Lecture 3
Linear Classification Models
Perceptron

Classification problem

Let’s look at the problem of spam filtering

Now anyone can learn how to earn $200 - $943 per day or More ! If you can type (hunt and peck is ok to start) and fill in forms, you can score big! So don't delay waiting around for the next opportunity...it is knocking now! Start here: http://redbluecruise.com/t/c/381/polohoo/yz37957.html

Do you Have Poetry that you think should be worth $10,000.00 USD, we do!.. Enter our International Open contest and see if you have what it takes. To see details or to enter your own poem, Click link below. http://e-suscriber.com/imys?e=0sAoo4q9s4zYYUoYQ&m=795314&l=0

View my photos!
I invite you to view the following photo album(s): zak-month27

Hey have you seen my new pics yet???? Me and my girlfreind would love it if you would come chat with us for a bit.. Well join us if you interested. Join live web cam chat here: http://e-commcentral.com/imys?e=0sAoo4q9s4zYYUoYQ&m=825314&l=0

Let’s look at the design choices

- Training data?
  - Past emails and whether they are considered spam or not (you can also choose to use non-spam or spam emails only, but that will require different choices later on)
- Target function?
  - Email -> spam or not
- Representation of the function?
  - ?
- Learning algorithm
  - ?

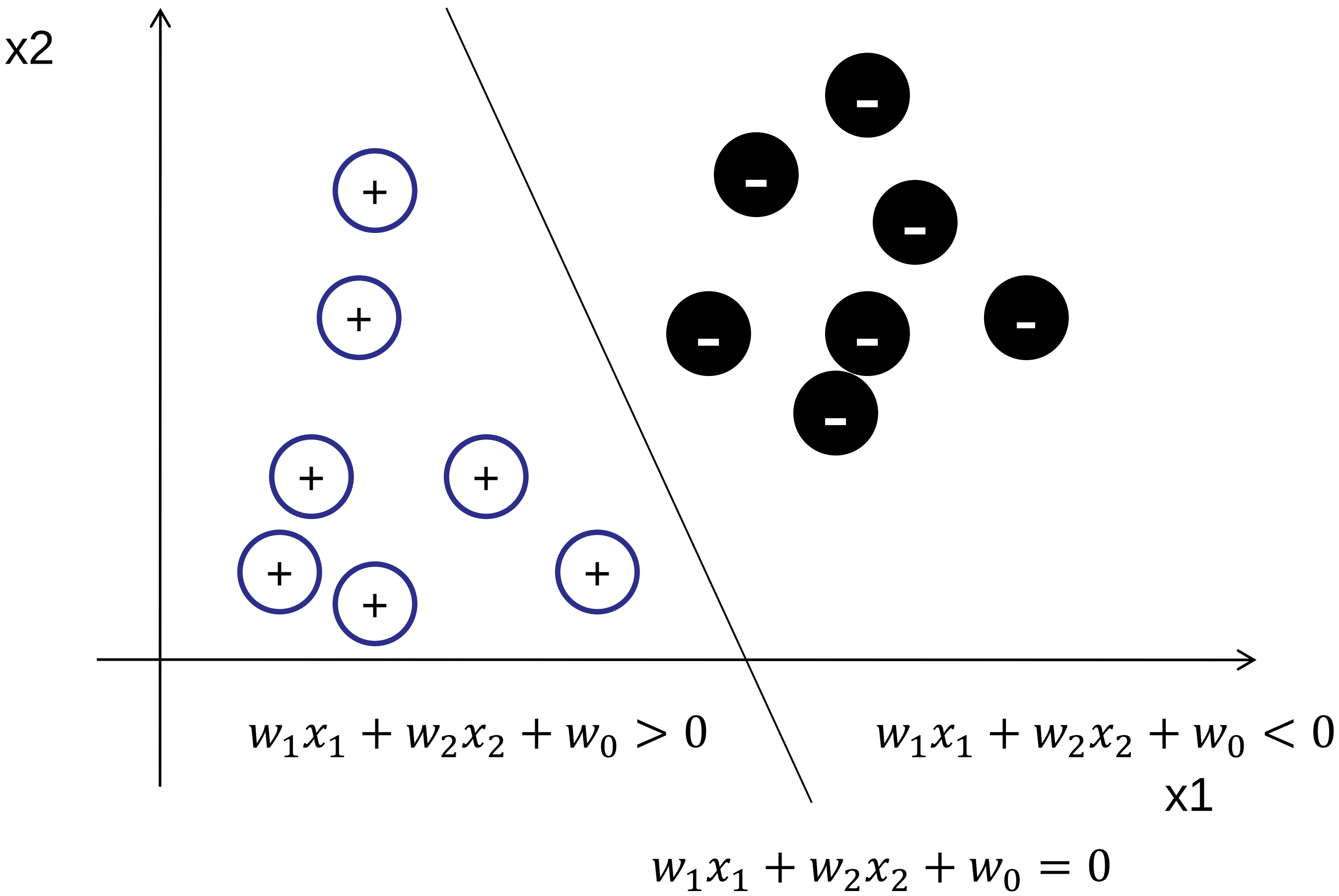
We will focus a lot on these two aspects in this class.

Continue with the design choices

- Representation of the function (email -> spam or not) ?
- First of all, how to represent an email?
  - Use ***bag-of-words*** to represent an email
  - Consider a fixed dictionary, it turns an email into a collection of features, e.g., where each feature describe whether a particular word in the dictionary is present in the email (alternatively, the feature could be the count or normalized count of the words)
- This gives us a standard supervised classification problem typically seen in text books and papers
  - **Training set:** a set of examples (instances, objects) with ***class labels***, e.g., positive (spam) and negative (non spam)
  - **Input representation:** an example is described by a set of attributes/features (eg. one feature could be whether “$” is present, etc.)
  - **Goal:** Given an unseen email, and its input representation, predict its label
- Next question: what function forms to use?

Linear Classifier

- We will be begin with the simplest choice: linear classifiers



Why linear model?

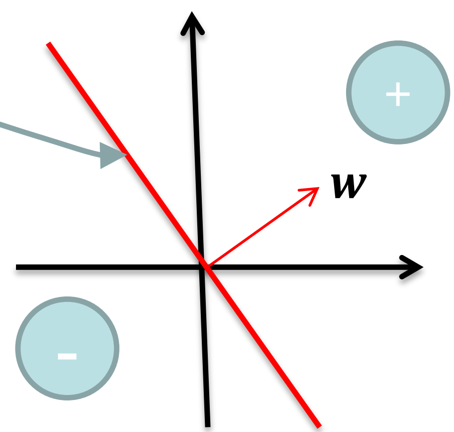
- Simplest model – fewer parameters to learn (requires less training data to learn reliably)

- Intuitively appealing -- draw a straight line (for 2-d inputs) or a linear hyper-plane (for higher dimensional inputs) to separate positive from negative

- Can be used to learn nonlinear models as well. How?
  - Introducing nonlinear features (e.g., $x_1^2, x_2^2, x_1 x_2 \ldots$)
  - Use kernel tricks (we will talk about this later this term)

Learning Goal

- Given a set training examples, each is described by m features $x_1, \ldots, x_m$, and belong to either the positive or negative class, $y \in \{+1, -1\}$
- Let $\boldsymbol{x} = [1, x_1, \ldots, x_m]^T$,
- $\boldsymbol{w} = [w_0, w_1, \ldots\ w_m]^T$ defines a decision boundary $\boldsymbol{w}^T \boldsymbol{x} = 0$ that the input space into two parts



- The vector $\boldsymbol{w}$ is the normal vector of the decision boundary, i.e., its perpendicular to the boundary
- $\boldsymbol{w}$ points to the positive side
- Goal: find a $\boldsymbol{w}$ s.t. the decision boundary separates positive examples from negative examples

How to learn: the perceptron algorithm

How can we achieve this? Perceptron is one approach.

1. It starts with some (random) vector ***w*** and incrementally updates ***w*** whenever it makes a mistake.
2. Let $\boldsymbol{w_t}$ be the current weight vector, and suppose it makes a mistake on example $(\boldsymbol{x}, y)$, that is to say $y\boldsymbol{x}^{\boldsymbol{T}}\boldsymbol{w_t}$<0.
3. The update intends to correct for the mistake

The Perceptron Algorithm

Let $\mathbf{w} \leftarrow (0,0,0,...,0)$ *//Start with 0 weights*

Repeat *//go through training examples one by one*

Accept training example $i : (\mathbf{x}_i \ , y_i)$

$u_i \leftarrow \mathbf{x}_i^T \mathbf{w}$ *//Apply the current weight*

if $y_i \ u_i \ <= \ 0$ *// If it is misclassified*

$\mathbf{w} \leftarrow \mathbf{w} + y_i \mathbf{x}_i$ *// update **w***

Important notes:

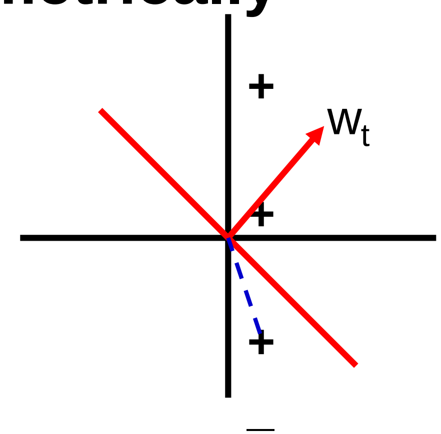
- Correcting for a mistake could move the decision boundary so much that previously correct examples are now misclassified.
- As such it must go over the training examples multiple times
- Each time it goes through the whole training set, it is called an epoch.
- It will terminate if no update is made to ***w*** during one epoch – which means it has converged.

Effect of Perceptron Updating Rule

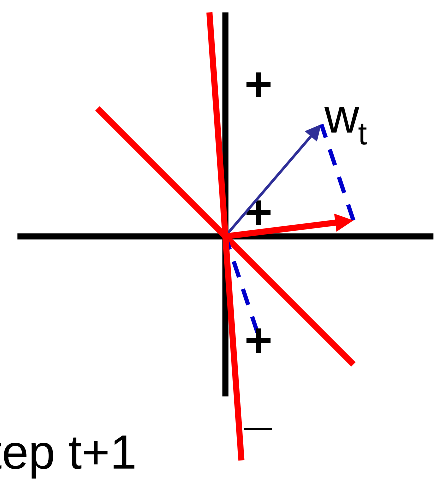
- **Mathematically speaking**
  - Let $(\boldsymbol{x}, y)$ be the mistake example, i.e., $y \cdot \boldsymbol{x}^{\boldsymbol{T}} \boldsymbol{w}_t < \boldsymbol{0}$
  - Now with the updated $\boldsymbol{w}_{t+1} = \boldsymbol{w}_{\boldsymbol{t}} + y \cdot \boldsymbol{x}$

  $$y\boldsymbol{x}^{\boldsymbol{T}}\boldsymbol{w}_{\boldsymbol{t+1}} = y\boldsymbol{x}^{\boldsymbol{T}}(\boldsymbol{w}_{\boldsymbol{t}} + y\boldsymbol{x}) = y\boldsymbol{x}^{\boldsymbol{T}}\boldsymbol{w}_{\boldsymbol{t}} + y^2||\boldsymbol{x}||^2 > y\boldsymbol{x}^{\boldsymbol{T}}\boldsymbol{w}_{\boldsymbol{t}}$$

  The updating rule makes $y\boldsymbol{x}^{\boldsymbol{T}}\boldsymbol{w}_{t+1}$ more positive, thus can potentially correct for the mistake
- **Geometrically**



Step t



Step t+1

Online vs Batch

- We call the above perceptron algorithm an ***online algorithm***
- Online algorithms perform learning each time it receives an training example
- In contrast, ***batch learning*** algorithms collect a batch of training examples and learn from them all at once.

Batch Perceptron Algorithm

$$
\begin{array}{l}
\text{Given : training examples } (\mathbf{x}_i\ , y_i),\ i = 1,\ldots, N \\
\text{Let } \mathbf{w} \leftarrow (0,0,0,\ldots,0) \\
\text{do} \\
\qquad \mathit{delta} \leftarrow (0,0,0,\ldots,0) \\
\qquad \text{for } i\ =\ 1 \text{ to } N \text{ do} \\
\qquad\qquad u_i \leftarrow \mathbf{w}^T \mathbf{x}_i \\
\qquad\qquad \text{if } \ y_i \cdot u_i \ <= \ 0 \\
\qquad\qquad\qquad \mathit{delta} \leftarrow \mathit{delta} - y_i \mathbf{x}_i \\
\qquad \mathit{delta} \leftarrow \mathit{delta} / N \\
\qquad \mathbf{w} \leftarrow \mathbf{w} - \eta\, \mathit{delta} \\
\text{until } \ |\mathit{delta}| < \varepsilon
\end{array}
$$

- Perceptron does gradient descent to minimize loss function $E(\mathbf{w}) = \frac{1}{N}\sum_{i=1}^{N}(-y_i\mathbf{w}^T\mathbf{x}_i)_+$
- $delta$ stores the gradient
- η is the learning rate of the gradient descent steps
- Too large η causes oscillation, too small leads to slow convergence
- Common to use large η first, then gradually reduce it

Good news

- C**onvergence Property:**

  For linearly separable data (i.e., there exists an linear decision boundary that perfectly separates positive and negative training examples), the perceptron algorithm converges in a **finite number of steps.**
- *Why?* If you are mathematically curious, read the following slide, you will find the answer.
- *And how many steps?* If you are practically curious, read the following slide, answer is in there too.
- **The further good news** is that you are not required to master this material, they are just provided for the curious ones

Proof

To show convergence, we just need to show that each update moves the weight vector closer to a solution vector by a lower bounded amount

Let w* be a solution vector, and w_t be our w at tth step,

$$\cos ine(w^*, w_t) = \frac{w^* \cdot w_t}{\|w^*\| \cdot \|w_t\|}$$

$$w^* \cdot w_t = w^* \cdot (w_{t-1} + y^t x^t) = w^* \cdot w_{t-1} + w^* y^t x^t$$

Assume that w^* classify all examples with a margin γ, i.e., $w^* yx > \gamma$ for all examples

$$w^* \cdot w_t = w^* \cdot w_{t-1} + w^* y^t x^t > w^* \cdot w_{t-1} + \gamma > w^* \cdot w_{t-2} + 2\gamma > \ldots > w^* w_0 + t\gamma = t\gamma$$

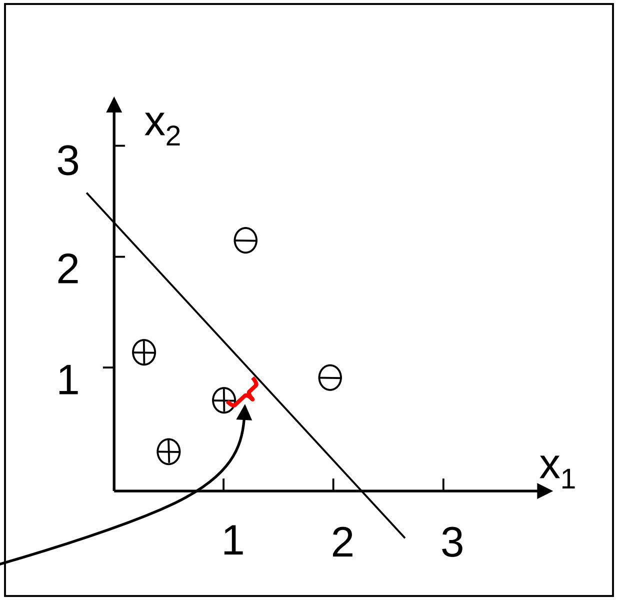
$$\|w_t\|^2 = \|w_{t-1} + y^t x^t\|^2 = \|w_{t-1}\|^2 + {y^t}^2 \|x^t\|^2 + 2 w_{t-1} y^t x^t < \|w_{t-1}\|^2 + \|x^t\|^2$$

Assume that $\|\mathbf{x}\|$ are bounded by D

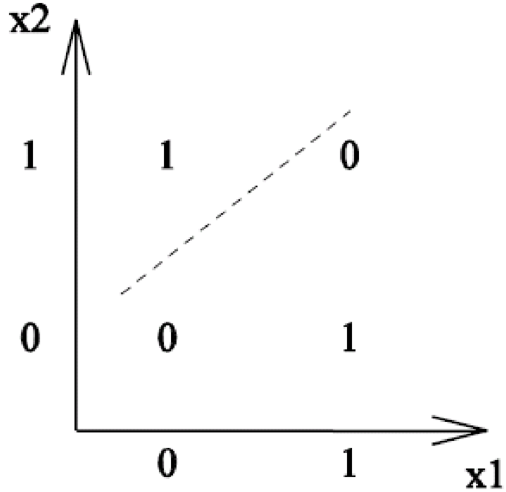
$$\|w_t\|^2 < \|w_{t-1}\|^2 + \|x^t\|^2 < \|w_{t-1}\|^2 + D^2 < \|w_{t-2}\|^2 + 2D^2 < \ldots < tD^2$$

$$\cos ine(w^*, w_t) = \frac{w^* \cdot w_t}{\|w^*\| \cdot \|w_t\|} > \frac{t\gamma}{\|w^*\| \cdot \|w_t\|} > \frac{t\gamma}{\|w^*\| \cdot \sqrt{tD^2}}$$

$$\frac{t\gamma}{\|w^*\| \cdot \sqrt{tD^2}} < 1 \Rightarrow \sqrt{t} < \frac{D\|w^*\|}{\gamma} \Rightarrow t < D^2 \Big/ \frac{\gamma^2}{\|w^*\|^2}$$



Bad news



What about non-linearly separable cases!

In such cases the algorithm will never stop! How to fix?

One possible solution: look for decision boundary that make as few mistakes as possible – NP-hard (refresh your 325 memory!)

Fixing the Perceptron

Idea one: only go through the data once, or a fixed number of times

$$
\begin{array}{l}
\text{Let } \mathbf{w} \leftarrow (0,0,0,\ldots,0) \\
\text{for i} = 1,\ldots,\text{T} \\
\qquad \text{Take training example } i: (\mathbf{x}_i\ , y_i) \\
\qquad u_i \leftarrow \mathbf{x}_i^T \mathbf{w} \\
\qquad \text{if }\ y_i\ u_i\ \ <=\ \ 0 \\
\qquad\qquad \mathbf{w} \leftarrow \mathbf{w} + y_i \mathbf{x}_i
\end{array}
$$

At least this stops!

Problem: the final w might not be good

e.g., right before terminating, the algorithm might perform an update on a total outlier…

Voted-Perceptron

Idea two: keep around intermediate hypotheses, and have them "vote" [Freund and Schapire, 1998]

$$
\begin{aligned}
&\text{Let } w_0 = (0,0,0,\ldots,0)\\
&c_0 = 0, n = 0\\
&\text{repeat for a fixed number of steps}\\
&\qquad \text{Take example } i : (\mathbf{x}_i, y_i)\\
&\qquad u_i \leftarrow \mathbf{x}_i^T \mathbf{w}_n\\
&\qquad \text{if } y_i u_i <= 0\\
&\qquad\qquad \mathbf{w}_{n+1} \leftarrow \mathbf{w}_n + y_i \mathbf{x}_i\\
&\qquad\qquad c_{n+1} = 0\\
&\qquad\qquad n = n+1\\
&\qquad \text{else}\\
&\qquad\qquad c_n = c_n + 1
\end{aligned}
$$

Store a collection of linear separators w_0, w_1,…, along with their survival time c_0, c_1, …

The c's can be good measures of reliability of the w's.

For classification, take a weighted vote among all *N* separators:

$$\operatorname{sgn}\left\{\sum_{n=0}^{N} c_n \operatorname{sgn}(\mathbf{x}^T \mathbf{w}_n)\right\}$$

Summary

- Perceptron incrementally learns a linear decision boundary to separate positive from negative
- It begins with a random weight vector or a zero weight vector, and incrementally update the weight vector whenever it makes a mistake
- Each mistaken example $(\boldsymbol{x}, y)$ contributes an addition $y\boldsymbol{x}$ (online) or $\frac{1}{n}y\boldsymbol{x}$ (batch) to the current weight vector
- For online perceptron, different orderings of the training examples can lead to different outputs
- Voted perceptron can handle non-linearly separable data, and is more robust to noise/outlier