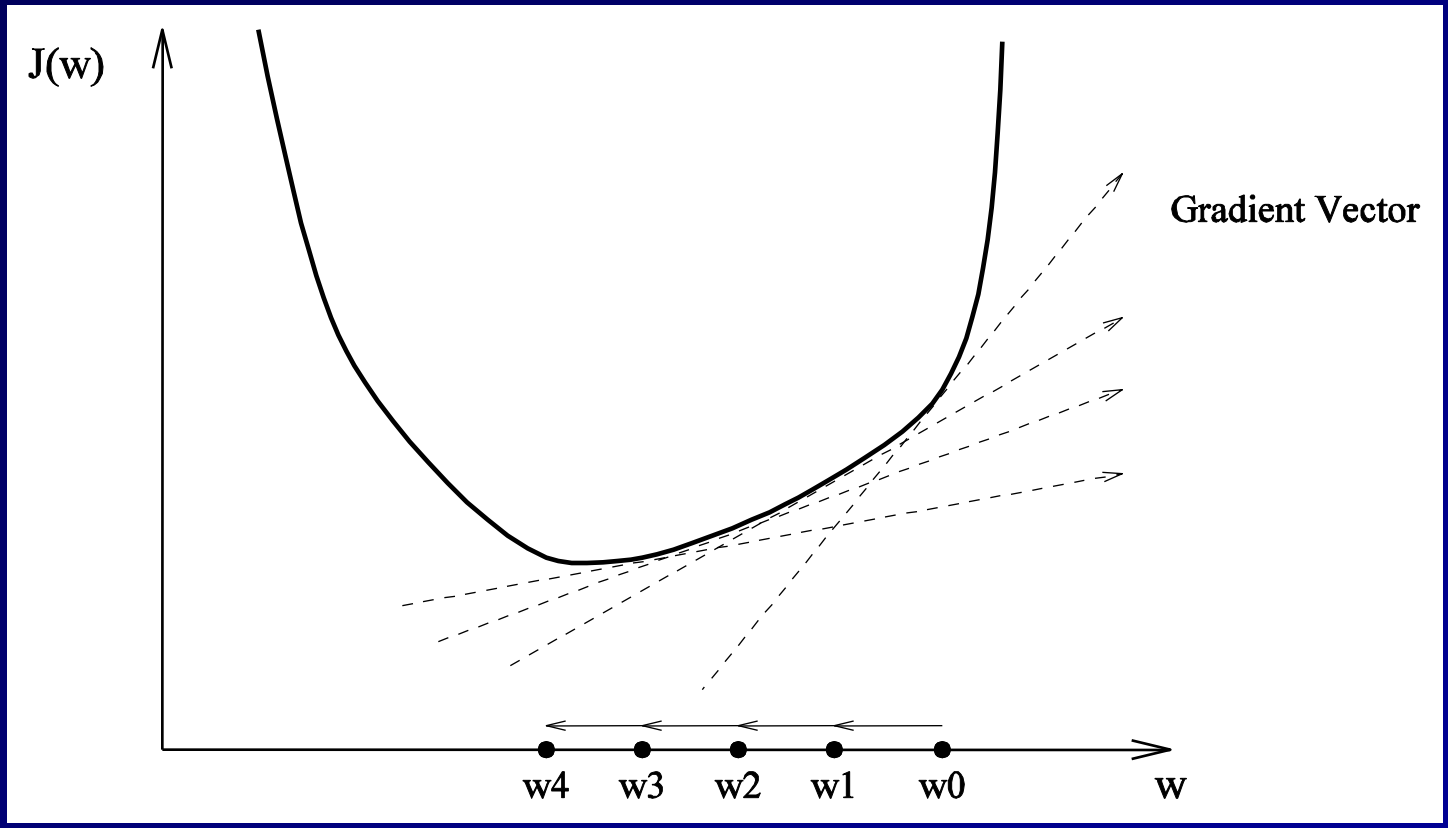
- Simplify the optimization problem by replacing the original objective function by a surrogate loss function. For example, consider the *hinge loss*:

$$\tilde{J}(\boldsymbol{w}) = \frac{1}{N}\sum_{i=1}^{N} \max\left(0, 1 - y_i \boldsymbol{w}^\top \boldsymbol{x}\right)$$

When $y = 1$:



# Minimizing $\tilde{J}$ by Gradient Descent Search




- Start with weight vector $\boldsymbol{w}^0$
- Compute gradient $\nabla\tilde{J}(\boldsymbol{w}^0) = \left(\frac{\partial\tilde{J}(\boldsymbol{w}^0)}{\partial w_0}, \frac{\partial\tilde{J}(\boldsymbol{w}^0)}{\partial w_1}, \ldots, \frac{\partial\tilde{J}(\boldsymbol{w}^0)}{\partial w_n}\right)$
- Compute $\boldsymbol{w}^1 = \boldsymbol{w}^0 - \eta\nabla\tilde{J}(\boldsymbol{w}^0)$
  where $\eta$ is a "step size" parameter
- Repeat until convergence

# Gradient of the Hinge Loss

Let $\tilde{J}_i(\boldsymbol{w}) = \max(0, -y_i \boldsymbol{w}^\top \boldsymbol{x})$

$$\frac{\partial \tilde{J}(\boldsymbol{w})}{\partial w_k} = \frac{\partial}{\partial w_k}\left(\frac{1}{N}\sum_{i=1}^{N} \tilde{J}_i(\boldsymbol{w})\right) = \frac{1}{N}\sum_{i=1}^{N} \frac{\partial}{\partial w_k} \tilde{J}_i(\boldsymbol{w})$$

$$\sum_{i=1}^{N} \frac{\partial}{\partial w_k} \tilde{J}_i(\boldsymbol{w}) = \frac{\partial}{\partial w_k} \max\left(0, -y_i \sum_j w_j x_{ij}\right)$$

$$= \begin{cases} 0 & \text{if } y_i \boldsymbol{w}^\top \boldsymbol{x} \geq 0 \\ -y_i x_{ik} & \text{otherwise} \end{cases}$$

# Batch Perceptron Algorithm

Input: training examples $\{(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_N, y_N)\}$
$\boldsymbol{w} = (0, \ldots, 0)$ // initial weight vector
Repeat until convergence
    $\boldsymbol{g} = (0, \ldots, 0)$ // initial gradient vector
    For $i = 1$ to $N$ do
        if $(y_i \boldsymbol{w}^\top \boldsymbol{x}_i < \boldsymbol{0})$ // $\boldsymbol{x}_i$ is misclassified
            For $j = 1$ to $n$ do
                $g_j = g_j - y_i x_{ij}$
    $\boldsymbol{g} \coloneqq \boldsymbol{g}/N$ // average gradient
    $\boldsymbol{w} \coloneqq \boldsymbol{w} - \eta \boldsymbol{g}$ // take a gradient step

# Online Perceptron Algorithm

$\boldsymbol{w} = (0, \ldots, 0)$ be the initial weight vector
$\boldsymbol{g} = (0, \ldots, 0)$ be the initial gradient vector
Repeat forever
    Accept training example $(\boldsymbol{x}_i, y_i)$
    if $(y_i \boldsymbol{w}^\top \boldsymbol{x}_i < \boldsymbol{0})$ // $\boldsymbol{x}_i$ is misclassified
        For $j = 1$ to $n$ do $g_j = -y_i x_{ij}$
    $\boldsymbol{w} \coloneqq \boldsymbol{w} - \eta \boldsymbol{g}$ // take a gradient step

This is called stochastic gradient descent because the overall gradient is approximated by the gradient from each individual example

# Learning Rates and Convergence

- The learning rate $\eta$ must decrease to zero in order to guarantee convergence. The online case is known as the Robbins-Munro algorithm. It is guaranteed to converge under the following assumptions:

$$\lim_{t \to \infty} \eta_t$$

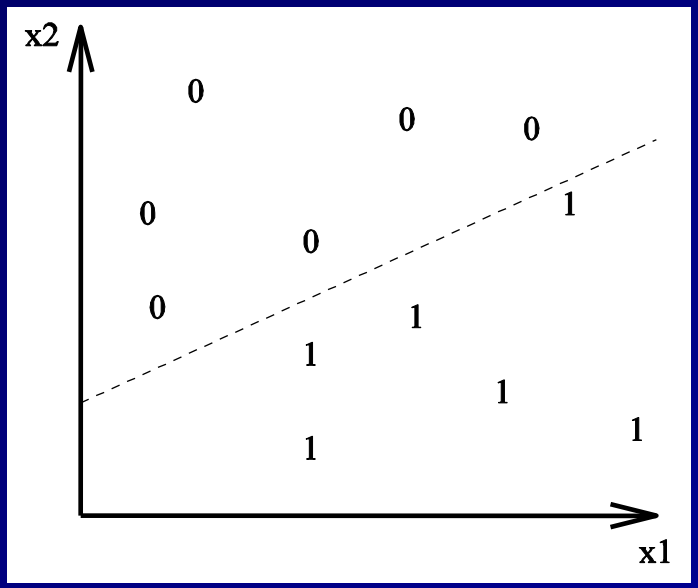
$$\sum_{t=0}^{\infty} \eta_t = \infty$$

$$\sum_{t=0}^{\infty} \eta_t^2 < \infty$$

- The learning rate is also called the step size. Some algorithms (e.g., Newton's method, conjugate gradient) choose the stepsize automatically and converge faster
- There is only one "basin" for linear threshold units, so a local minimum is the global minimum. Choosing a good starting point can make the algorithm converge faster

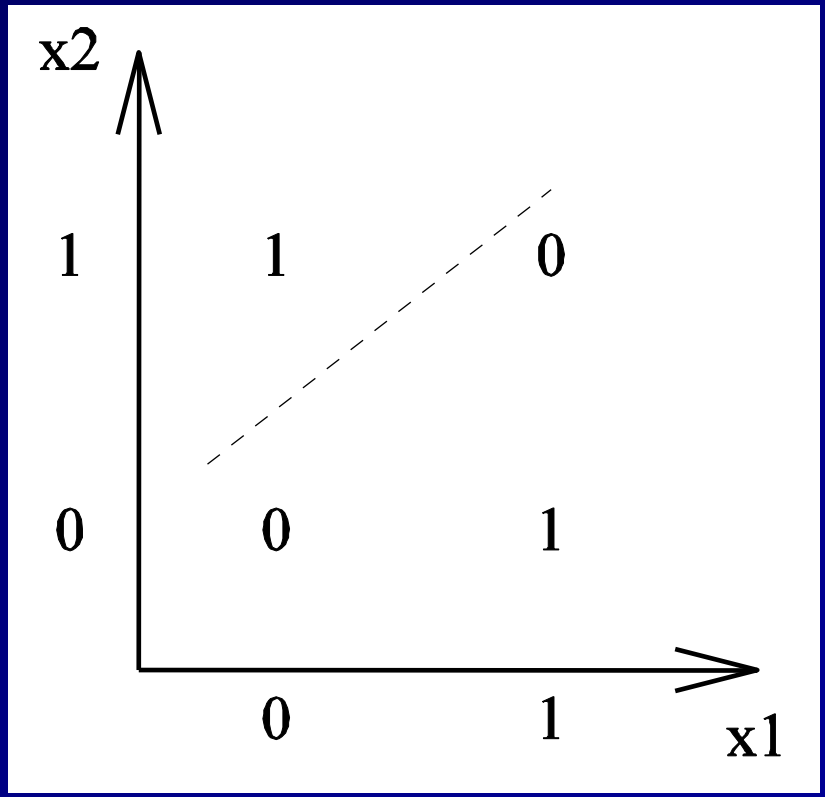
# Decision Boundaries

- A classifier can be viewed as partitioning the input space or feature space X into decision regions




- A linear threshold unit always produces a linear decision boundary. A set of points that can be separated by a linear decision boundary is said to be linearly separable.

# Exclusive-OR is Not Linearly Separable



# Review

- We adopted the discriminant function approach (no probabilistic model)
- We adopted the hinge loss as a surrogate for the 0/1 loss
- We formulated the optimization problem of minimizing the average hinge loss on the training data
- We solved this problem using gradient descent ➔ Perceptron algorithm

# Logistic Regression

- Learn the conditional probability $P(y \mid \boldsymbol{x})$
- Let $p_y(\boldsymbol{x};\ \boldsymbol{w})$ be our estimate of $P(y \mid \boldsymbol{x})$, where $\boldsymbol{w}$ is a vector of adjustable parameters. Assume only two classes $y = 0$ and $y = 1$, and

$$p_1(\boldsymbol{x}|\boldsymbol{w}) = \frac{\exp \boldsymbol{w}^\top \boldsymbol{x}}{\mathbf{1} + \exp \boldsymbol{w}^\top \boldsymbol{x}}$$

$$p_0(\boldsymbol{x}|\boldsymbol{w}) = 1 - p_1(\boldsymbol{x}|\boldsymbol{w})$$

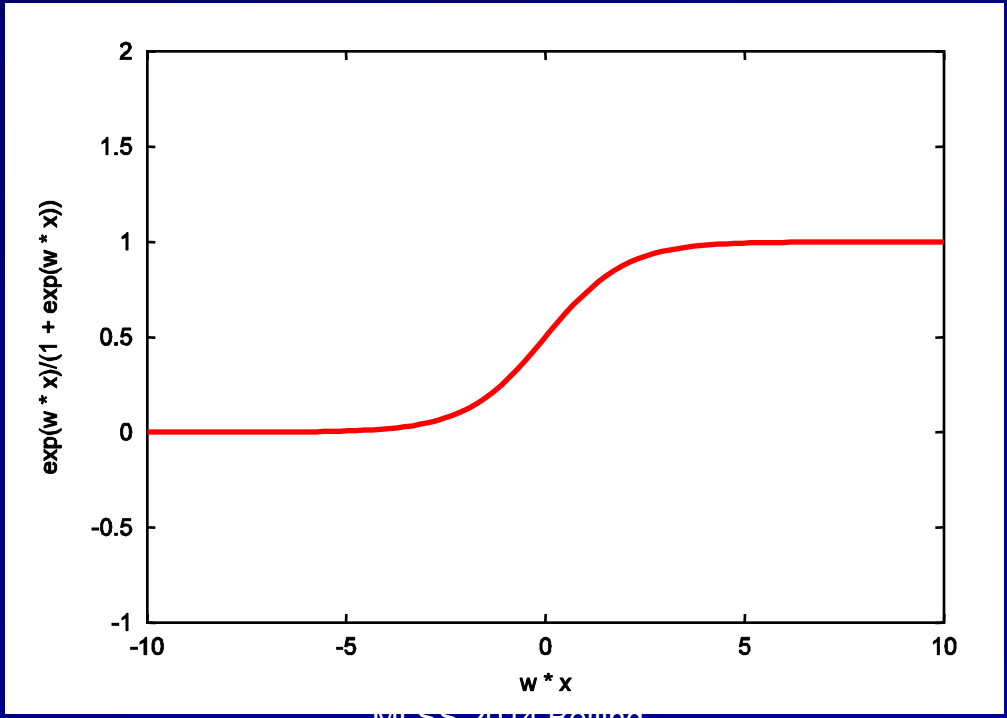
- It is easy to show that this is equivalent to

$$\log \frac{p_1(\boldsymbol{x}; \boldsymbol{w})}{p_0(\boldsymbol{x}; \boldsymbol{w})} = \boldsymbol{w}^\top \boldsymbol{x}$$

- In other words, the log odds of class 1 is a linear function of $\boldsymbol{x}$.

# Why the exp function?

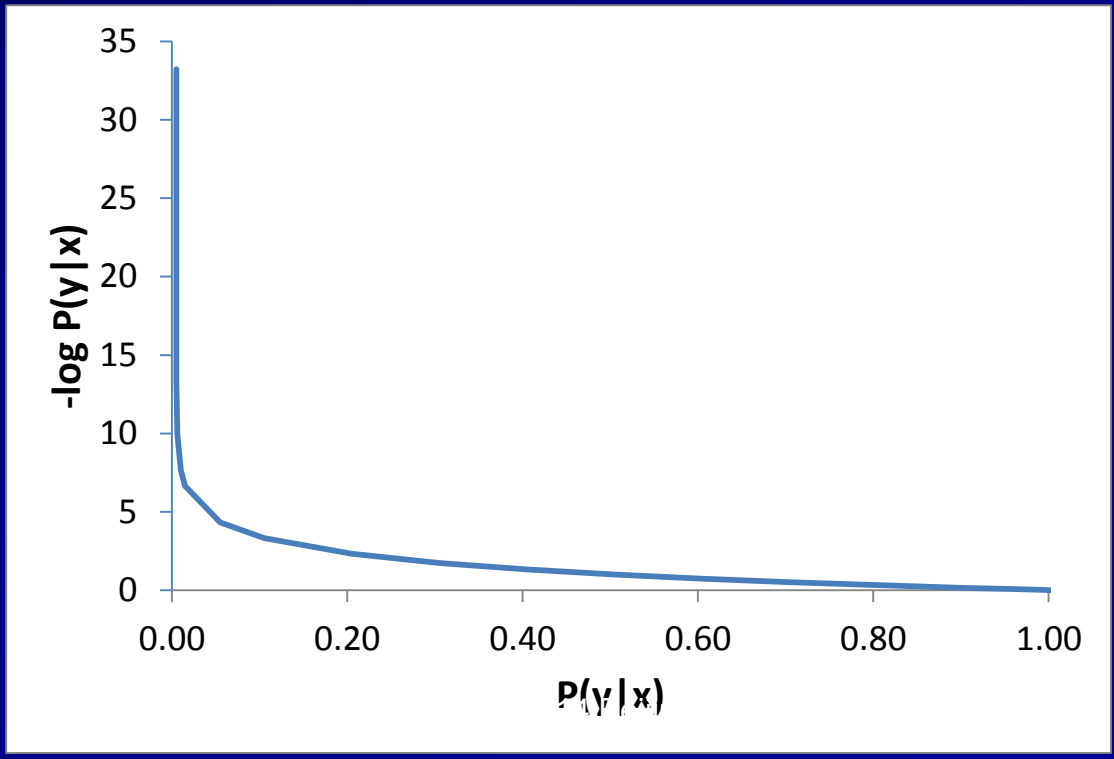
- One reason: A linear function has a range from $-\infty$ to $+\infty$ and we need to force it to be positive and sum to 1 in order to be a probability:



# Deriving a Learning Algorithm: Choosing the Loss Function

- For probabilistic models, we use the log loss:

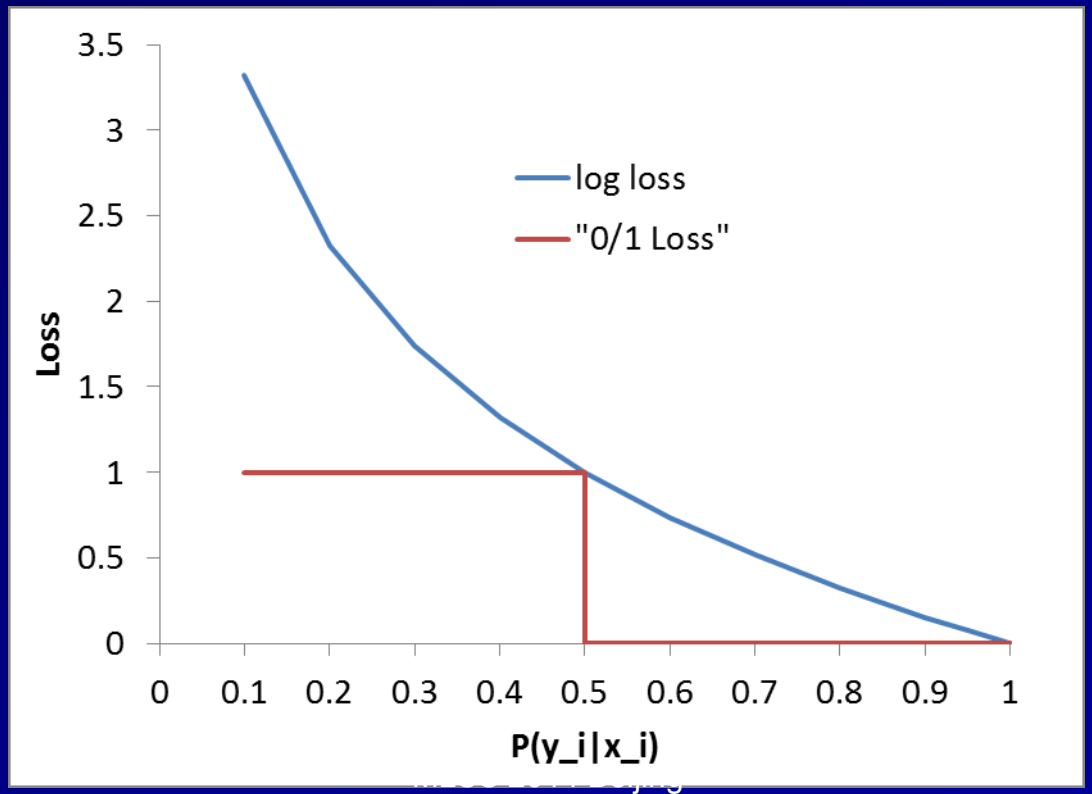
$$\mathcal{L}\left(\hat{P}(y|x), y\right) = \begin{cases} -\log \hat{P}(y = 1|\boldsymbol{x}_i) & \text{if } y_i = 1 \\ -\log \hat{P}(y = 0|\boldsymbol{x}_i) & \text{if } y_i = 0 \end{cases}$$



# Comparison with 0/1 Loss

- For probabilistic models, we use the log loss:

$$\mathcal{L}\left(\hat{P}(y|x), y\right) = \begin{cases} -\log \hat{P}(y = 1|\boldsymbol{x}_i) & \text{if } y_i = 1 \\ -\log \hat{P}(y = 0|\boldsymbol{x}_i) & \text{if } y_i = 0 \end{cases}$$



# Maximum Likelihood Fitting

- To minimize the log loss, we should maximize $\log \hat{P}(y_i|x_i)$
- The *likelihood* of the data is:

$$\prod_i \hat{P}(y_i|x_i)$$

- It is easier to work with the *log likelihood*:

$$\sum_i \log \hat{P}(y_i|x_i)$$

# Maximizing the log likelihood via gradient ascent

- Rewrite the log likelihood in terms of $p_1(\boldsymbol{x}_i; \boldsymbol{w})$:

$$\ell(\boldsymbol{w}) = \sum_i y_i \log p_1(\boldsymbol{x}_i; \boldsymbol{w}) + (1 - y_i) \log\big(1 - p_1(\boldsymbol{x}_i; \boldsymbol{w})\big)$$

Differentiate with respect to $w_j$:

$$\frac{\partial \ell(y_i; \boldsymbol{x}_i, \boldsymbol{w})}{\partial w_j} = \frac{y_i}{p_1(\boldsymbol{x}_i, \boldsymbol{w})} \frac{\partial p_1(\boldsymbol{x}_i, \boldsymbol{w})}{\partial w_j} + \frac{1 - y_i}{1 - p_1(\boldsymbol{x}_i, \boldsymbol{w})} \left( - \frac{\partial p_1(\boldsymbol{x}_i, \boldsymbol{w})}{\partial w_j} \right)$$

Gather terms

$$\frac{\partial \ell(y_i; \boldsymbol{x}_i, \boldsymbol{w})}{\partial w_j} = \left[ \frac{y_i}{p_1(\boldsymbol{x}_i, \boldsymbol{w})} - \frac{1 - y_i}{1 - p_1(\boldsymbol{x}_i, \boldsymbol{w})} \right] \left( \frac{\partial p_1(\boldsymbol{x}_i, \boldsymbol{w})}{\partial w_j} \right)$$

# Maximizing the log likelihood via gradient ascent (2)

- Collect over common denominator:

$$\frac{\partial \ell(y_i; \boldsymbol{x}_i, \boldsymbol{w})}{\partial w_j} = \left[\frac{y_i\big(1 - p_1(\boldsymbol{x}_i, \boldsymbol{w})\big) - (1 - y_i)p_1(\boldsymbol{x}_i, \boldsymbol{w})}{p_1(\boldsymbol{x}_i, \boldsymbol{w})(1 - p_1(\boldsymbol{x}_i, \boldsymbol{w}))}\right]\left(\frac{\partial p_1(\boldsymbol{x}_i, \boldsymbol{w})}{\partial w_j}\right)$$

- Simplify:

$$\frac{\partial \ell(y_i; \boldsymbol{x}_i, \boldsymbol{w})}{\partial w_j} = \left[\frac{y_i - p_1(x_i, w)}{p_1(\boldsymbol{x}_i, \boldsymbol{w})(1 - p_1(\boldsymbol{x}_i, \boldsymbol{w}))}\right]\left(\frac{\partial p_1(\boldsymbol{x}_i, \boldsymbol{w})}{\partial w_j}\right)$$

- Now we just need to compute

$$\frac{\partial p_1(\boldsymbol{x}_i, \boldsymbol{w})}{\partial w_j}$$

# Computing $\frac{\partial p_1(\boldsymbol{x}_i, \boldsymbol{w})}{\partial w_j}$

- $p_1$ can be written as

$$p_1(x_i; w) = \frac{1}{1 + \exp(w^\top x)}$$

- From this, we obtain:

$$\begin{aligned}
\frac{\partial p_1(\boldsymbol{x}_i, \boldsymbol{w})}{\partial w_j} &= \frac{1}{(1 + \exp(-w^\top x_i))^2} \frac{\partial (1 + \exp(-w^\top x_i))}{\partial w_j} \\
&= -\frac{1}{(1 + \exp(-w^\top x_i))^2} \exp(-w^\top x_i) \frac{\partial}{\partial w_j} (-w^\top x_i) \\
&= -\frac{1}{(1 + \exp(-w^\top x_i))^2} \exp(-w^\top x_i) \left(-x_{ij}\right) \\
&= p_1(x_i; w)\big(1 - p_1(x_i; w)\big) x_{ij}
\end{aligned}$$

# Putting it together we have

$$\frac{\partial \ell(y_i; \boldsymbol{x}_i, \boldsymbol{w})}{\partial w_j} = \left[\frac{y_i - p_1(\boldsymbol{x}_i; \boldsymbol{w})}{p_1(\boldsymbol{x}_i, \boldsymbol{w})(1 - p_1(\boldsymbol{x}_i, \boldsymbol{w}))}\right]\left(\frac{\partial p_1(\boldsymbol{x}_i; \boldsymbol{w})}{\partial w_j}\right)$$

$$\frac{\partial p_1(\boldsymbol{x}_i, \boldsymbol{w})}{\partial w_j} = p_1(\boldsymbol{x}_i; \boldsymbol{w})\big(1 - p_1(\boldsymbol{x}_i; \boldsymbol{w})\big)x_{ij}$$

$$\frac{\partial \ell(y_i; \boldsymbol{x}_i, \boldsymbol{w})}{\partial w_j} = [y_i - p_1(\boldsymbol{x}_i; \boldsymbol{w})]x_{ij}$$

- The overall gradient is therefore

$$\frac{\partial \ell(\boldsymbol{w})}{\partial w_j} = \sum_i (y_i - p_1(\boldsymbol{x}_i; \boldsymbol{w}))x_{ij}$$

- Note that the first term is the error on the probability scale

# Batch Gradient Ascent for Logistic Regression

Input: training examples $\{(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_N, y_N)\}$
$\boldsymbol{w} = (0, \ldots, 0)$ be the initial weight vector
Repeat until convergence
    $\boldsymbol{g} = (0, \ldots, 0)$ be the initial gradient vector
    For $i = 1$ to $N$ do
        $p_i = 1/(1 + \exp(-\boldsymbol{w}^\top \boldsymbol{x}_i))$
        $\text{error}_i := y_i - p_i$
        For $j = 1$ to $n$ do
            $g_j = g_j + \text{error}_i \cdot x_{ij}$
    $\boldsymbol{g} := \boldsymbol{g}/N$  // average gradient
    $\boldsymbol{w} := \boldsymbol{w} + \eta \boldsymbol{g}$  // take a gradient step

- An online gradient ascent algorithm can be constructed, of course
- Most statistical packages use a second-order (Newton-Raphson) algorithm for faster convergence

# Logistic Regression Implements a Linear Discriminant Function

- In the 2-class 0/1 loss function case, we should predict ŷ = 1 if $\hat{P}(y = 1|\boldsymbol{x}; \boldsymbol{w}) > 0.5$

$$\frac{P(y = 1|\boldsymbol{x}; \boldsymbol{w})}{P(y = 0|\boldsymbol{x}; \boldsymbol{w})} > 1$$

- Take log of both sides

$$\log \frac{P(y = 1|\boldsymbol{x}; \boldsymbol{w})}{P(y = 0|\boldsymbol{x}; \boldsymbol{w})} > 0$$

- or

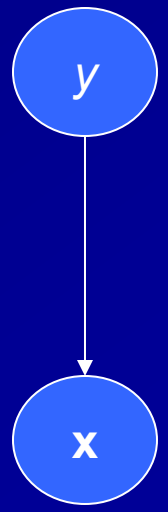
$$\boldsymbol{w}^{\top}\boldsymbol{x} > 0$$

# Review

- We adopted the conditional probability approach: $P(y|\boldsymbol{x})$
- We adopted the log loss as a surrogate for 0/1 loss
- We formulated the optimization problem of maximizing the average log likelihood on the training data
- We solved this problem using gradient ascent

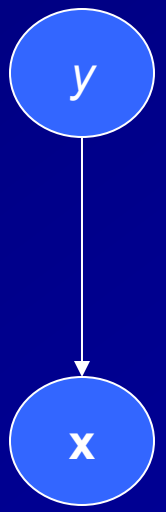
# The Joint Probability Approach: Linear Discriminant Analysis

- Learn $P(\boldsymbol{x}, y)$. This is called the generative approach, because we can think of $P(\boldsymbol{x}, y)$ as a model of how the data is generated.
  - For example, if we factor the joint distribution into the form
    P(**x**,$y$) = P($y$) P(**x** | $y$)
  - Generative "story"
    - draw $y \sim P(y)$ choose a class
    - draw $\boldsymbol{x} \sim P(\boldsymbol{x}|y)$ generate the features for $\boldsymbol{x}$
  - This can be represented as a probabilistic graphical model



# Linear Discriminant Analysis (2)

- P($y$) is a discrete multinomial distribution
  - example: P($y$ = 0) = 0.31, P($y$ = 1) = 0.69 will generate 31% negative examples and 69% positive examples
- For LDA, we assume that $P(\boldsymbol{x} \mid y)$ is a multivariate normal distribution with mean $\mu_k$ and covariance matrix $\Sigma$



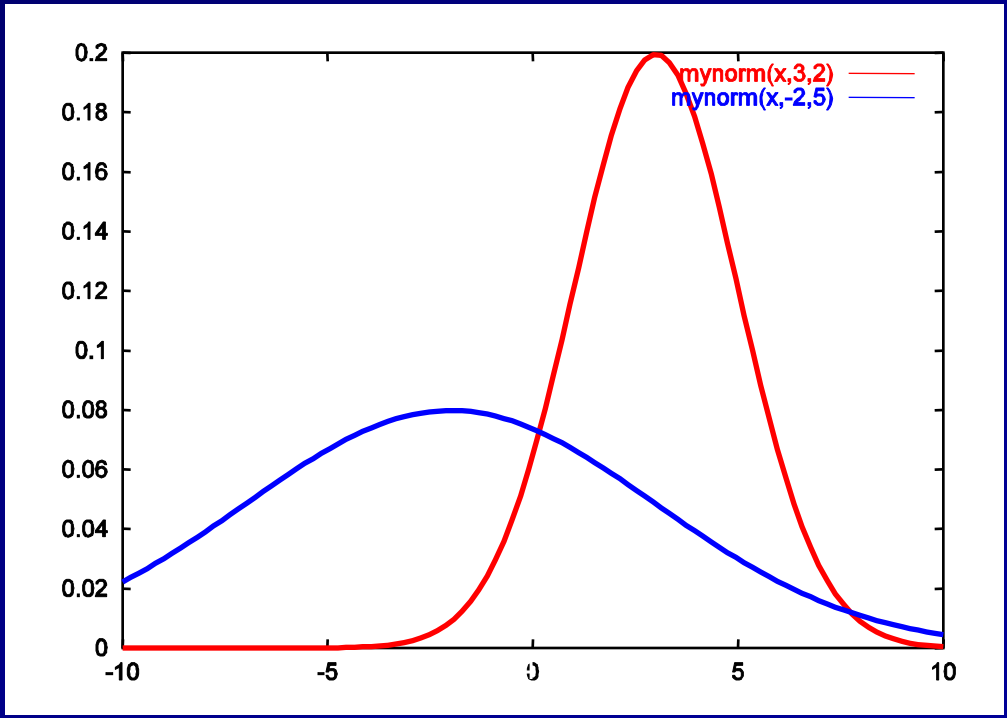

$$P(\boldsymbol{x}|y=k) = \frac{1}{(2\pi)^{n/2}|\Sigma|^{1/2}} \exp\left(-\frac{1}{2}[\boldsymbol{x}-\boldsymbol{\mu}_k]^{\top}\Sigma^{-1}[\boldsymbol{x}-\boldsymbol{\mu}_k]\right)$$

# Multivariate Normal Distributions: A tutorial

- Recall that the univariate normal (Gaussian) distribution has the formula

$$p(x) = \frac{1}{(2\pi)^{\frac{1}{2}}\sigma} \exp\left[-\frac{1}{2}\frac{(x-\mu)^2}{\sigma^2}\right]$$

- where $\mu$ is the mean and $\sigma^2$ is the variance
- Graphically, it looks like this:



# The Multivariate Gaussian

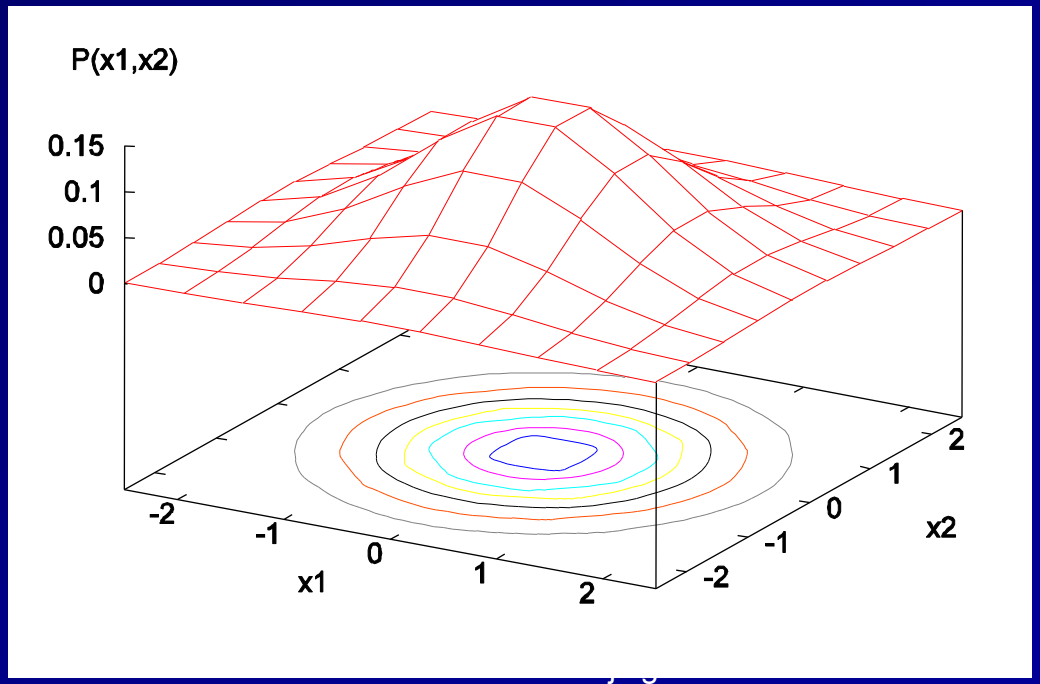
- A 2-dimensional Gaussian is defined by a mean vector $\mu = (\mu_1, \mu_2)$ and a covariance matrix

$$\Sigma = \begin{bmatrix} \sigma_{1,1}^2 & \sigma_{1,2}^2 \\ \sigma_{2,1}^2 & \sigma_{2,2}^2 \end{bmatrix}$$

- where $\sigma_{i,j}^2 = E\left[(x_i - \mu_i)(x_j - \mu_j)\right]$ is the variance (if $i = j$) or co-variance (if $i \neq j$). $\Sigma$ is symmetric and positive-definite

# The Multivariate Gaussian (2)

- If $\Sigma$ is the identity matrix $\Sigma = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ and $\mu = (0, 0)$, we get the 2-D standard normal distribution:

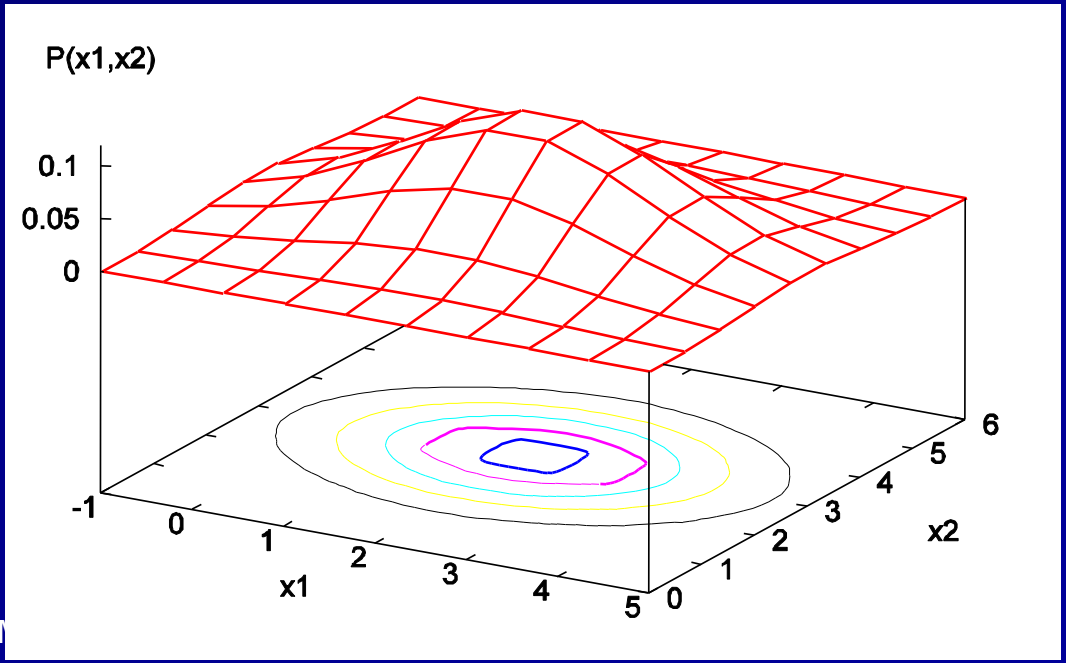


# The Multivariate Gaussian (3)

- If $\Sigma$ is a diagonal matrix, then $x_1$, and $x_2$ are independent random variables, and lines of equal probability are ellipses parallel to the coordinate axes. For example, when

$$\Sigma = \begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix}$$ and

$$\boldsymbol{\mu} = (2,3)$$ we obtain

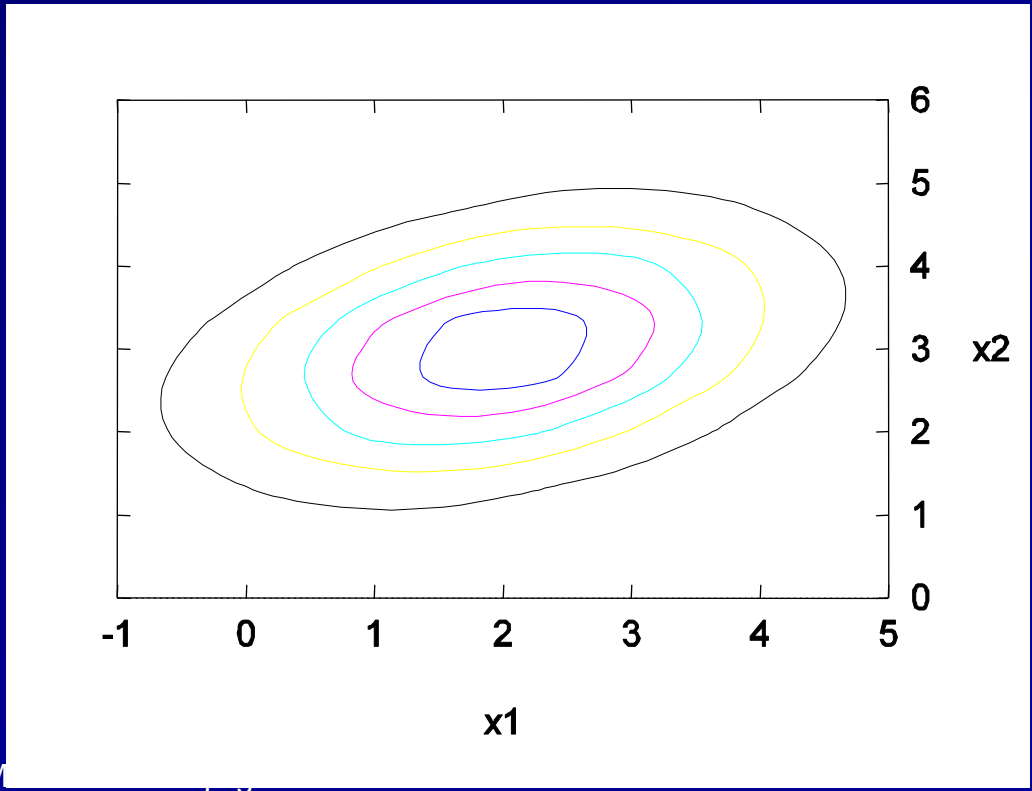


# The Multivariate Gaussian (4)

- Finally, if $\Sigma$ is an arbitrary matrix, then $x_1$ and $x_2$ are dependent, and lines of equal probability are ellipses tilted relative to the coordinate axes. For example, when

$$\Sigma = \begin{bmatrix} 2 & 0.5 \\ 0.5 & 1 \end{bmatrix}$$ and

$$\boldsymbol{\mu} = (2,3)$$ we obtain



# Estimating a Multivariate Gaussian

- Given a set of $N$ data points $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N$, we can compute the maximum likelihood estimate for the multivariate Gaussian distribution as follows:

$$\hat{\boldsymbol{\mu}} = \frac{1}{N}\sum_i \boldsymbol{x}_i \qquad \hat{\Sigma} = \frac{1}{N}\sum_i (\boldsymbol{x}_i - \hat{\boldsymbol{\mu}})(\boldsymbol{x}_i - \hat{\boldsymbol{\mu}})^\top$$

- Note that the dot product in the second equation is an *outer product*. The outer product of two vectors is a matrix:

$$\boldsymbol{x}\boldsymbol{y}^\top = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \begin{bmatrix} y_1 & y_2 & y_3 \end{bmatrix} = \begin{bmatrix} x_1 y_1 & x_1 y_2 & x_1 y_3 \\ x_2 y_1 & x_2 y_2 & x_2 y_3 \\ x_3 y_1 & x_3 y_2 & x_3 y_3 \end{bmatrix}$$

- For comparison, the usual dot product is written as $\boldsymbol{x}^\top \boldsymbol{y}$

# The LDA Model

- Linear discriminant analysis assumes that the joint distribution has the form

$$P(\boldsymbol{x}, y) = P(y)\frac{1}{(2\pi)^{n/2}|\Sigma|^{1/2}}\exp\left(-\frac{1}{2}[\boldsymbol{x}-\boldsymbol{\mu}_y]^{\top}\boldsymbol{\Sigma}^{-1}[\boldsymbol{x}-\boldsymbol{\mu}_y]\right)$$

where each $\boldsymbol{\mu}_y$ is the mean of a multivariate Gaussian for examples belonging to class $y$ and $\Sigma$ is a single covariance matrix *shared by all classes*.

# Fitting the LDA Model

- It is easy to learn the LDA model in a single pass through the data:
  - Let $\hat{\pi}_k$ be our estimate of $P(y = k)$
  - Let $N_k$ be the number of training examples belonging to class $k$.

$$\hat{\pi}_k = \frac{N_k}{N}$$

$$\hat{\boldsymbol{\mu}}_k = \frac{1}{N_k} \sum_{\{i:y_i=k\}} \boldsymbol{x}_i$$

$$\hat{\Sigma} = \frac{1}{N} \sum_i (\boldsymbol{x}_i - \hat{\boldsymbol{\mu}}_{y_i})(\boldsymbol{x}_i - \hat{\boldsymbol{\mu}}_{y_i})^\top$$

- Note that each $\boldsymbol{x}_i$ is subtracted from its corresponding $\boldsymbol{\mu}_{y_i}$ prior to taking the outer product. This gives us the "pooled" estimate of $\Sigma$
- This is known as the Method of Moments

# LDA learns an LTU

- Just as with Logistic Regression, we should classify $\boldsymbol{x}$ into class 1 if $\hat{P}(y = 1|\boldsymbol{x}) > 0.5$
- Our model contains $P(y)$ and $P(\boldsymbol{x}|y)$, so we need to perform probabilistic inference to obtain the condition probability:

$$P(y|\boldsymbol{x}) = \frac{P(\boldsymbol{x}|y)P(y)}{P(\boldsymbol{x})} = \frac{Norm(\boldsymbol{x}; \boldsymbol{\mu}_y, \Sigma)\pi_y}{\sum_k Norm(\boldsymbol{x}; \boldsymbol{\mu}_k, \Sigma)\pi_k}$$

- As before, we re-express this as

$$\frac{P(y = 1|\boldsymbol{x})}{P(y = 0|\boldsymbol{x})} > 1$$

- The denominators cancel, and we have

$$\frac{Norm(\boldsymbol{x}; \boldsymbol{\mu}_1, \Sigma)\pi_1}{Norm(\boldsymbol{x}; \boldsymbol{\mu}_0, \Sigma)\pi_0} > 1$$

# LDA Learns an LTU (2)

- Substitute the formula for $Normal$:

$$\frac{\pi_1 \frac{1}{(2\pi)^{n/2}|\Sigma|^{1/2}} \exp\left(-\frac{1}{2}[\boldsymbol{x}-\boldsymbol{\mu}_1]^\top \boldsymbol{\Sigma}^{-1}[\boldsymbol{x}-\boldsymbol{\mu}_1]\right)}{\pi_0 \frac{1}{(2\pi)^{n/2}|\Sigma|^{1/2}} \exp\left(-\frac{1}{2}[\boldsymbol{x}-\boldsymbol{\mu}_0]^\top \boldsymbol{\Sigma}^{-1}[\boldsymbol{x}-\boldsymbol{\mu}_0]\right)} > 1$$

- Cancel terms, take logs

$$\log\frac{\pi_1}{\pi_0} + [\boldsymbol{x}-\boldsymbol{\mu}_1]^\top \boldsymbol{\Sigma}^{-1}[\boldsymbol{x}-\boldsymbol{\mu}_1] - [\boldsymbol{x}-\boldsymbol{\mu}_0]^\top \boldsymbol{\Sigma}^{-1}[\boldsymbol{x}-\boldsymbol{\mu}_0] > 0$$

- With a bit more work

$$\log\frac{\pi_1}{\pi_0} + 2\boldsymbol{x}^\top \Sigma^{-1}(\boldsymbol{\mu}_1-\boldsymbol{\mu}_0) + \boldsymbol{\mu}_1^\top \Sigma^{-1}\boldsymbol{\mu}_1 - \boldsymbol{\mu}_0^\top \Sigma^{-1}\boldsymbol{\mu}_0 > 0$$

# LDA Learns an LTU (3)

$$\log\frac{\pi_1}{\pi_0} + 2\boldsymbol{x}^\top\Sigma^{-1}(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_0) + \boldsymbol{\mu}_1^\top\Sigma^{-1}\boldsymbol{\mu}_1 - \boldsymbol{\mu}_0^\top\Sigma^{-1}\boldsymbol{\mu}_0 > 0$$

- Define

$$\boldsymbol{w} = 2\Sigma^{-1}(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_0)$$

$$c = \log\frac{\pi_1}{\pi_0} + \boldsymbol{\mu}_1^\top\Sigma^{-1}\boldsymbol{\mu}_1 - \boldsymbol{\mu}_0^\top\Sigma^{-1}\boldsymbol{\mu}_0$$

- Then LDA will classify into $\hat{y} = 1$ iff

$$\boldsymbol{x}^\top\boldsymbol{w} > -c$$

- which is a linear threshold unit

# Two Geometric Views of LDA
# View 1: Mahalanobis Distance

- The quantity $D_M(\boldsymbol{x}, \boldsymbol{u})^2 = [\boldsymbol{x} - \boldsymbol{u}]^\top \boldsymbol{\Sigma}^{-1} [\boldsymbol{x} - \boldsymbol{u}]$ is known as the (squared) Mahalanobis distance between $\boldsymbol{x}$ and $\boldsymbol{u}$. We can think of the matrix $\Sigma^{-1}$ as a linear distortion/rotation of the coordinate system that converts the standard Euclidean distance into the Mahalanobis distance
- Note that

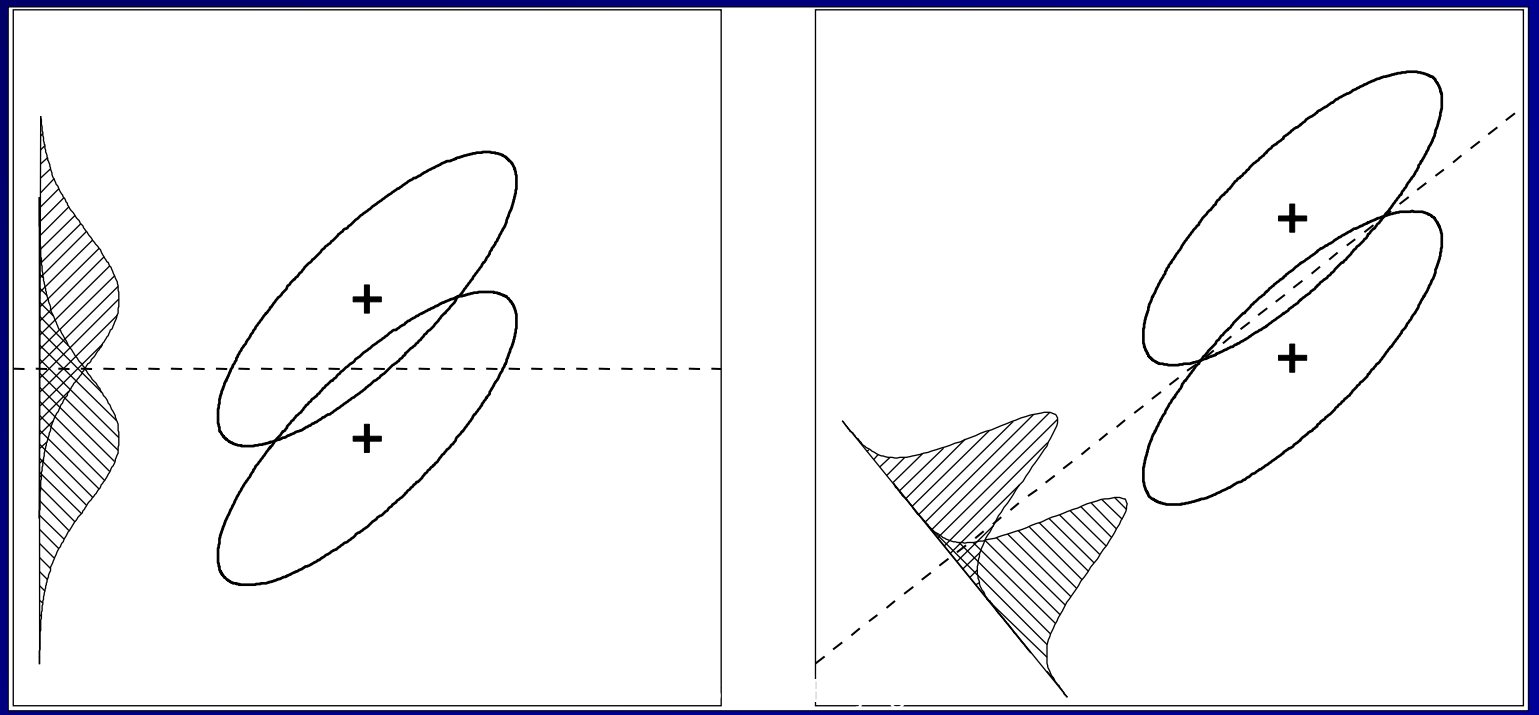
$$\log P(y = k|\boldsymbol{x}) \propto \log \pi_k - \frac{1}{2} [\boldsymbol{x} - \boldsymbol{\mu}_k]^\top \boldsymbol{\Sigma}^{-1} [\boldsymbol{x} - \boldsymbol{\mu}_k]$$

$$\log P(y = k|\boldsymbol{x}) \propto \log \pi_k - \frac{1}{2} D_M(\boldsymbol{x}, \boldsymbol{u})^2$$

- Therefore, we can view LDA as computing $D_M(\boldsymbol{x}, \boldsymbol{\mu}_0)^2 \; and D_M(\boldsymbol{x}, \boldsymbol{\mu}_1)^2$ and then classifying $\boldsymbol{x}$ according to which mean $\boldsymbol{\mu}_0$ or $\boldsymbol{\mu}_1$ is closer in Mahalanobis distance (corrected by log $\pi_k$)

# View 2: Most Informative Low-Dimensional Projection

- LDA can also be viewed as finding a hyperplane of dimension $K - 1$ such that $\boldsymbol{x}$ and the $\{\mu_k\}$ are projected down into this hyperplane and then $\boldsymbol{x}$ is classified to the nearest $\mu_k$ using *Euclidean distance* inside this hyperplane



# Generalizations of LDA

- General Gaussian Classifier
  - Instead of assuming that all classes share the same $\Sigma$, we can allow each class *k* to have its own $\Sigma_k$. In this case, the resulting classifier will be a quadratic threshold unit (instead of an LTU)
- Naïve Gaussian Classifier
  - Allow each class to have its own $\Sigma_k$, but require that each $\Sigma_k$ be diagonal. This means that *within* each class, any pair of features $x_{j1}$ and $x_{j2}$ will be assumed to be statistically independent. The resulting classifier is still a quadratic threshold unit (but with a restricted form)

# Review
# Linear Discriminant Analysis

- We adopted the joint probability approach: $P(\boldsymbol{x}, y)$
- We adopted the log loss as a surrogate for 0/1 loss
- We fit the model directly, via the method of moments

# Comparing Perceptron, Logistic Regression, and LDA

- How should we choose among these three algorithms?
- There are several trade-offs
- There is a big debate in the machine learning community!

# Issues in the Debate

- Statistical Efficiency. If the generative model $P(\boldsymbol{x}, y)$ is correct, then LDA usually gives the highest accuracy, particularly when the amount of training data is small. If the model is correct, LDA requires 30% less data than Logistic Regression in theory

- Computational Efficiency. Generative models typically are the easiest to learn. In our example, the LDA parameters can be computed directly from the data without using gradient descent.

# Issues in the Debate

- Robustness to changing loss functions. Both generative and conditional probability models allow the loss function to be changed at run time without re-learning. Perceptron requires re-training the classifier when the loss function changes.
  - Probabilistic modelling separates model learning from making predictions or decisions
  - Suppose the cost of a false positive SPAM prediction is 10 whereas a false negative is 1
  - The cost of classifying $x$ as $y = 1$ is $P(y = 0|\boldsymbol{x}) \times 10$
  - The cost of classifying $x$ as $y = 0$ is $P(y = 1|\boldsymbol{x}) \times 1$
  - So we can choose the value of $y$ that minimizes the expected cost

# Issues in the Debate

- Vapnik's Principle
  - If your goal is to minimize 0/1 loss, then you should do that directly, rather than first solving a harder problem (probability estimation)
  - This is what Perceptron does
  - Other algorithms that follow this principle
    - Support Vector Machines
    - Decision Trees
    - Neural Networks

# Issues in the Debate

- Robustness to model assumptions. The generative model usually performs poorly when the assumptions are violated. For example, if $P(\boldsymbol{x}|y)$ is very non-Gaussian, then LDA won't work well. Logistic Regression is more robust to model assumptions, and Perceptron is even more robust.

  Consequently, making the generative approach work often requires more detailed modeling of $P(\boldsymbol{x}|y)$.

- Robustness to missing values and noise. In many applications, some of the features $x_{ij}$ may be missing or corrupted in some of the training examples. Generative models typically provide better ways of handling this than non-generative models.

# Questions?

# Some Questions to Think About

- Machine learning has many powerful non-linear classifiers, why didn't you discuss those?
  - Linear methods are surprisingly powerful, especially in computer vision and natural language processing where the number of features is very large
- You have showed how to fit these models to training data, but that doesn't guarantee that they will make good predictions on new data points
  - Excellent question! That is the subject of Part 2

# Break

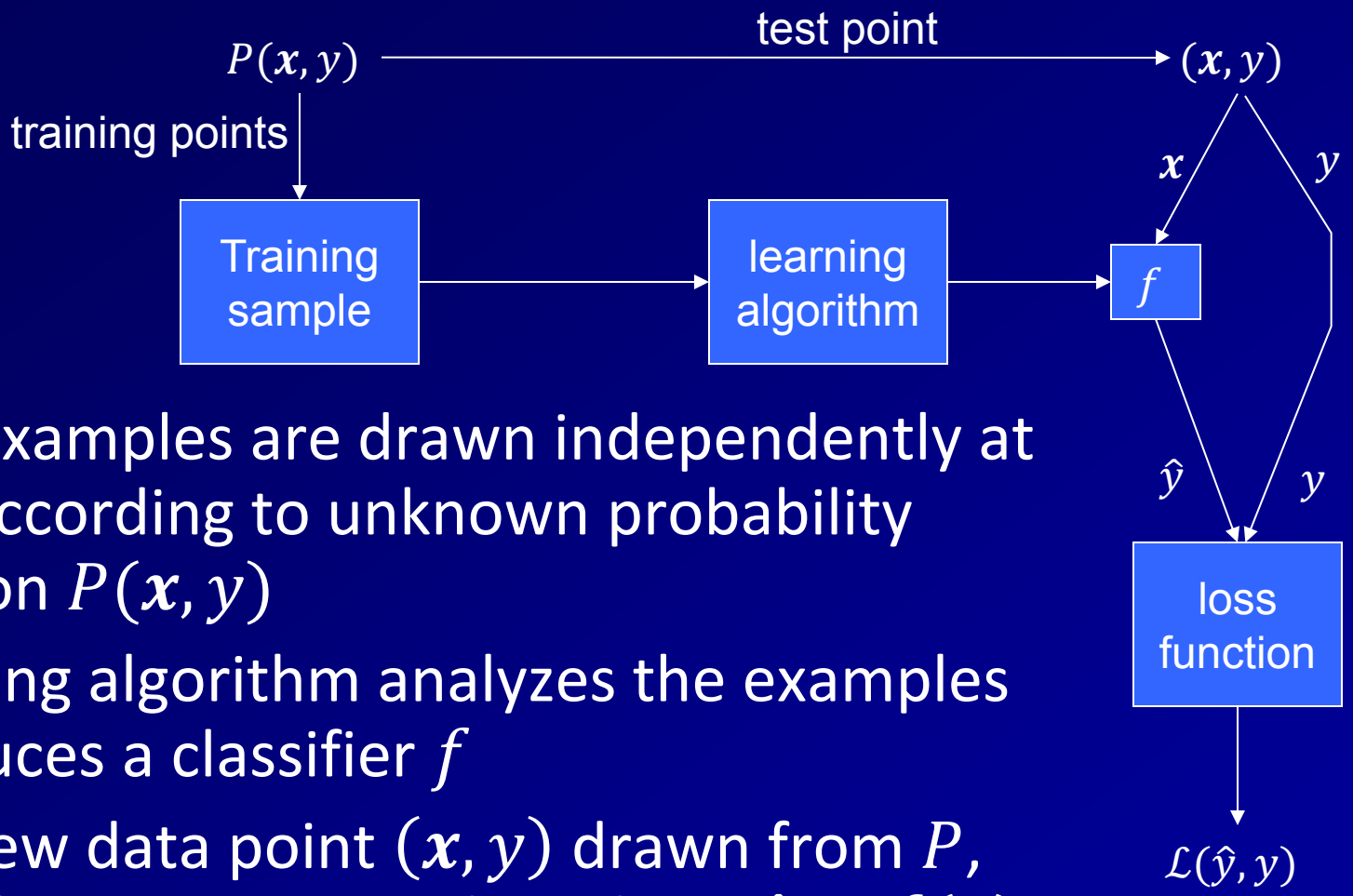
# Introduction to Machine Learning Part 2

Thomas G. Dietterich
tgd@eecs.oregonstate.edu

# Outline

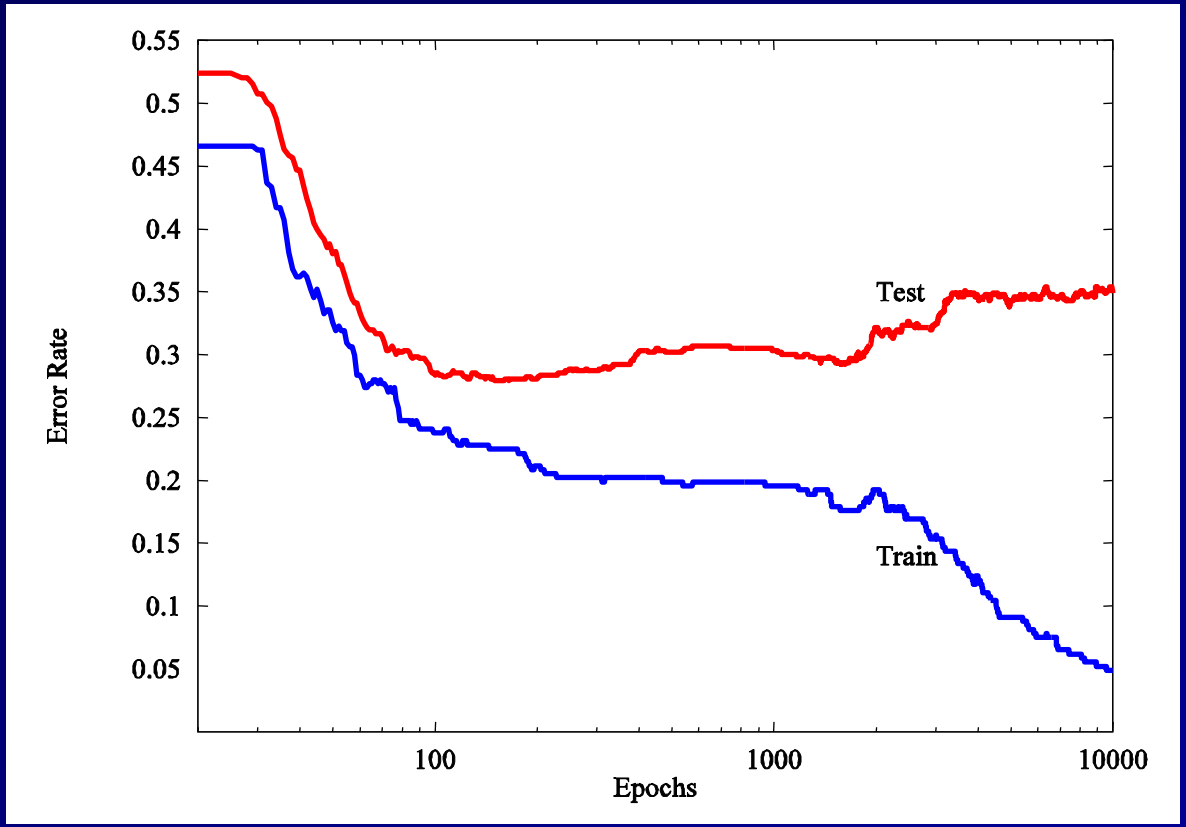
- What is Machine Learning?
- Introduction to Supervised Learning: Linear Methods
- Overfitting, Regularization, and the Bias-Variance Tradeoff
- Review and Summary

# Formal Setting




- Training examples are drawn independently at random according to unknown probability distribution $P(\boldsymbol{x}, y)$
- The learning algorithm analyzes the examples and produces a classifier $f$
- Given a new data point $(\boldsymbol{x}, y)$ drawn from $P$, the classifier is given $\boldsymbol{x}$ and predicts $\hat{y} = f(\boldsymbol{x})$
- The loss $\mathcal{L}(\hat{y}, y)$ is then measured
- Goal of the learning algorithm: Find the $f$ that minimizes the *expected loss* on new points

# The Problem of Overfitting

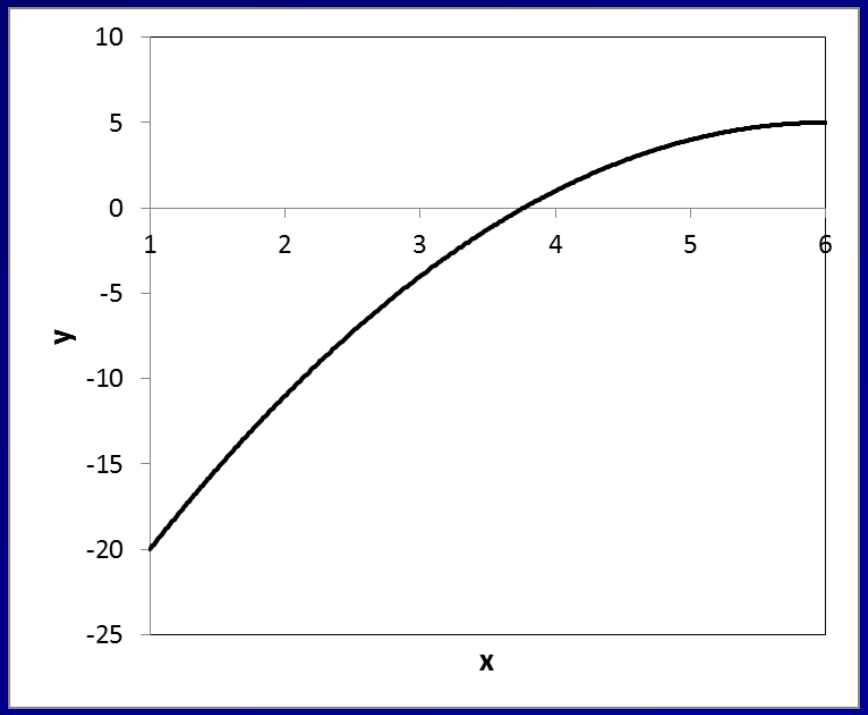



- Model: Neural Network
- Epoch: One batch gradient descent step
- After about 180 epochs, error on the test data starts to increase even though the model continues to become more accurate on the training data

Lesson:
It is not enough to minimize the loss on the training data
if our goal is to optimize accuracy on *new* data points

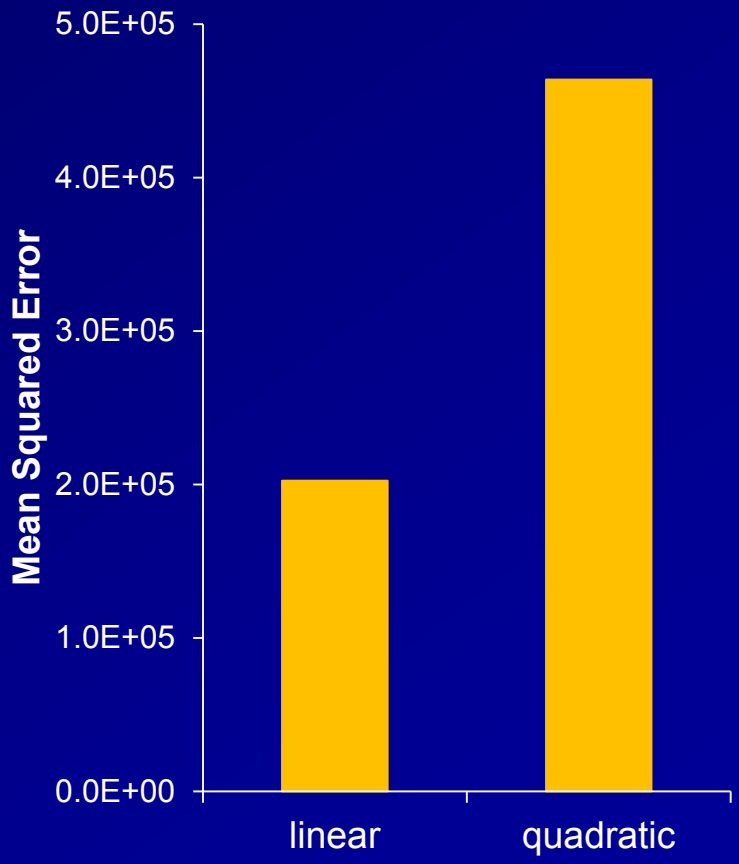
# Another Example

- True function:
  $y = f(x_1) = -(x_1 - 6)^2 + 5$
- Training data:
  $x_1 \sim unif(1,6)$
  $x_2 \sim unif(-6,6)$
  $y = f(x_1) + norm(0,10)$
- Two models:
  Model 1: $y = w_0 + w_1 x_1 + w_2 x_2$
  Model 2: $y = w_0 + w_1 x_1 + w_{12} x_1^2 + w_2 x_2 + w_{22} x_2^2$



# Small, Noisy Training Set

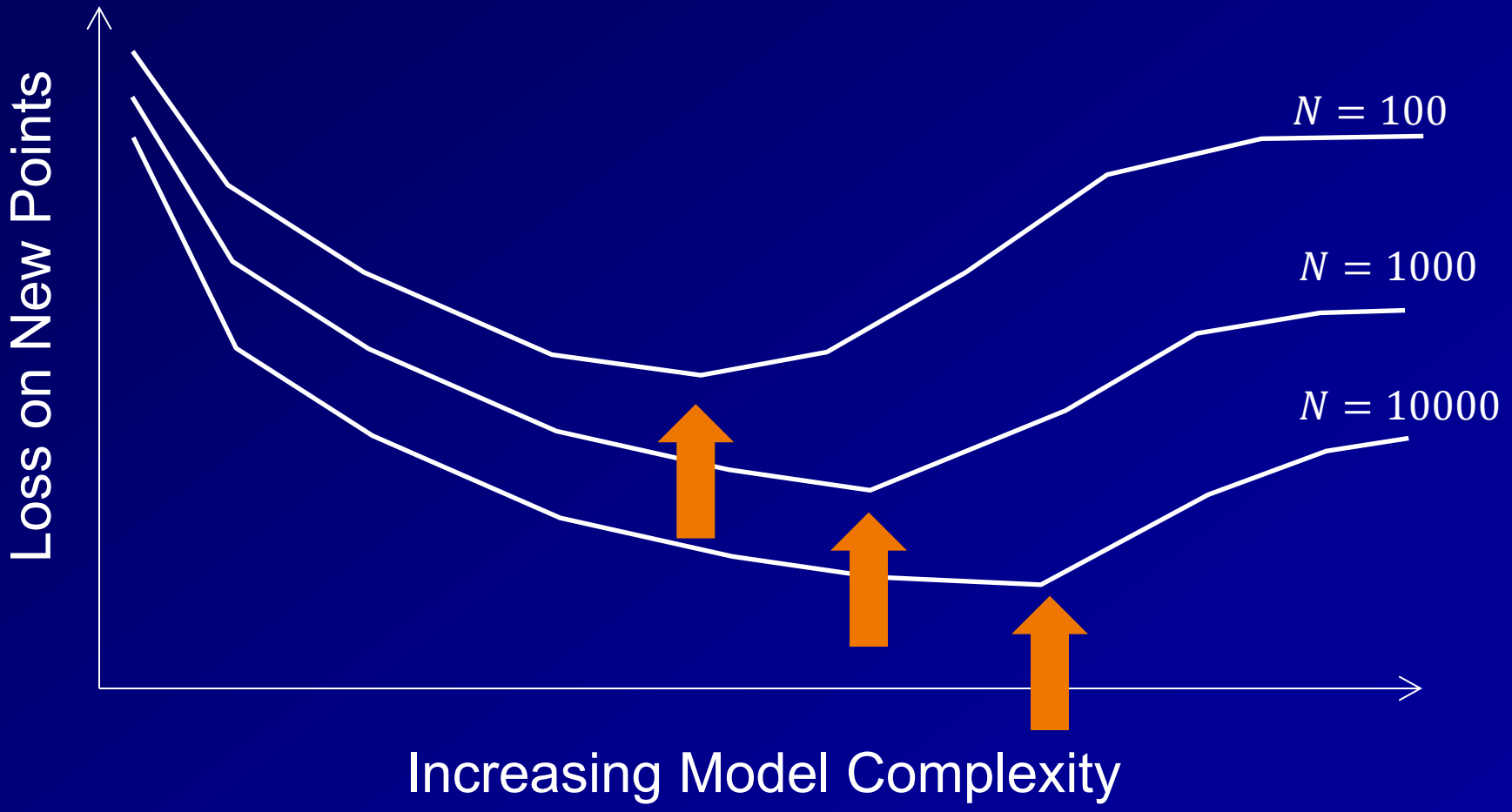
- 3 training examples
- Linear model is much more accurate even though
  - It *cannot* represent the true model
  - The quadratic model *can* represent the true model



# The Three-Way Tradeoff

- There is a tradeoff between
  - amount of data
  - complexity of the model fit to the data
  - accuracy of the model on new data points

# Three-way Tradeoff



# What is Model Complexity?

- Parametric models: Complexity = number of weights
  - Linear model is less complex than a quadratic model
  - Extra features → extra complexity
- More subtle:
  - Weights that are zero don't contribute to complexity
  - Small weights contribute less complexity that large weights

# Controlling Model Complexity Via Regularization

- *Regularization*: Penalize the magnitude of the weights in the model
- Example: "square" penalty

$$J(\boldsymbol{w}) = \frac{1}{N}\sum_i \mathcal{L}(f(\boldsymbol{x}_i;\boldsymbol{w}), y_i) + \lambda \sum_j w_j^2$$

The severity of the penalty is controlled by $\lambda > 0$

Adds a term of $2\lambda \sum_j w_j$ to the gradient descent

In neural networks, this is called "weight decay"

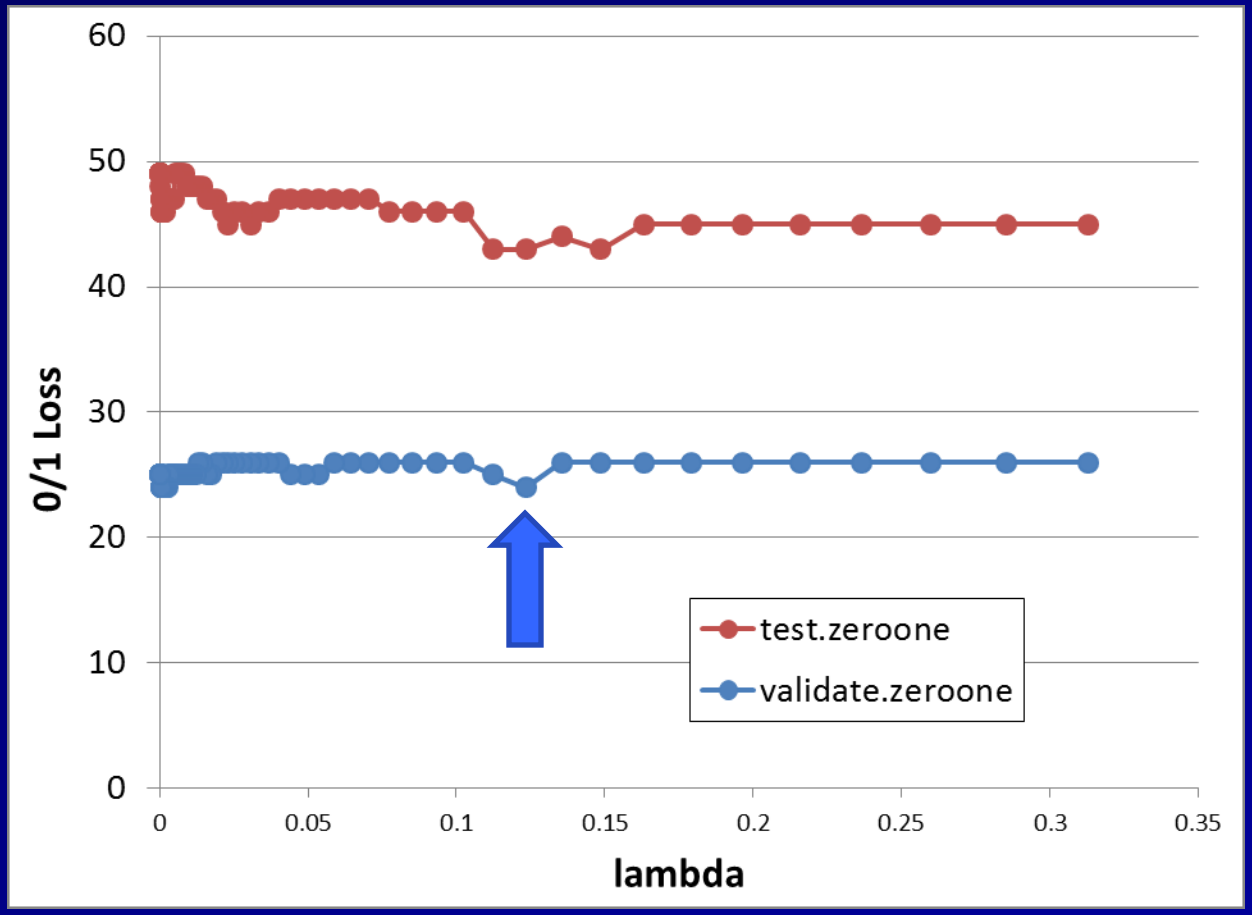
# How to choose $\lambda$?

- Simple Holdout Method
- Subdivide training data $S$ into two subsets: $S_{train}$ and $S_{holdout}$.
- Choose a set of candidate $\lambda$ values = $\lambda_0, \lambda_1, \ldots, \lambda_R$
- Minimize the penalized loss on $S_{train}$, measure the prediction loss on $S_{holdout}$
- Choose the $\lambda$ value that gives the smallest prediction loss

# Simple Holdout Example

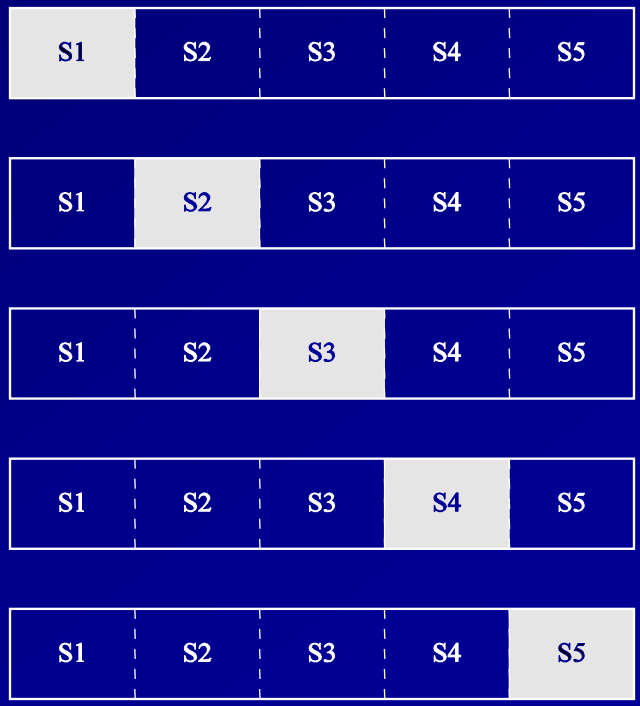
Logistic regression with regularization penalty

Overfitting was not too severe in this problem



# $k$-fold Cross-Validation to determine $\lambda$

- To evaluate a value for $\lambda$
- Randomly divide $S$ into $k$ equal-sized subsets
- Run learning algorithm $k$ times, each time use one subset for $S_{eval}$ and the rest for $S_{train}$
- Compute the average loss on $S_{eval}$
- Choose the $\lambda$ value with minimum loss



# A Bayesian Perspective

- A fully-generative Bayesian story:
- Generate the weights: $\boldsymbol{w} \sim P(\boldsymbol{w})$
- For each data point $i$

  Generate the class label: $y_i \sim P(y_i)$

  Generate the features: $\boldsymbol{x}_i \sim P(\boldsymbol{x}_i | y_i, \boldsymbol{w})$

  Assemble the data set: $S = \{(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_N, y_N)\}$

# Inferring the most likely $\boldsymbol{w}$

$$\hat{\boldsymbol{w}} = \arg\max_{\boldsymbol{w}} P(\boldsymbol{w}|S)$$

$$P(w|S) = P(\boldsymbol{w}) \prod_i P(y_i) P(\boldsymbol{x}_i|y_i, \boldsymbol{w})$$

$$\log P(\boldsymbol{w}|S) = \log P(\boldsymbol{w}) + \sum_i \log P(y_i) + \log P(\boldsymbol{x}_i|y_i, \boldsymbol{w})$$

$$\hat{\boldsymbol{w}} = \arg\max_{\boldsymbol{w}} \ell(S, \boldsymbol{w}) + \log P(\boldsymbol{w})$$

The regularization penalty is the same as the log prior on the weights

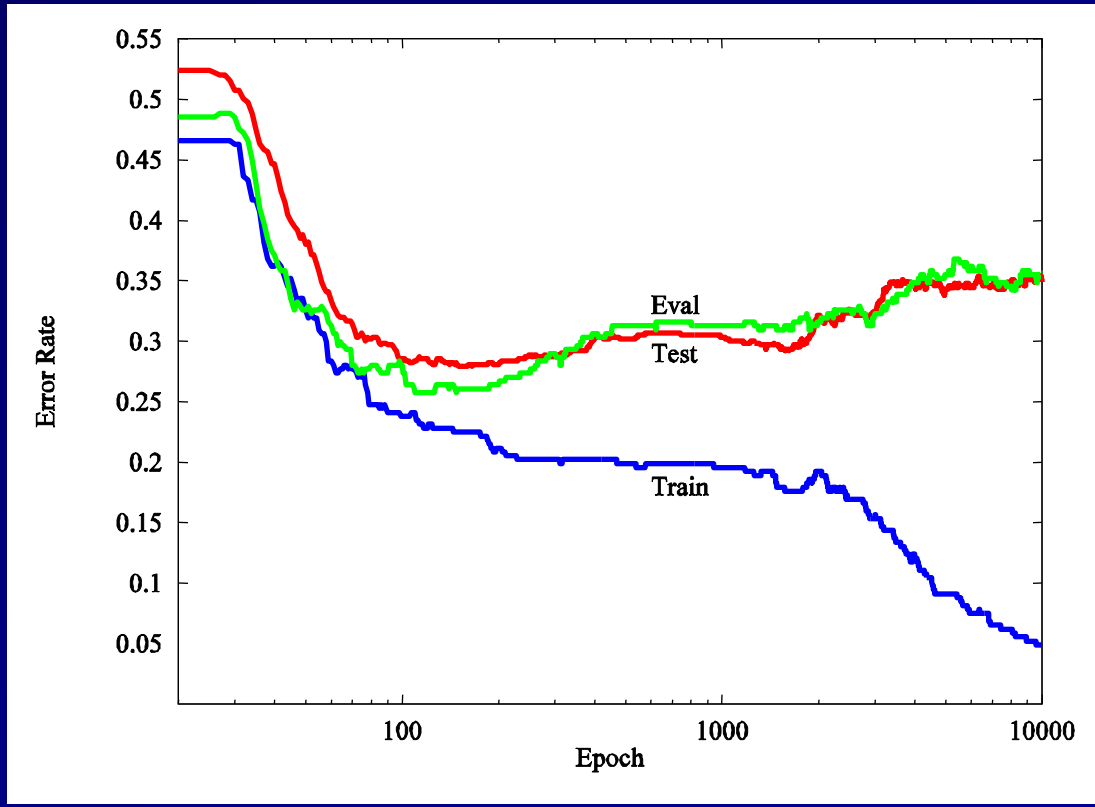
This provides a way of incorporating different penalties on different weights in a model, based on prior knowledge

# Controlling Complexity by Early Stopping

- Incrementally add complexity
  - monitor performance on $S_{eval}$
  - stop when performance drops
- Examples:
  - incrementally adding variables in regression
  - growing a decision tree
  - early stopping in stochastic gradient descent

# Early Stopping

- Subdivide $S$ into $S_{train}$ and $S_{eval}$
- Initialize weights to 0
- Perform gradient descent on $S_{train}$, which gradually causes the weights to grow (in magnitude)
- Measure the loss on $S_{eval}$ after each gradient step
- Stop when the loss starts to increase



# Other Methods for Controlling Complexity

- Model pruning
  - over-fit a model, then prune/shrink
- Adding noise
  - to the inputs or to intermediate quantities
  - makes it harder to overfit
- Ensembles (see below)

# Bias-Variance Analysis

- An alternative view of over-fitting and complexity control
- Bias: Systematic error in the model
  - typically caused by an inability to express the full complexity of the data
- Variance: Variability in the fitted model
  - typically caused by having a model that is too complex for the amount of data
- The total error of the model can be partitioned into the sum of a bias term and a variance term
- As we increase $\lambda$ we increase bias but reduce variance

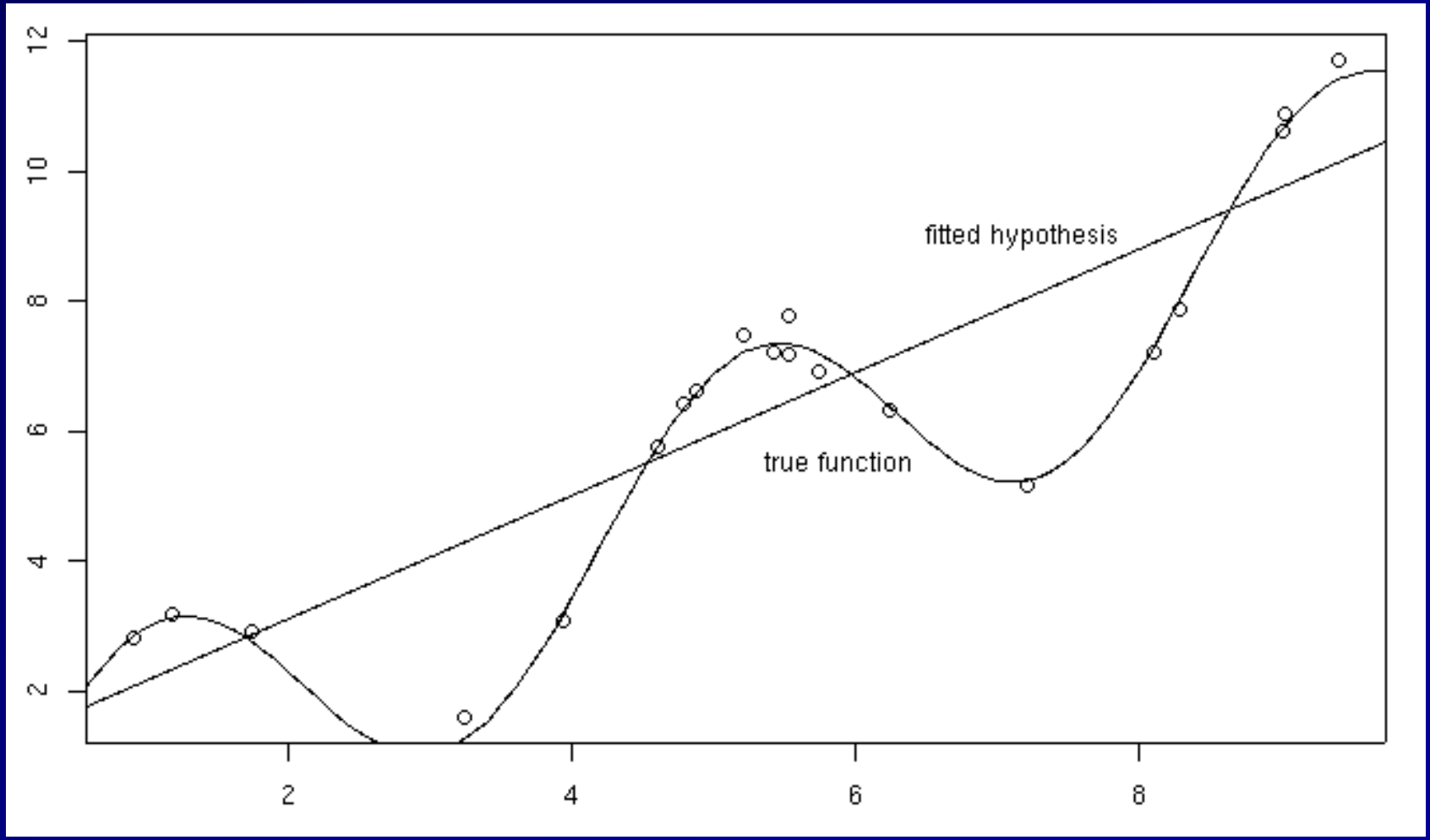
# Bias-Variance Analysis of Regression

- True function: $y = f(\boldsymbol{x}) + \epsilon$
  - where $\epsilon \sim Norm(0, \sigma^2)$
- We are given a set $S = \{(x_1, y_1), \ldots, (x_N, y_N)\}$
- We fit a model $h(x)$ to minimize the square loss

$$\mathcal{L}(\hat{y}, y) = (\hat{y} - y)^2$$

$$J(h) = \sum_i (\hat{y}_i - y_i)^2$$

# Example: $N = 20$ points
# $y = x + 2\sin(1.5x) + Norm(0,0.2)$



# Bias-Variance Analysis

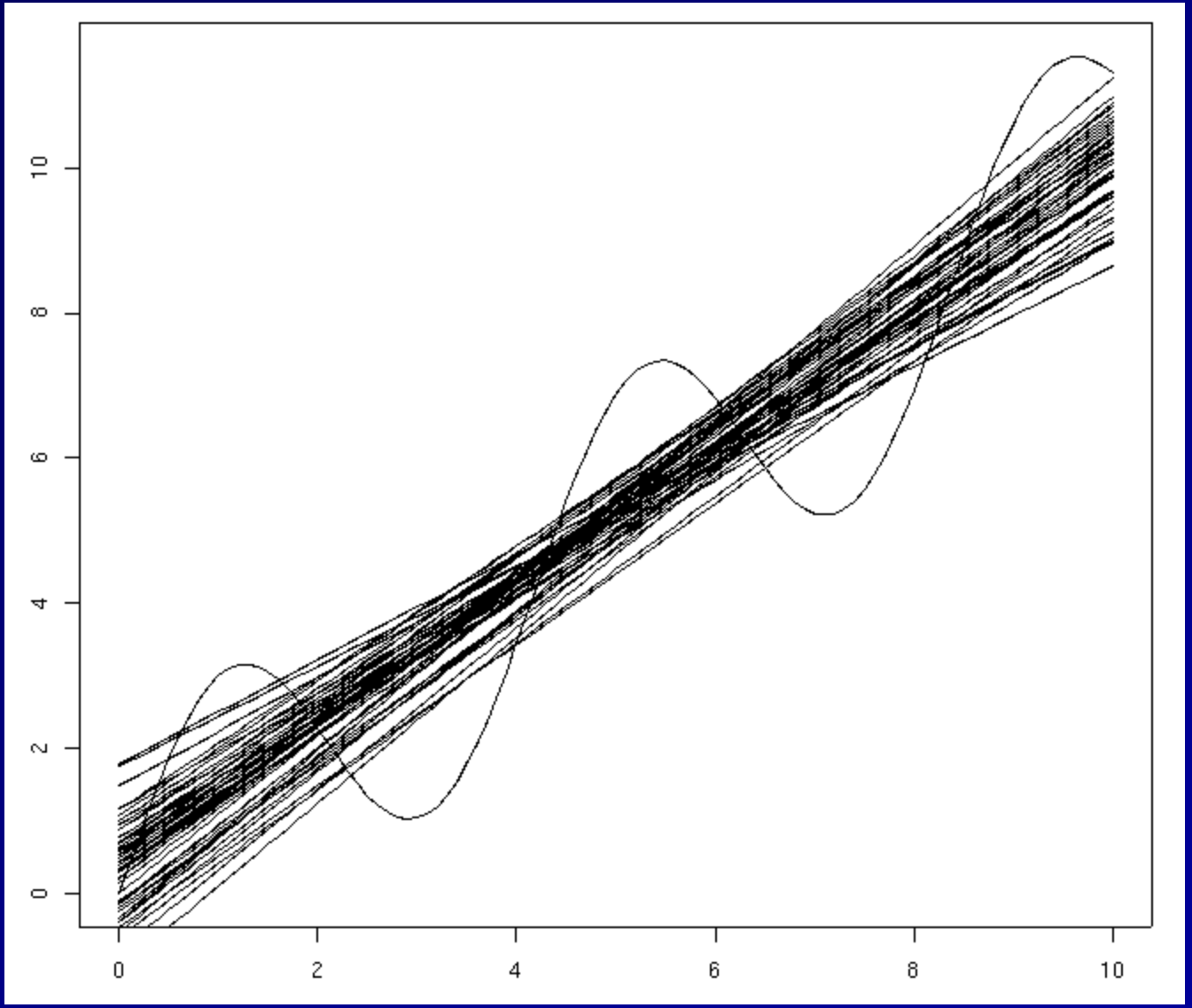
- Given a new data point $(x^*, y^*)$ (with predicted value $h(x^*)$), we would like to decompose our error

$$(y^* - h(x^*))^2$$

# Classical Statistical Analysis

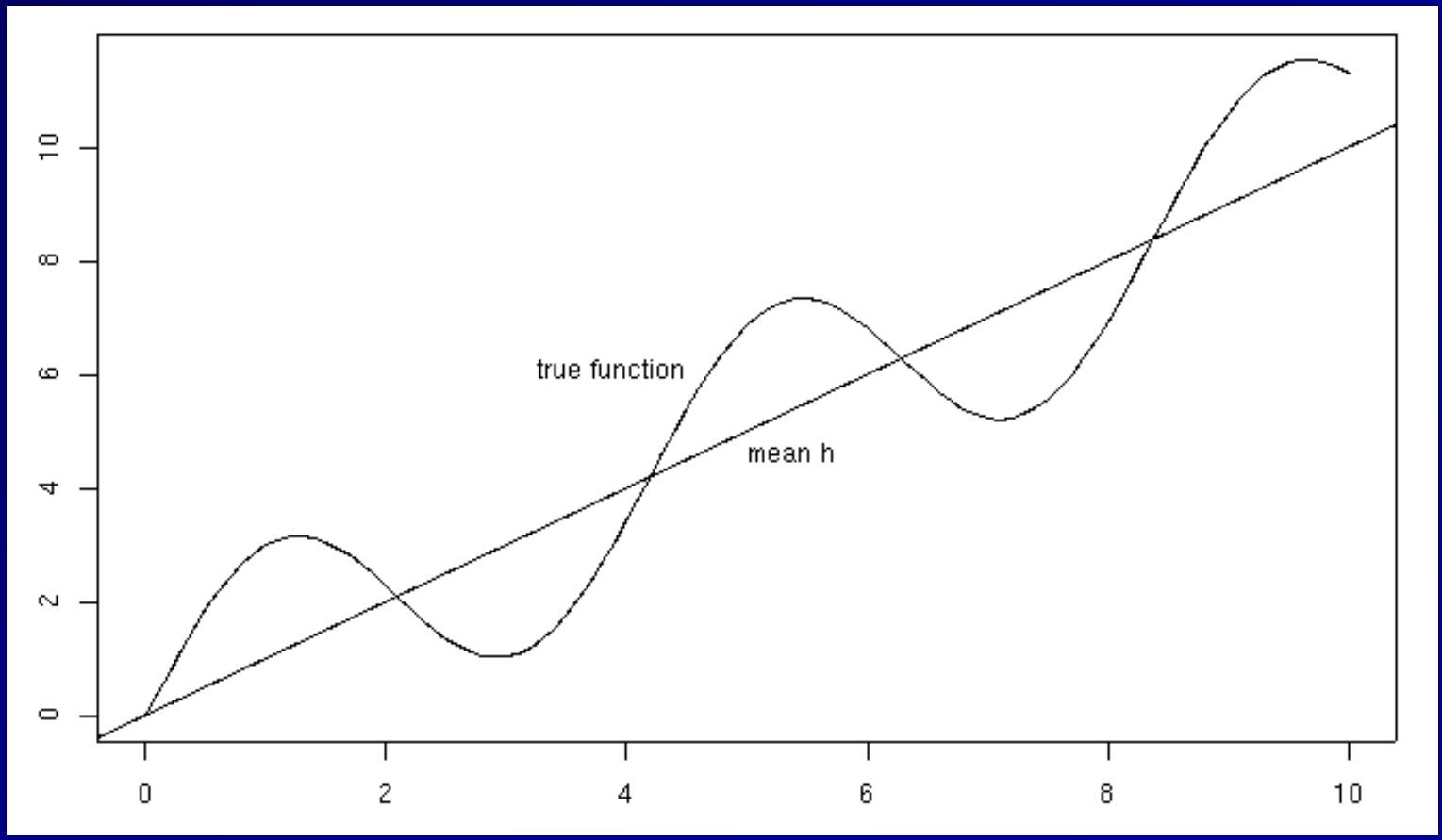
- Imagine that our particular training sample $S$ is drawn from some population of possible training samples according to $P(S)$.
- We fit $h$ to $S$ ($h$ is a random quantity)
- Compute $\mathbb{E}_P[(y^* - h(x^*))^2]$
- Decompose this into "bias", "variance", and "noise"

# 50 fits (20 examples each)



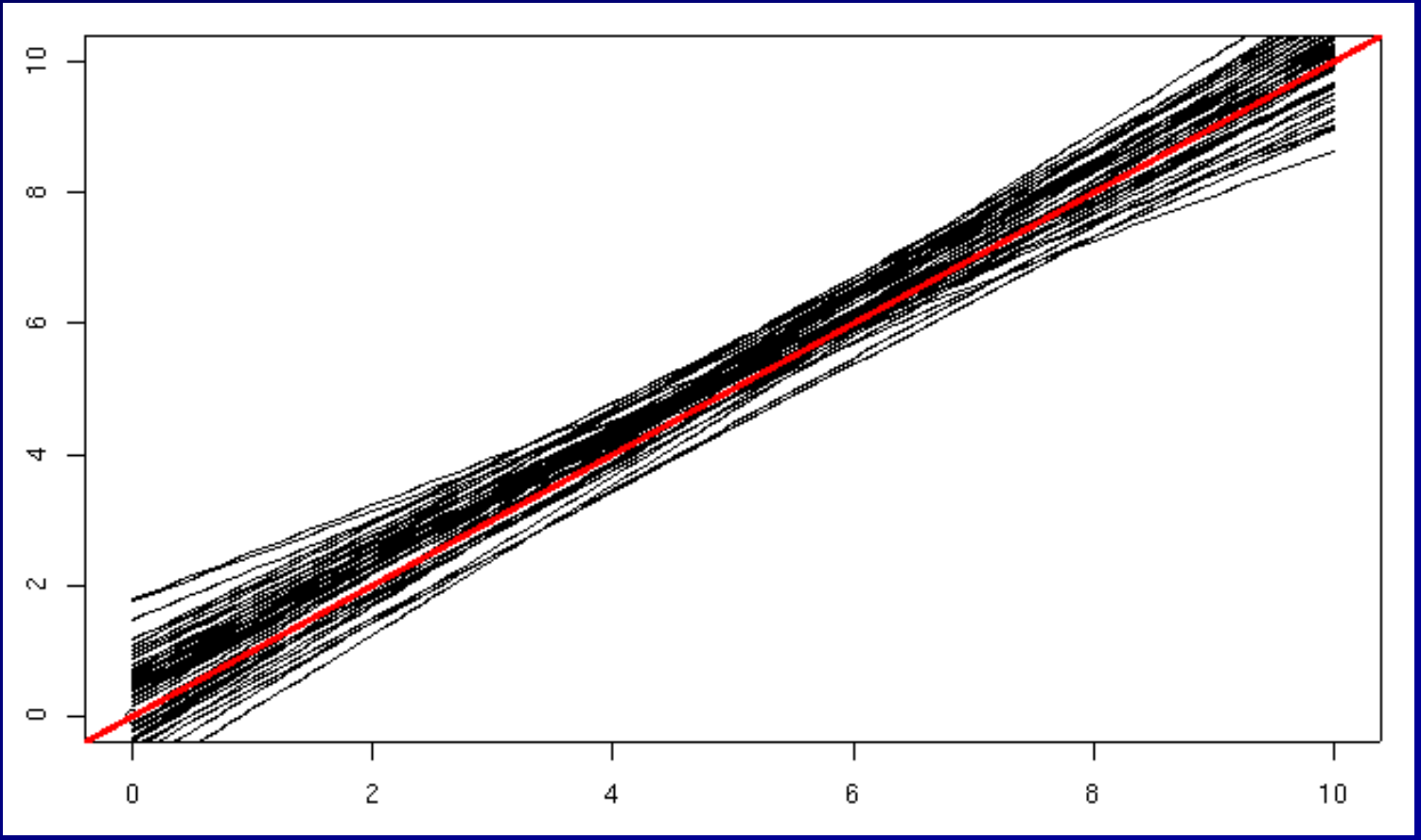
# Bias

- The difference between the average predicted value $\mathbb{E}_P[h(x^*)]$ and the value of the true function $f(x^*)$: $\mathbb{E}_P[h(x^*)] - f(x^*)$



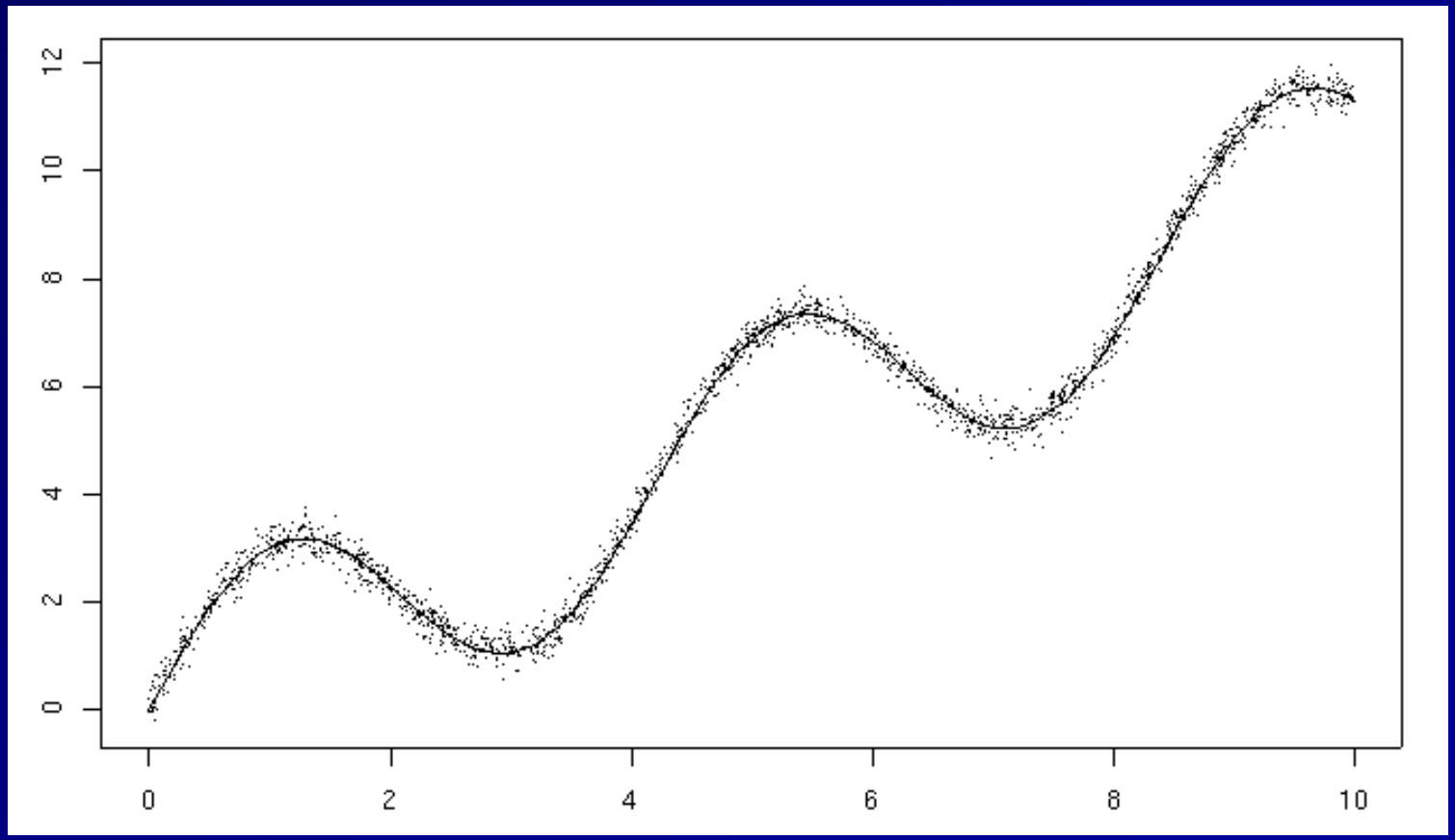
# Variance

- The variance of $h(x^*)$ around its average value $\mathbb{E}_P[h(x^*)]$: $\mathbb{E}_P[(h(x^*) - \mathbb{E}_P[h(x^*)])^2]$



# Noise

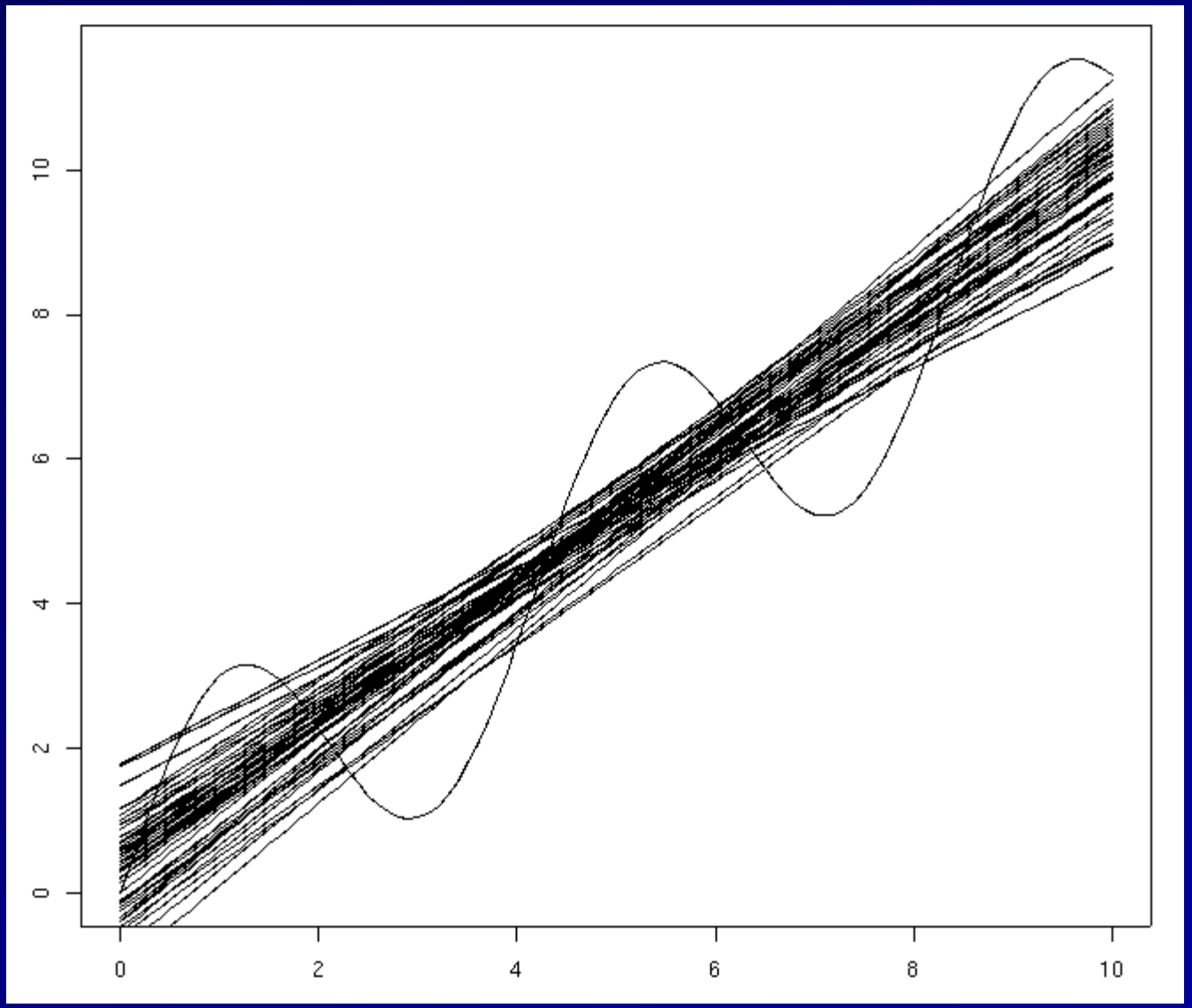
- The variation of $y^*$ around its true average value $f(x^*)$: $(y^* - f(x^*))^2$



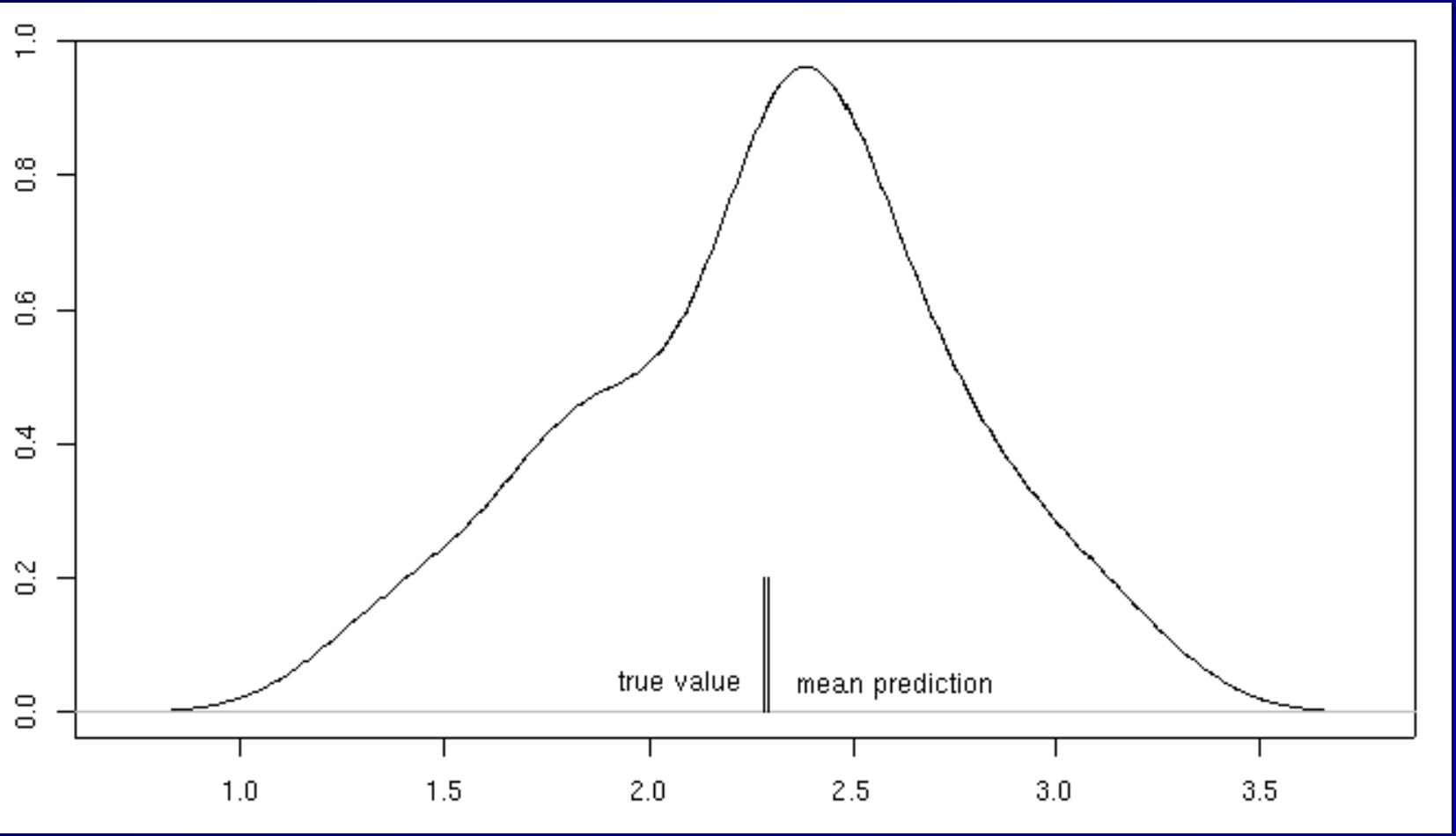
# The Bias-Variance Decomposition

$$\mathbb{E}_h[(y^* - h(x^*))^2]$$
$$= \mathbb{E}_h[(h(x^*) - \mathbb{E}_h[h(x^*)])^2] \quad \text{variance}$$
$$+ (\mathbb{E}_h[h(x^*)] - f(x^*))^2 \quad \text{squared bias}$$
$$+ (y^* - f(x^*))^2 \quad \text{noise}$$

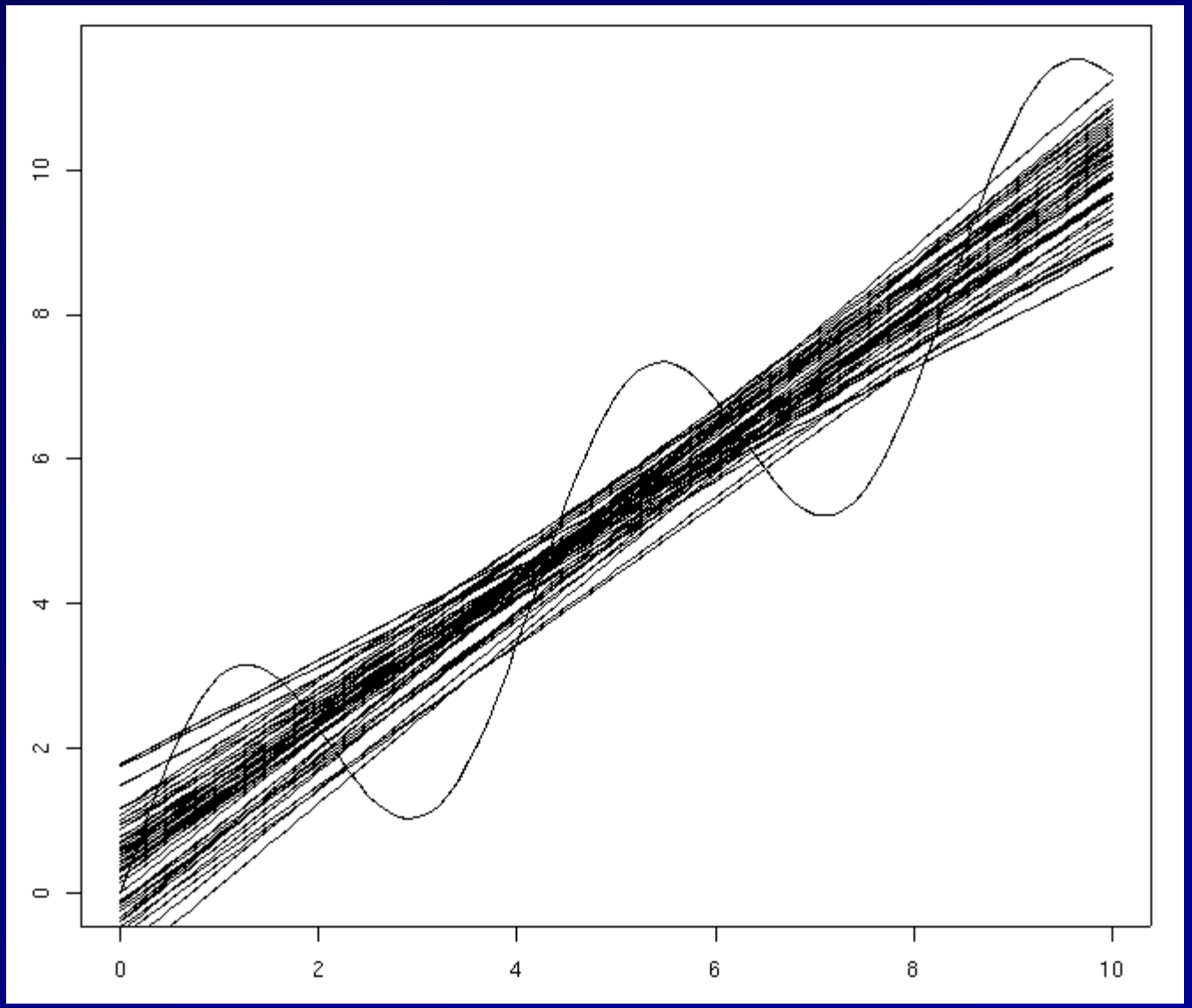
# 50 fits (20 examples each)



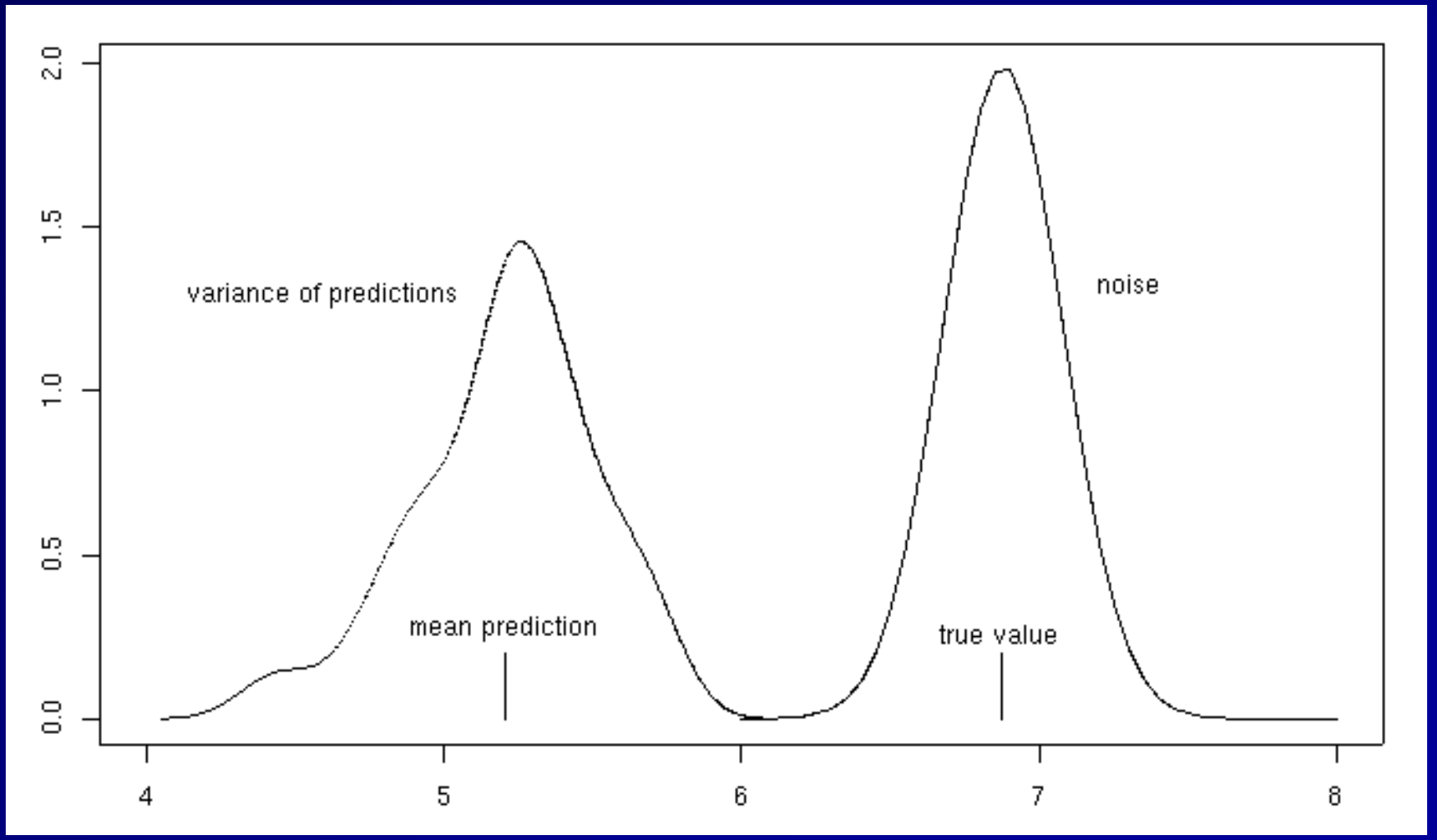
# Distribution of predictions at x=2.0



# 50 fits (20 examples each)



# Distribution of predictions at x=5.0



# Derivation

Let $Z$ be a random variable with distribution $P(Z)$

Let $\overline{Z} = \mathbb{E}[Z]$ be the expected value of $Z$

Lemma: $\mathbb{E}\left[\left(Z - \overline{Z}\right)^2\right] = \mathbb{E}[Z^2] - \overline{Z}^2$

Corollary: $\mathbb{E}[Z^2] = \mathbb{E}\left[\left(Z - \overline{Z}\right)^2\right] + \overline{Z}^2$

# Proof

$$\mathbb{E}\left[\left(Z-\overline{Z}\right)^2\right] = \mathbb{E}\left[Z^2 - 2Z\overline{Z} + \overline{Z}^2\right]$$

$$= \mathbb{E}[Z^2] - 2\overline{Z}\mathbb{E}[Z] + \overline{Z}^2$$

$$= \mathbb{E}[Z^2] - 2\overline{Z}\overline{Z} + \overline{Z}^2$$

$$= \mathbb{E}[Z^2] - \overline{Z}^2$$

# Derivation of the Decomposition

Expand the quadratic:

$$\mathbb{E}_P[(h(x^*) - y^*)^2] = \mathbb{E}_P\left[h(x^*)^2 - 2h(x^*)y^* + {y^*}^2\right]$$

Push the expectation inside

$$= \mathbb{E}_P[h(x^*)^2] - 2\mathbb{E}_P[h(x^*)]\mathbb{E}_P[y^*] + \mathbb{E}_P\left[{y^*}^2\right]$$

Apply the lemma twice

$$= \mathbb{E}_P[(h(x^*) - \mathbb{E}_P[h(x^*)])^2] + \mathbb{E}_P[h(x^*)]^2 - 2\mathbb{E}_P[h(x^*)]f(x^*)$$
$$+ \mathbb{E}_P[(y^* - f(x^*))^2] + f(x^*)^2$$

Collapse the quadratic to get the squared bias term

$$= \mathbb{E}_P[(h(x^*) - \mathbb{E}_P[h(x^*)])^2] + (\mathbb{E}_P[h(x^*)] - f(x^*))^2$$
$$+ \mathbb{E}_P[(y^* - f(x^*))^2]$$

Note that we are also taking expectations wrt the noise $\epsilon$

# Measuring Bias and Variance

- In practice (unlike in theory), we have only ONE training set S.
- We can simulate multiple training sets by bootstrap replicates

  S' = {x | x is drawn at random with replacement from S} and |S'| = |S|.

# Procedure for Measuring Bias and Variance

- Construct B bootstrap replicates of $S$ (e.g., $B = 200$): $S_1, \ldots, SB$
- Apply learning algorithm to each replicate $S_b$ to obtain hypothesis $h_b$
- Let $T_b = S \setminus S_b$ be the data points that do not appear in $S_b$ (out of bag points)
- Compute predicted value $h_b(x)$ for each $x$ in $T_b$

# Estimating Bias and Variance (continued)

- For each data point $x$, we will now have the observed corresponding value $y$ and several predictions $y_1, \ldots, y_K$.
- Compute the average prediction $\bar{y} = \frac{1}{K}\sum_k y_k$.
- Estimate bias as $\bar{y} - y$
- Estimate variance as $\frac{1}{K-1}\sum_k (y_k - \bar{y})^2$
- Assume noise is 0

# Approximations in this Procedure

- Bootstrap replicates are not real fresh data
- We ignore the noise
  - If we have multiple data points with the same $x$ value, then we can estimate the noise
  - We can also estimate noise by pooling $y$ values from nearby $x$ values

# Applying Bias-Variance Analysis

- By measuring the bias and variance on a problem, we can determine how to improve our model
  - If bias is high, we need to allow our model to be more complex
  - If variance is high, we need to reduce the complexity of the model
- Bias-variance analysis also suggests a way to reduce variance: bagging

# Ensemble Learning Methods

- Given training sample $S$
- Generate multiple hypotheses, $h_1, h_2, \ldots, h_L$.
- Optionally: determining corresponding weights $\alpha_1, \alpha_2, \ldots, \alpha_L$
- Classify new points according to

$$\sum_\ell \alpha_\ell h_\ell(x) > \theta$$

"weighted majority vote"

# Bagging: Bootstrap Aggregating

- For $b = 1, \ldots, B$ do

  $S_b$ = bootstrap replicate of $S$

  Apply learning algorithm to $S_b$ to learn $h_b$

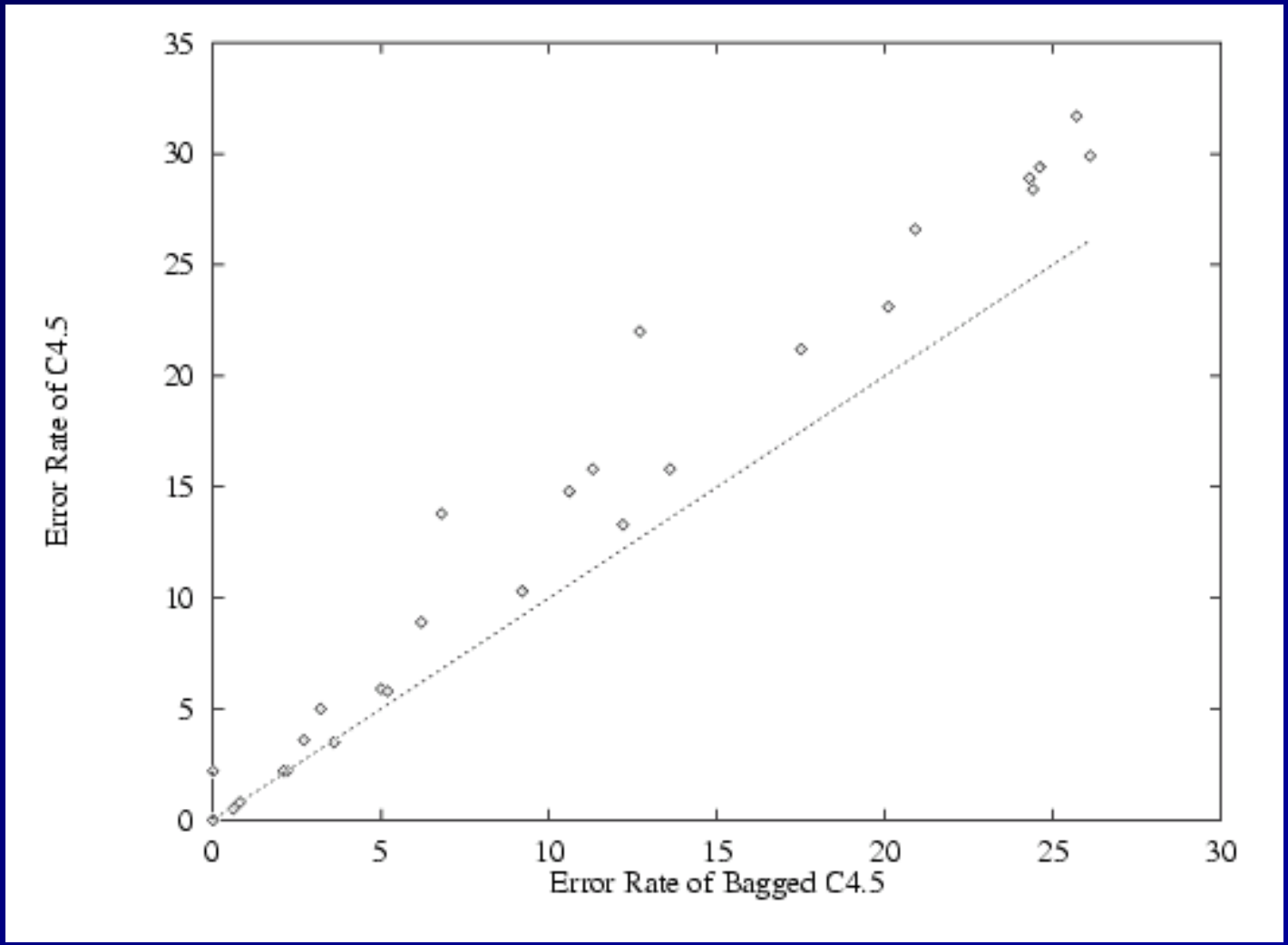
- Classify new points by unweighted vote:

$$\frac{1}{B}\sum_b h_b(x) > 0$$

# Estimated Bias and Variance of Bagging

- If we estimate bias and variance using the same B bootstrap samples, we will have:
  - Bias = $\bar{y} - y$ [same as before]
  - Variance = $\frac{1}{\mathrm{K}-1}\sum_k (\bar{y} - \bar{y})^2 = 0$
- Hence, according to this approximate way of estimating variance, bagging removes the variance while leaving bias unchanged.
- In reality, bagging only *reduces* variance and tends to slightly increase bias

# Bagging Decision Trees (Freund & Schapire)



# Bayesian Ensembles: Bayesian Model Averaging

- $P(y^*|x^*, S) = \int_{\boldsymbol{w}} P(\boldsymbol{w})P(S|\boldsymbol{w})P(y^*|x^*, \boldsymbol{w})d\boldsymbol{w}$
- where $P(S|w) = \prod_i P(y_i|\boldsymbol{w})P(\boldsymbol{x}_i|y_i, \boldsymbol{w})$

- This is rarely practical to evaluate, but suppose we could sample some good values for $\boldsymbol{w}$: $(\boldsymbol{w}_1, \ldots, \boldsymbol{w}_C)$
- We could approximate the integral by a sum:

$$P(y^*|x^*, S) = \sum_c P(\boldsymbol{w}_c|S)P(y^*|x^*, \boldsymbol{w}_c)$$

- This is called Bayesian Model Averaging

# Review of Part 2

- Goal: making accurate predictions on *new* data points
  - the problem of Overfitting
  - occurs when the model becomes too complex for the amount of data
- Complexity can be controlled
  - regularization penalty
  - early stopping
  - choosing $\lambda$ by holdout or cross-validation
- Bias-variance error decomposition
  - Squared loss can be decomposed into bias^2+variance+noise
  - bias and variance can be (approximately) measured using bootstrapping
  - they provide a diagnostic tool for machine learning
- Bagging is an ensemble method that applies bootstrapping to reduce variance
  - Bagging a low-bias, high-variance classifier can produce excellent results
- Bayesian analysis provides an alternative view of
  - regularization penalty = log prior
  - ensemble methods = integrating out the prior

# Questions?

# Questions to think about

- Most machine learning methods involve complex, flexible models
  - decision trees
  - support vector machines
  - neural networks (esp. deep ones)

  Hence, complexity control (variance management) is a central challenge

  "drop out" is a cool new technique in this area
- The bias-variance analysis was done for regression. Can it be extended to classification?
  - Yes, see James (2003) "Variance and bias for general loss functions" *Machine Learning*.