CS440/ECE448: Intro to Artificial Intelligence

Lecture 24: Perceptrons

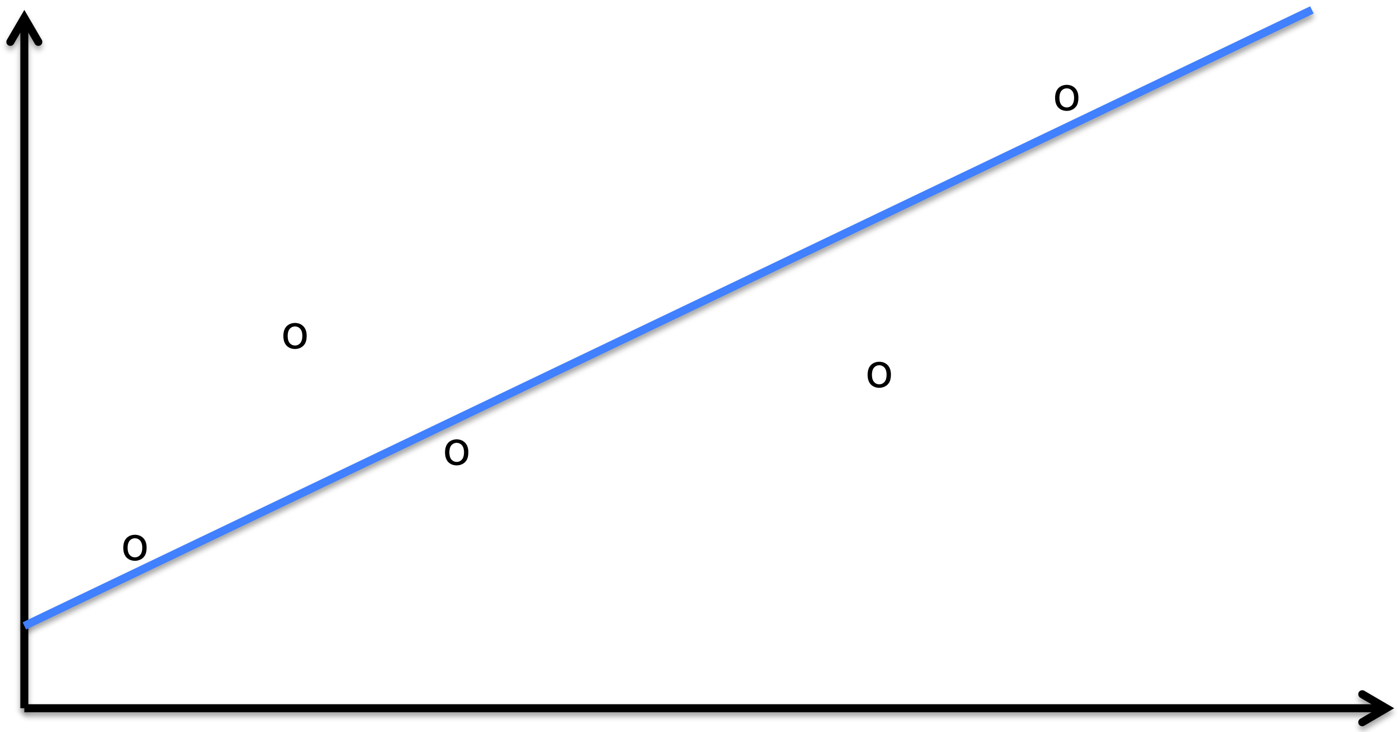
Prof. Julia Hockenmaier
juliahmr@illinois.edu

http://cs.illinois.edu/fa11/cs440

Regression

Linear regression

Given some data $\{(x,y)\ldots\}$, with $x, y \in R$,
find a function $f(x) = w_1x + w_0$ such that $f(x) = y$.



Squared Loss

We want to find a weight vector **w** which minimizes the loss (error) on the training data $\{(x_1,y_1)\ldots(x_N, y_N)\}$

$$L(\mathbf{w}) = \sum_{i=1}^{N} L_2(f_{\mathbf{w}}(x_i), y_i)$$

$$= \sum_{i=1}^{N} (y_i - f_{\mathbf{w}}(x_i))^2$$

Linear regression

We need to minimize the loss on the training data: $\mathbf{w} = \operatorname{argmin}_{\mathbf{w}} \mathrm{Loss}(f_{\mathbf{w}})$

We need to set partial derivatives of Loss($f_{\mathbf{w}}$) with respect to w1, w0 to zero.

This has a closed-form solution for linear regression (see book).

Gradient descent

In general, we won't be able to find a closed-form solution, so we need an iterative (local search) algorithm.

We will start with an initial weight vector **w,** and update each element iteratively in the direction of its gradient:

$$w_i := w_i - \alpha \, d/dw_i \, \mathrm{Loss}(\mathbf{w})$$

Binary classification with Naïve Bayes

For each item $\mathbf{x} = (x_{1 \ldots .} x_d)$, we compute

$$f_k(\mathbf{x}) = P(\mathbf{x} \mid C_k)P(C_k) = P(C_k) \prod_i P(x_i \mid C_k)$$

for both class C_1 and C_2

We assign class C_1 to $\mathbf{x}$ if $f_1(\mathbf{x}) > f_2(\mathbf{x})$

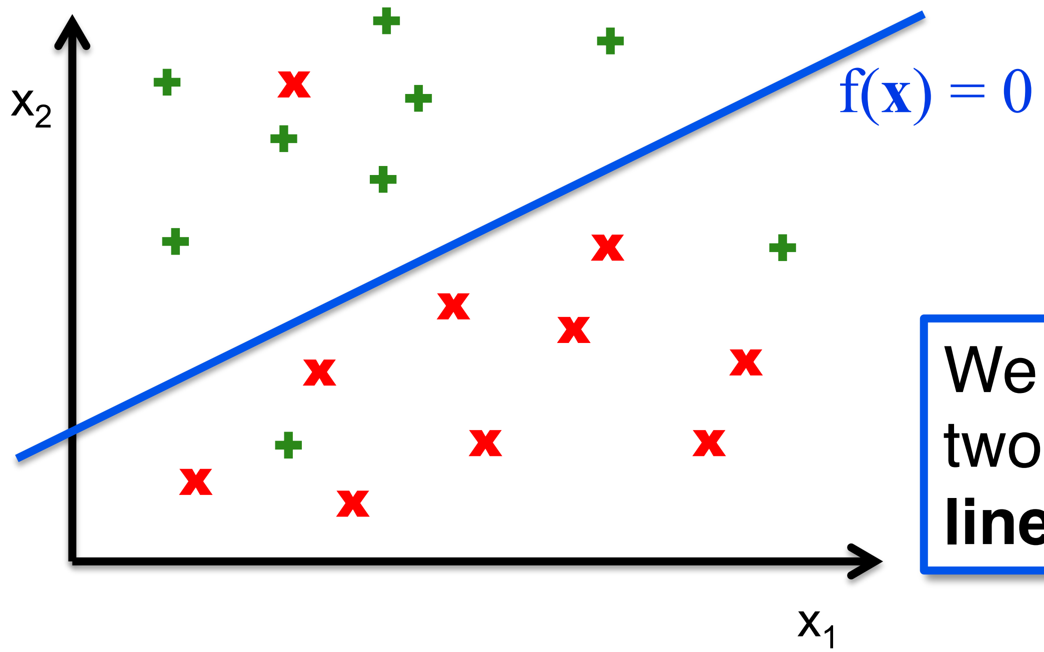
Equivalently, we can define a 'discriminant function'

$f(\mathbf{x}) = f_1(\mathbf{x}) - f_2(\mathbf{x})$

and assign class C_1 to $\mathbf{x}$ if $f(\mathbf{x}) > 0$

Binary classification

The input $\mathbf{x} = (x_{1 \dots} x_d) \in R^d$ is real-valued vector,
We want to learn $f(\mathbf{x})$.



We assume the two classes are **linearly separable**

Binary classification

The input $\mathbf{x} = (x_{1 \ldots .} x_d) \in R^d$ is real-valued vector
We want to learn $f(\mathbf{x})$.

We assume the classes are linearly separable, so we choose a linear discriminant function:

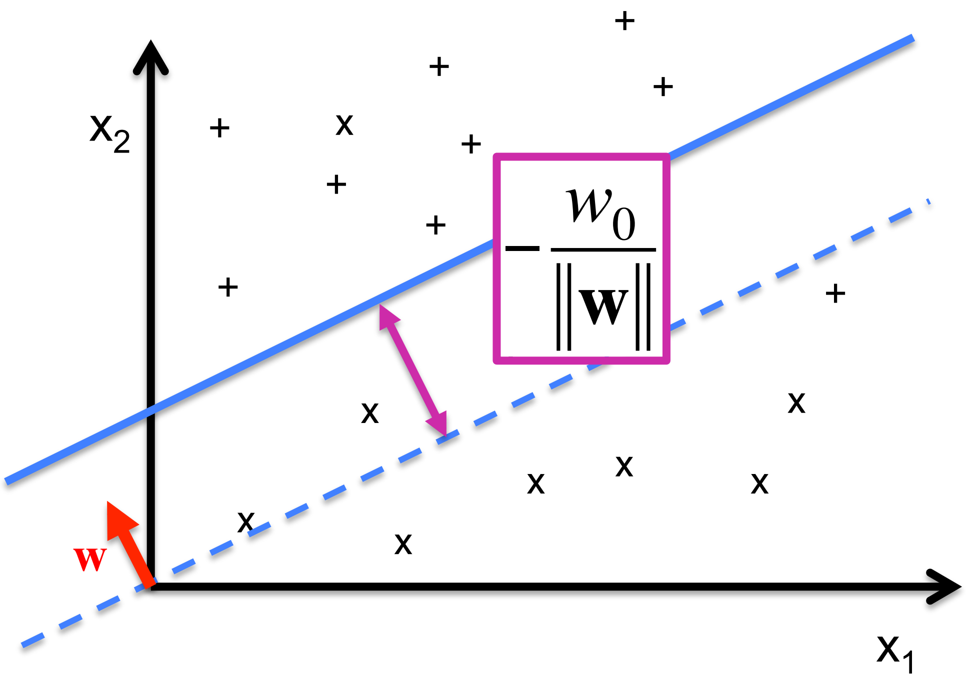
$$f(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x} + w_0$$

- $\mathbf{w} = (w_{1 \ldots .} w_d) \in R^d$ is a weight vector
- w_0 is a bias term
- $-w_0$ is also called a threshold: $-w_0 = \mathbf{w} \cdot \mathbf{x}$

Binary classification

The weight vector **w** defines the orientation of the decision boundary.

The bias term w_0 defines the perpendicular distance of the decision boundary to the origin.



Binary classification

Equivalently, redefine

$\mathbf{x} = (1, x_{1 \dots} x_d) \in R^{d+1}$

$\mathbf{w} = (w_{0,}\ w_{1 \dots} w_d) \in R^{d+1}$

$f(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x}$

Define $C_1 = 1$ $C_2 = 0$

Our classification hypothesis then becomes

$h_{\mathbf{w}}(x) = 1$ if $f(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x} \geq 0$

0 otherwise

Binary classification

Our classification hypothesis then becomes

$h_{\mathbf{w}}(\mathbf{x}) = 1$ if $f(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x} \geq 0$

0 otherwise

We can also think of $h_{\mathbf{w}}(x)$ as a threshold function.

$h_{\mathbf{w}}(\mathbf{x}) = \text{Threshold}(\mathbf{w} \cdot \mathbf{x})$,

where $\text{Threshold}(\mathbf{z}) = 1$ if $\mathbf{z} \geq 0$

0 otherwise

Learning the weights

We need to choose **w** to minimize classification loss.

But we cannot compute this in closed form, because the gradient of **w** is either 0 or undefined.

Iterative solution:

- Start with initial weight vector **w.**
- For each example (**x,**y) update weights **w** until all items are correctly classified.

Observations

If we classify an item $(\mathbf{x}, y)$ correctly,
we don't need to change $\mathbf{w}$.

If we classify an item $(\mathbf{x}, y)$ incorrectly,
there are two cases:

- $\mathbf{y} = 1$ (above the true decision boundary)
  $h_{\mathbf{w}}(\mathbf{x}) = 0$ (below the true decision boundary)
  We need to move our decision boundary up!

- $\mathbf{y} = 0$ (below the true decision boundary)
  $h_{\mathbf{w}}(\mathbf{x}) = 1$ (above the true decision boundary)
  We need to move our decision boundary down!

Learning the weights

Evaluating $y - h_{\mathbf{w}}(\mathbf{x})$ will tell us what to do:

– $h_{\mathbf{w}}(\mathbf{x})$ is correct: $y - h_{\mathbf{w}}(\mathbf{x}) = 0$ (stay!)

– If $y = 1$, but we predict $h_{\mathbf{w}}(\mathbf{x}) = 0$
$y - h_{\mathbf{w}}(\mathbf{x}) = 1 - 0 = 1$
(move up!)

– If $y = 0$, but we predict $h_{\mathbf{w}}(\mathbf{x}) = 1$
$y - h_{\mathbf{w}}(\mathbf{x}) = 0 - 1 = -1$
(move down!)

Learning the weights (initial attempt)

Iterative solution:

- Start with initial weight vector **w.**
- For each example (**x,**y) update weights **w** until all items are correctly classified.

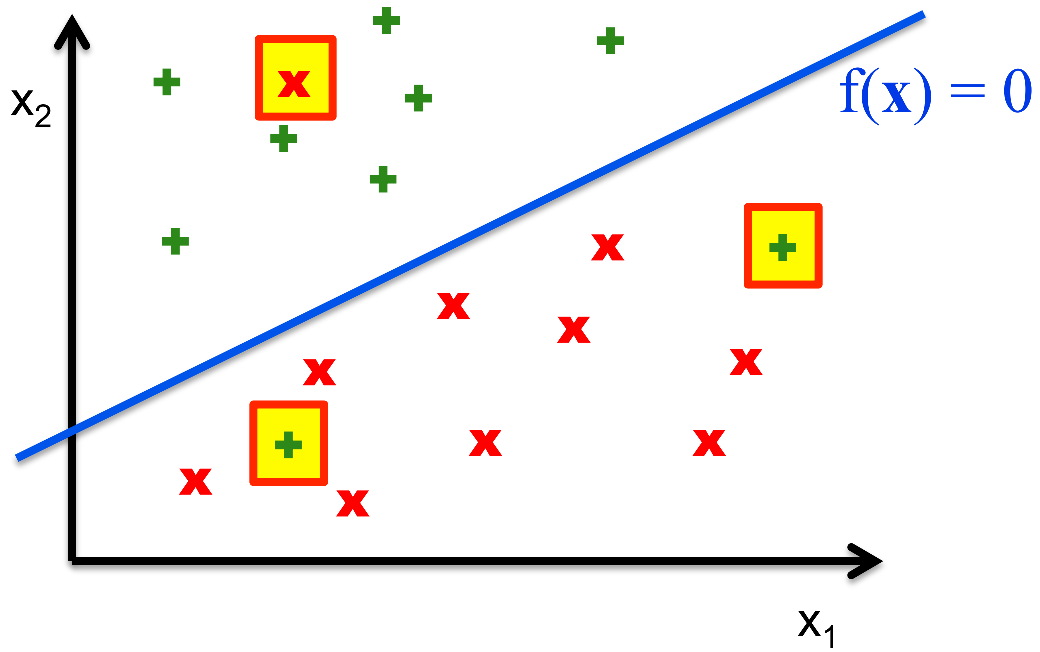
Update rule:

For each example $(\mathbf{x}, y)$ update each weight w_i:

$$w_i := w_i + (y - h_{\mathbf{w}}(\mathbf{x}))x_i$$

There is a problem:

Real data is not perfectly separable.
There will be noise, and our features may not be sufficient.



Learning the weights

Observation:
When we've only seen a few examples, we want the weights to change a lot.

After we've seen a lot of examples, we want the weights to change less and less, because we can now classify most examples correctly.

Solution: We need a learning rate which decays over time.

Learning the weights (Perceptron algorithm)

Iterative solution:

- Start with initial weight vector **w.**
- For each example (**x**,y) update weights **w** until **w** has converged (does not change significantly anymore)

Perceptron update rule ('online'):

- For each example $(\mathbf{x},y)$ update each weight w_i:
  $w_i := w_i + \alpha\,(y - h_{\mathbf{w}}(\mathbf{x}))x_i$
- α decays over time t (t=#examples) e.g $\alpha = n/(n+t)$

Batch/Epoch Perceptron Learning

Choose a convergence criterion (#epochs, min $|\Delta \mathbf{w}|$, …)

Choose a learning rate α, an initial $\mathbf{w}$

Repeat until convergence:

$\Delta \mathbf{w} = \Sigma_{\mathbf{x}} \, \alpha \text{ err } \mathbf{x}$ (sum over training set holding $\mathbf{w}$)

$\mathbf{w} \leftarrow \mathbf{w} + \Delta \mathbf{w}$ (update with accumulated changes)

Now it always converges, regardless of α (will influence the rate), and whether or not training points are linearly