CPSC 4420/6420

Artificial Intelligence

24 – Vectorization, Binary Perceptron

November 12, 2020

Announcements

- Project 4 is due on 11/12
- Project 5 will be assigned by tomorrow
  - The deadline is on 11/25
- Quiz 8 will be assigned later this evening
  - The deadline is on 11/19

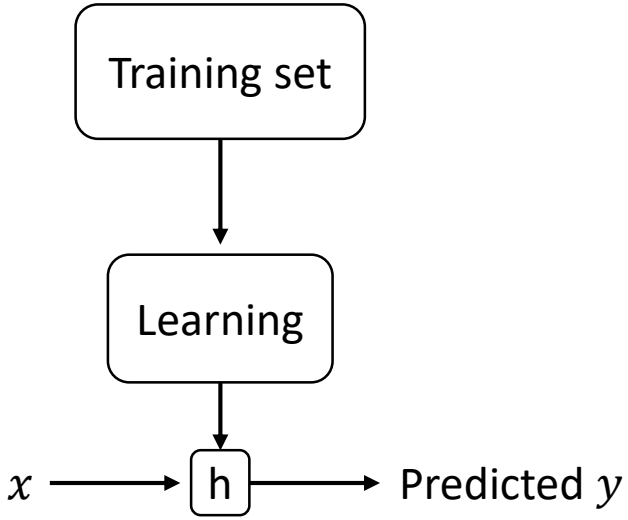
Lecture 24

Slide Credits:
Pieter Abbeel
Dan Klein
Ioannis Karamouzas

Quick Recap

Supervised learning

- Let x denote the "input" variables/features, and y the "output" or target variable
- Let $\mathcal{X}$ denote the space of input values, and $\mathcal{Y}$ the space of output values
- Supervised learning
  - We are given a training set, consisting of a list of m training examples $\{(x^{(i)}, y^{(i)}), i = 1 \ldots m\}$
  - Our goal is to learn a function $h : \mathcal{X} \mapsto \mathcal{Y}$, so that $y^{(i)} \approx h(x^{(i)})$ for each training example

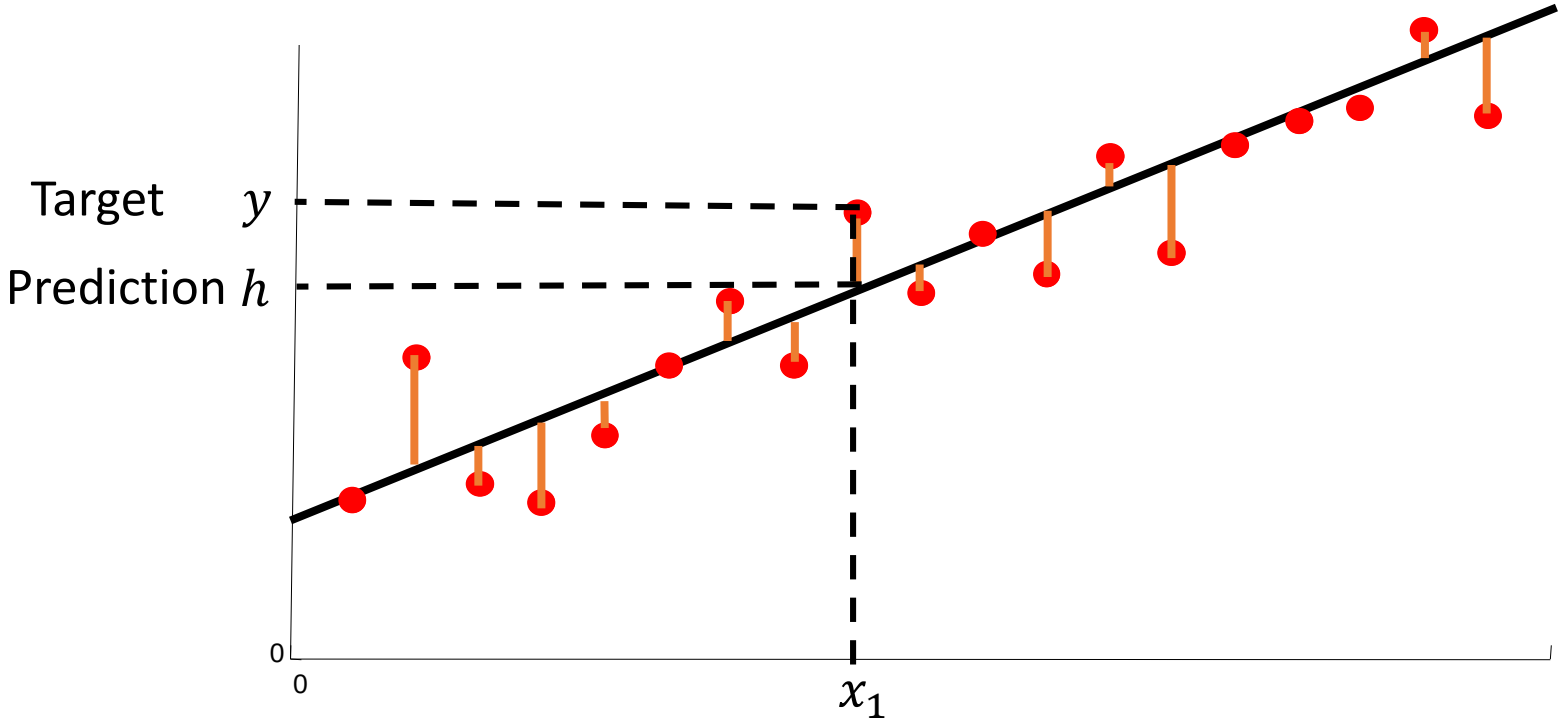


Least squares cost function

- Search for w that minimizes the following cost/loss/objective function:

$$J(w) = \frac{1}{2}\sum_{i=1}^{m}(h_w(x^{(i)}) - y^{(i)})^2$$

- This is known as least squares cost function

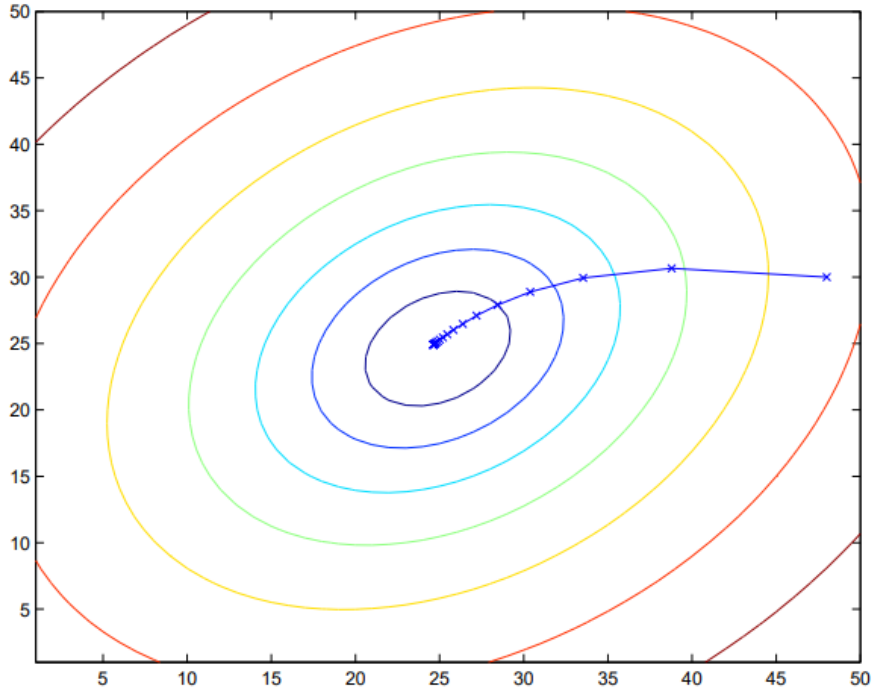


Gradient Descent

- Let $f(w_1, \ldots, w_n)$ be a multivariate objective function
- To find a local minimum

  - `init` w
  - `for iter = 1,2,…`

    $w \leftarrow w - \alpha * \nabla f(w)$



Batch gradient descent on least squares objective

$$\min_w J(w) = \min_w \frac{1}{2}\sum_{i=1}^{m}(h_w(x^{(i)}) - y^{(i)})^2$$

- `init` w
- `for iter = 1,2,…`

$$w \leftarrow w - \alpha * \frac{1}{2}\sum_{i=1}^{m}\nabla(h_w(x^{(i)}) - y^{(i)})^2$$

Stochastic gradient descent on least squares objective

$$\min_w J(w) = \min_w \frac{1}{2}\sum_{i=1}^{m}(h_w(x^{(i)}) - y^{(i)})^2$$

Idea: Once gradient on one training example has been computed, might as well incorporate before computing next one

- `init` w
- `for iter = 1,2,…`
  - `pick random k`

$$w \leftarrow w - \alpha * \frac{1}{2}\nabla(h_w(x^{(k)}) - y^{(k)})^2$$

Mini-batch gradient descent on least squares objective

$$\min_w J(w) = \min_w \frac{1}{2}\sum_{i=1}^{m}(h_w(x^{(i)}) - y^{(i)})^2$$

Idea: Gradient over small set of training examples (=mini-batch) can be computed in parallel, might as well do that instead of a single one

- `init` w
- `for iter = 1,2,…`
  - `pick random subset of training examples B`

$$w \leftarrow w - \alpha * \frac{1}{2}\sum_{b\in \mathbf{B}} \nabla(h_w(x^{(b)}) - y^{(b)})^2$$

Linear regression update rules

- Batch gradient descent (for every *j*)

$$w_j \leftarrow w_j - \alpha \sum_{i=1}^{m} (h_w(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

- Stochastic gradient descent (for every *j*)

$$w_j \leftarrow w_j - \alpha \, (h_w(x^{(k)}) - y^{(k)}) x_j^{(k)}$$

Vectorization

Batch gradient descent

- `init` $w_1, w_2, \ldots, w_n$
- `for iter = 1,2,…`
  - `for j = 1,2,…,n`

$$w_j = w_j - \alpha \sum_{i=1}^{m} (h_w(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

Linear regression update rules

- init $w_1, w_2, \ldots, w_n$
- for iter = 1, 2, …
  - for j = 1, 2, …, n

$$w_j = w_j - \alpha \sum_{i=1}^{m} (w \cdot x^{(i)} - y^{(i)}) x_j^{(i)}$$

Vectorization

- We are given m training examples $\{(x^{(i)}, y^{(i)}), i = 1 \ldots m\}$
  - Let $x^{(i)}$ consists of (n+1) input features stacked into a column vector
  - Let $W = \begin{bmatrix} w_0 \\ w_1 \\ \cdots \\ w_n \end{bmatrix}$ denote the features' weights
- We can form a matrix X by concatenating the $x^{(i)}$'s to be the columns of X

$$X = \begin{bmatrix} | & | & | & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & | & | \end{bmatrix}$$

- Similarly, we can append all $y^{(i)}$'s into a column vector $Y = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \cdots \\ y^{(m)} \end{bmatrix}$

Vectorization

- For every j we need to compute $\frac{\partial J(w)}{\partial w_j} = \sum_i^m (h_w(x^{(i)}) - y^{(i)}) x_j^{(i)}$
  - We can compute h_w for all $x^{(i)}$ at once as $h = X^T W$
  - The partial derivative is the j[th] element of $\nabla_w J = X(h - Y)$
- Gradient update

$$W = W - \alpha X(h - Y)$$

- In Python, you can use *numpy* for efficiently implementing gradient descent
  - np.array for storing vectors and matrices
  - numpy.dot and/or numpy.multmat for computing matrix-vector products

`

Movie Name	Score	Profit (in $IM)
AI	1	1.0
Endless Search	2	2.0
Value in Iteration	2	3.0
Back to A*	3	4.5
Learn to Q	4	6.0

Movie Name	Score	Profit (in $IM)
AI	1	1.0
Endless Search	2	2.0
Value in Iteration	2	3.0
Back to A*	3	4.5
Learn to Q	4	6.0

Classification

Example: Digit classification

- Input: images / pixel grids
  - Each input maps to a feature vector
  - E.g. one feature (variable) for each grid position based on pixel intensity

     $\rightarrow \langle 0, 0, 1, 1, 0 \ldots 0 \rangle$
- Output: a digit 0-9
- Setup
  - Get a large collection of example images, each labeled with a digit
  - Note: someone has to hand label all this data!
  - Want to learn to predict labels of new, future digit images

0

1

2

1

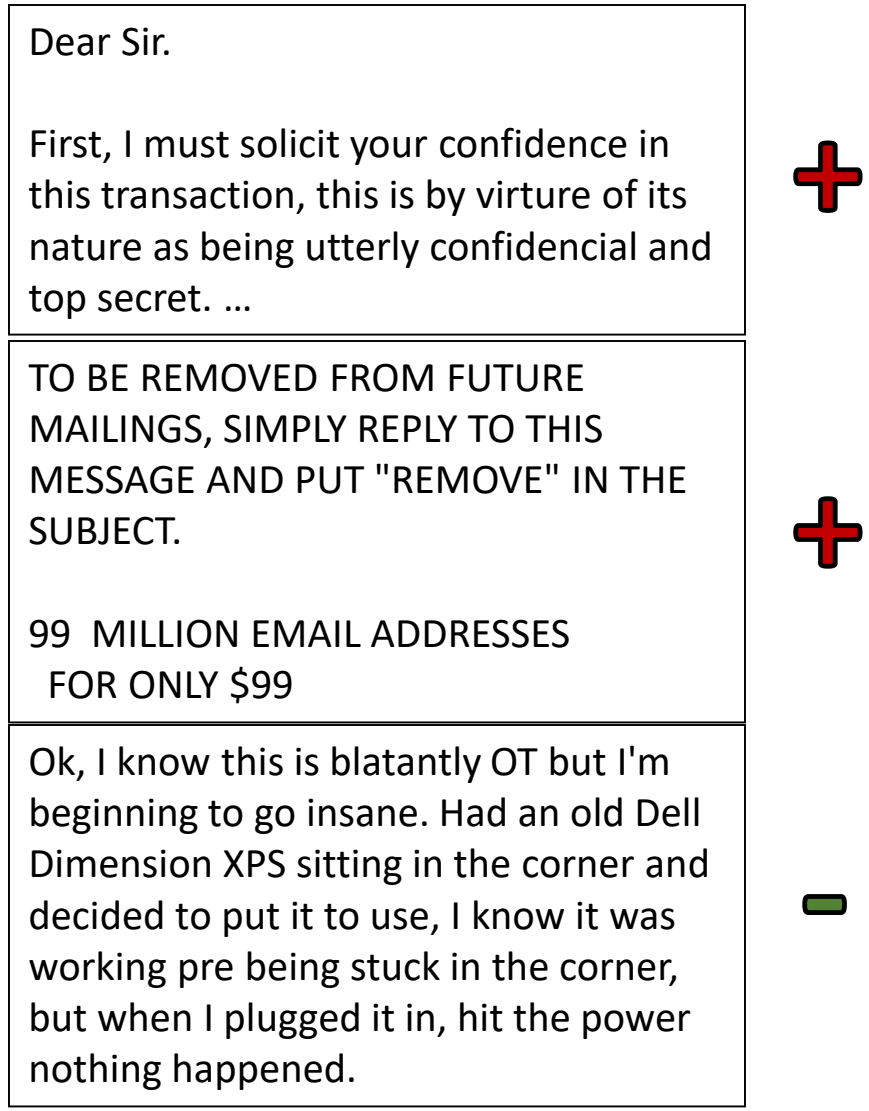
Classification tasks

- Classification: given inputs x, predict labels (classes) y
- Examples:
  - Medical diagnosis (input: symptoms, classes: diseases)
  - Fraud detection (input: account activity, classes: fraud / no fraud)
  - Automatic essay grading (input: document, classes: grades)
  - Object detection (input: images, classes: animals)
  - Customer service email routing
  - … many more

Binary classification example: Spam filter

- Input: an email
- Output: spam/ham (+1/-1 or 1/0)

- Setup
  - Collect example emails, each labeled “spam” or “ham”
  - Note: someone has to hand label all this data!
  - Want to learn to predict labels of new, future emails



Feature vectors

x y

```
Hello,

Do you want free printr
cartriges?  Why pay more
when you can get them
ABSOLUTELY FREE!  Just
```

```
# free       : 2
YOUR_NAME    : 0
MISSPELLED   : 2
FROM_FRIEND  : 0
...
```

SPAM
or
+

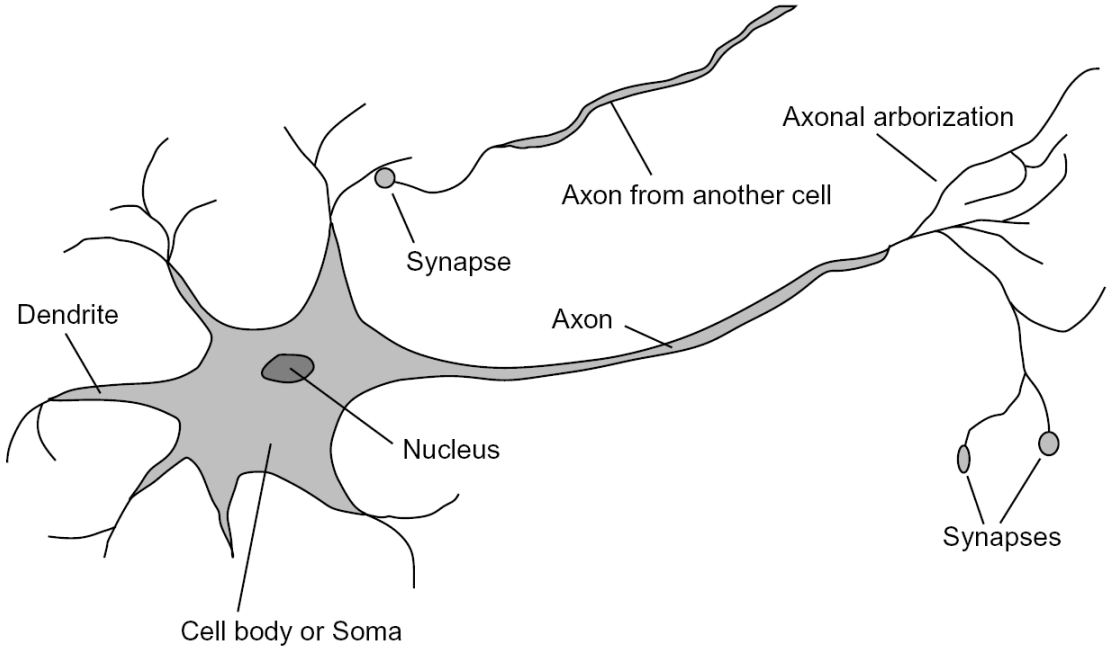
```
PIXEL-7,12   : 1
PIXEL-7,13   : 0
...
NUM_LOOPS    : 1
...
```

“1”
or
+

Linear Classifier

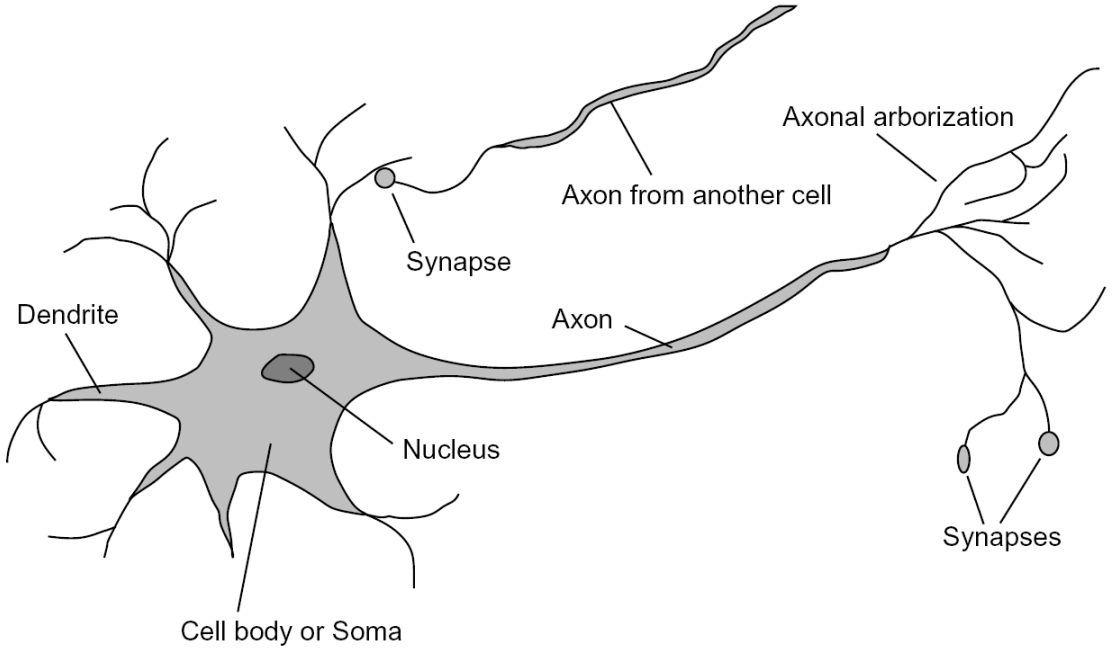
Perceptron learning

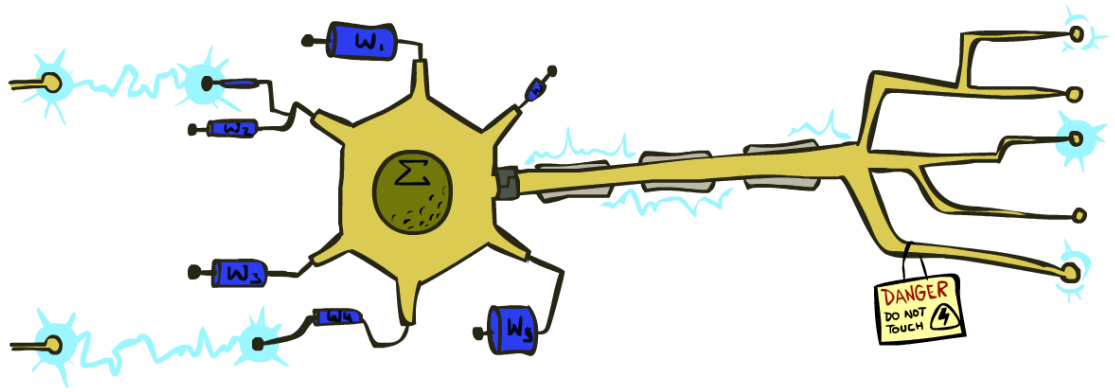
- Very loose inspiration: human neurons



Perceptron learning

- Very loose inspiration: human neurons



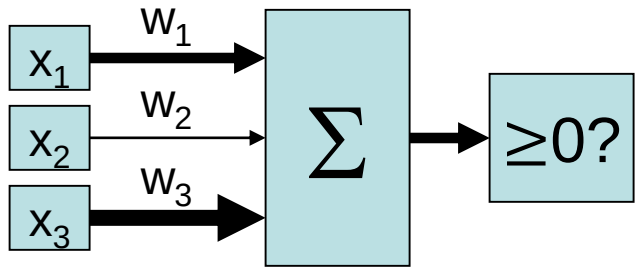


Linear classifiers

- Inputs are feature values
- Each feature has a weight
- Sum is the activation

$$\text{activation}_w(x) = \sum_i w_i x_i = w \cdot x$$

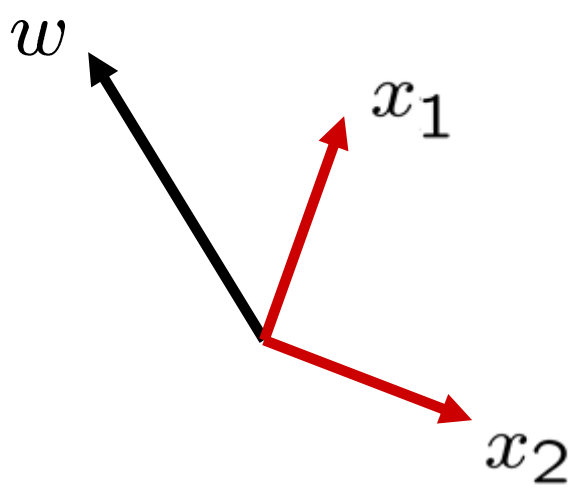
- If the activation is:
  - Positive, output is 1
  - Negative, output is 0



Weights

- Binary case: compare features to a weight vector

```
# free       : 4
YOUR_NAME    :-1
MISSPELLED   : 1
FROM_FRIEND  :-3
...
```




```
# free       : 2
YOUR_NAME    : 0
MISSPELLED   : 2
FROM_FRIEND  : 0
...
```

Dot product $w \cdot x$ positive means the positive class

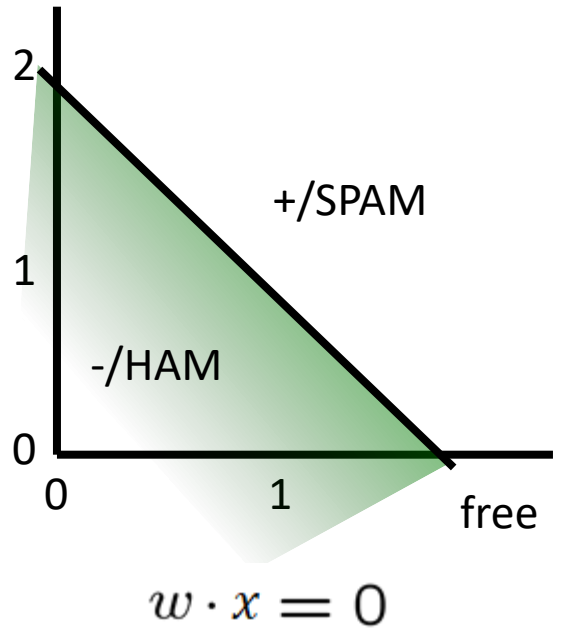
```
# free       : 0
YOUR_NAME    : 1
MISSPELLED   : 1
FROM_FRIEND  : 1
...
```

Binary decision rule

- Learning: figure out the weight vector from examples
- In the space of feature vectors
  - Examples are points
  - Any weight vector defines a hyperplane
  - One side corresponds to positive class (Y=1)
  - Other corresponds to negative class (Y=0)

w

```
BIAS   : 0
free   : 1
money  : 1
...
```



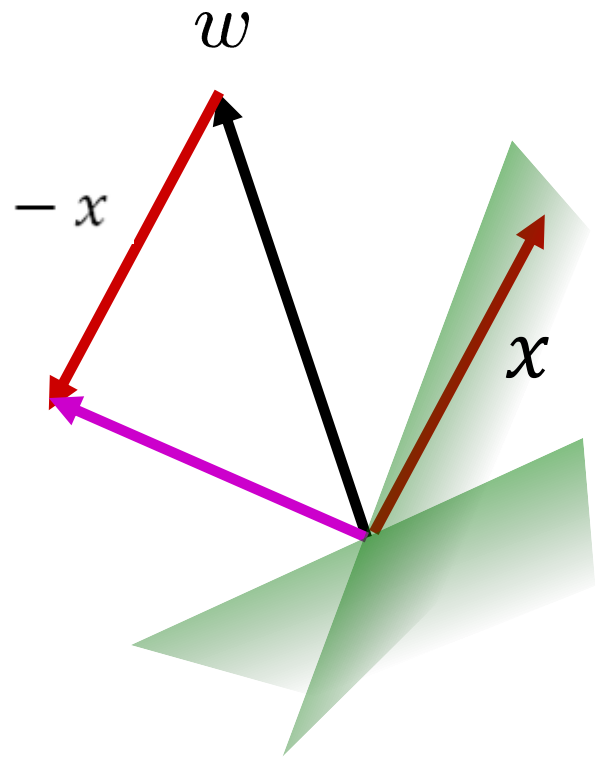
Binary perceptron algorithm

- Start with $w = 0$
- For each training instance:
  - Classify with current weights

$$\hat{y} = \begin{cases} 1 & \text{if } w \cdot x \geq 0 \\ 0 & \text{if } w \cdot x < 0 \end{cases}$$

  - If correct (i.e., $\hat{y}$=y), no change!
  - If wrong: adjust the weight vector by adding or subtracting the feature vector. Subtract if $\hat{y}$ is 0

$$w = w + (y - \hat{y}) \cdot x$$



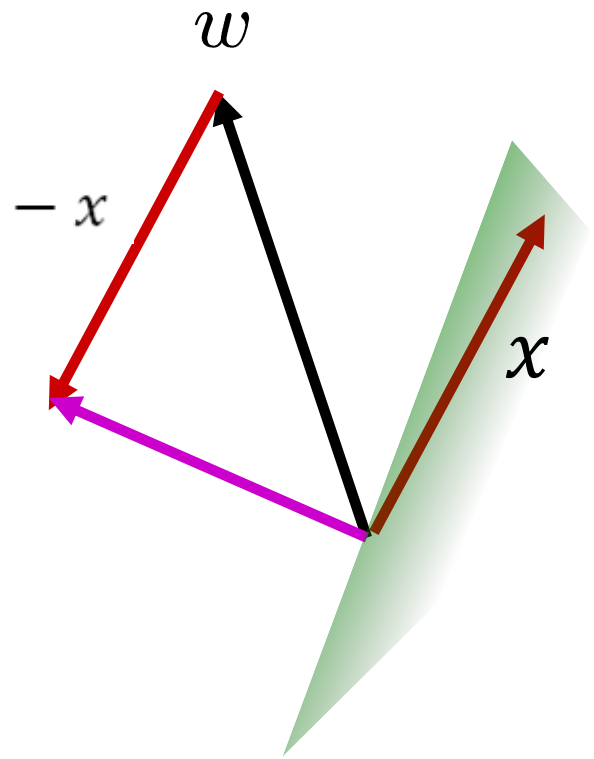
Binary perceptron algorithm

- Start with $w = 0$
- For each training instance:
  - Classify with current weights

$$\hat{y} = \begin{cases} 1 & \text{if } w \cdot x \geq 0 \\ 0 & \text{if } w \cdot x < 0 \end{cases}$$

  - If correct (i.e., $\hat{y}$=y), no change!
  - If wrong: adjust the weight vector by adding or subtracting the feature vector. Subtract if $\hat{y}$ is 0

$$w = w + (y - \hat{y}) \cdot x$$



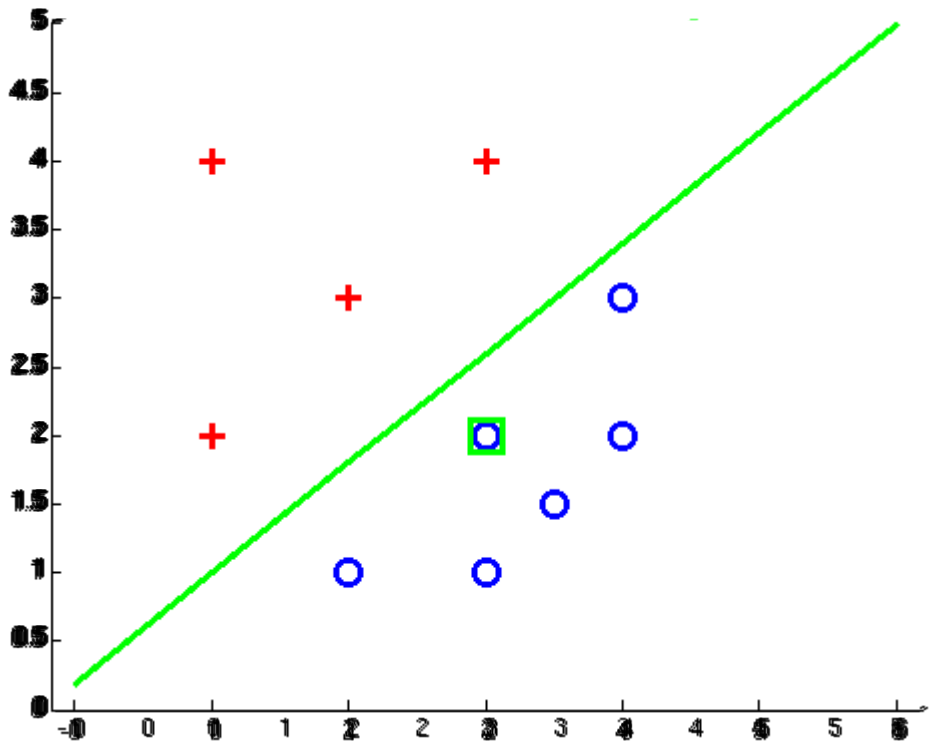
Linear classifier & stochastic gradient descent

- Linear regression
  - $h_w(x) = \sum_i^n w_i \, x_i = w \cdot x$
  - Update rule: $w_j \leftarrow w_j + \alpha \, (y^{(k)} - h_w(x^{(k)}))x_j^{(k)}$

Linear classifier & stochastic gradient descent

- Linear regression
  - $h_w(x) = \sum_i^n w_i \, x_i = w \cdot x$
  - Update rule: $w_j \leftarrow w_j + \alpha \, (y^{(k)} - h_w(x^{(k)}))x_j^{(k)}$
- Linear classifier
  - $h_w(x) = \begin{cases} 1 & \text{if } w \cdot x \geq 0 \\ 0 & \text{if } w \cdot x < 0 \end{cases}$
  - Update rule: $w_j \leftarrow w_j + \alpha \, (y^{(k)} - h_w(x^{(k)}))x_j^{(k)}$

Separable case example



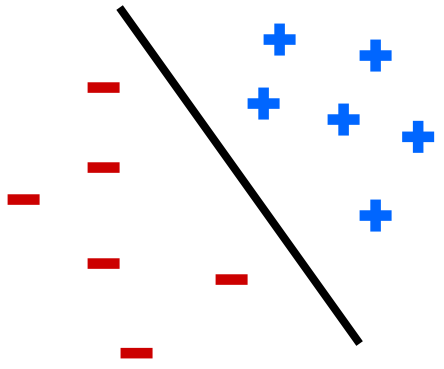
Properties of perceptrons

- Separability: true if some parameters get the training set perfectly correct

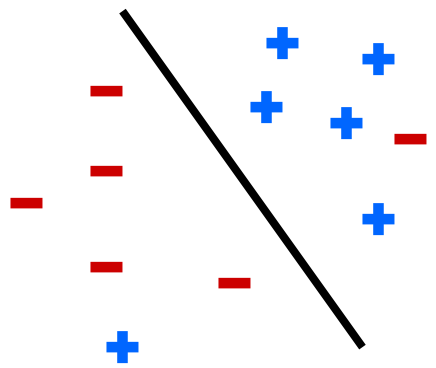
- Convergence: if the training is separable, binary perceptron will eventually converge

- Mistake bound: the maximum number of mistakes related to the *margin* or degree of separability (binary case) is $\frac{k}{\delta^2}$

Separable



Non-Separable



Issues

- Noise: if the data isn't separable, weights might thrash
- Mediocre generalization: finds a "barely" separating solution
- Overtraining: test / held-out accuracy usually rises, then falls
  - Overtraining is a kind of overfitting
  - We will see this next week