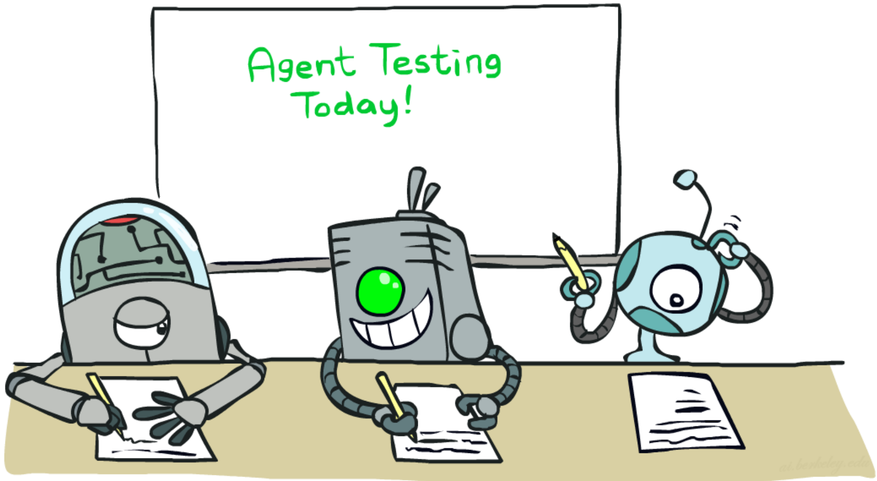
# CSE 573 : Artificial Intelligence

Hanna Hajishirzi

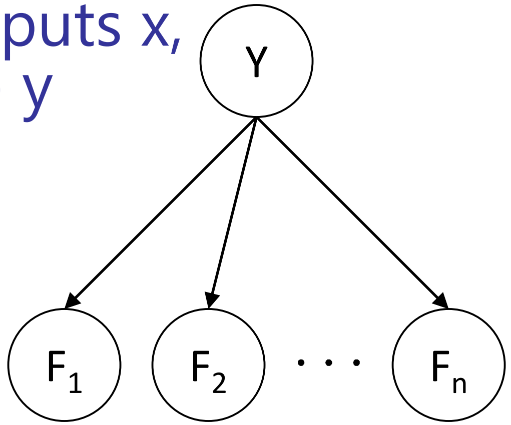
Perceptrons and Logistic Regression



slides adapted from
Dan Klein, Pieter Abbeel ai.berkeley.edu
And Dan Weld, Luke Zettlemoyer

# Last Lecture

- Classification: given inputs x, predict labels (classes) y



- Naïve Bayes

$$P(Y|F_{0,0} \ldots F_{15,15}) \propto P(Y) \prod_{i,j} P(F_{i,j}|Y)$$

- Parameter estimation:
  - MLE, MAP, priors

$$P_{\text{ML}}(x) = \frac{\text{count}(x)}{\text{total samples}}$$

- Laplace smoothing

$$P_{LAP,k}(x) = \frac{c(x)+k}{N+k|X|}$$

- Training set, held-out set, test set

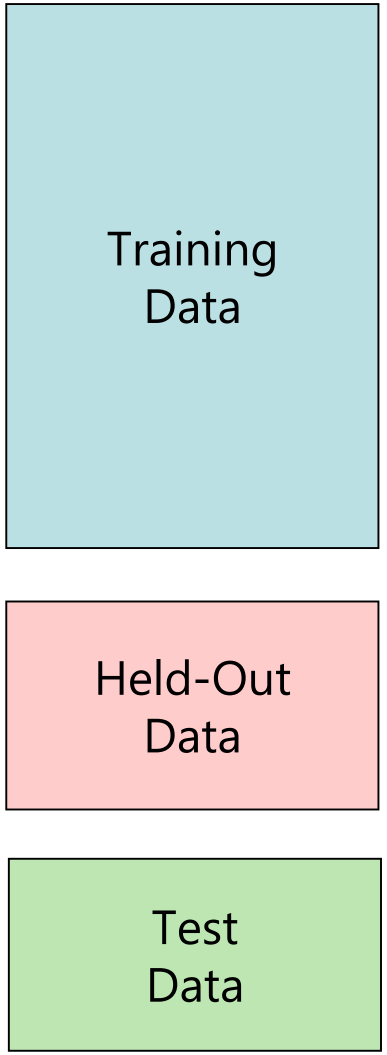
# Workflow

- **Phase 1: Train model on Training Data. Choice points for “tuning”**
  - Attributes / Features
  - Model types: Naïve Bayes vs. Perceptron vs. Logistic Regression vs. Neural Net etc..
  - Model hyperparameters
    - E.g. Naïve Bayes – Laplace k
    - E.g. Logistic Regression – weight regularization
    - E.g. Neural Net – architecture, learning rate, …
  - Make sure good performance on training data (why?)

- **Phase 2: Evaluate on Hold-Out Data**
  - If Hold-Out performance is close to Train performance
    - We achieved good generalization, onto Phase 3! ☺
  - If Hold-Out performance is much worse than Train performance
    - We overfitted to the training data! ☹
    - Take inspiration from the errors and:
      - Either: go back to Phase 1 for tuning (typically: make the model less expressive)
      - Or: if we are out of options for tuning while maintaining high train accuracy, collect more data (i.e., let the data drive generalization, rather than the tuning/regularization) and go to Phase 1

- **Phase 3: Report performance on Test Data**

**Possible outer-loop: Collect more data ☺**



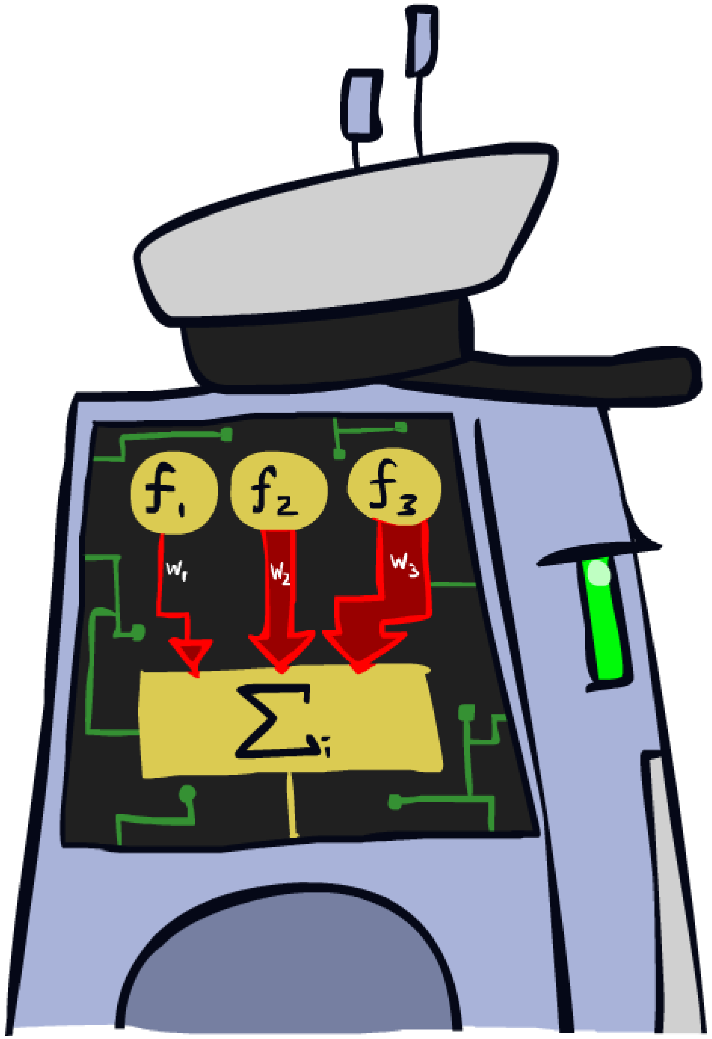
# Practical Tip: Baselines

- First step: get a baseline
  - Baselines are very simple “straw man” procedures
  - Help determine how hard the task is
  - Help know what a “good” accuracy is

- Weak baseline: most frequent label classifier
  - Gives all test instances whatever label was most common in the training set
  - E.g. for spam filtering, might label everything as ham
  - Accuracy might be very high if the problem is skewed
  - E.g. calling everything “ham” gets 66%, so a classifier that gets 70% isn’t very good…

- For real research, usually use previous work as a (strong) baseline

# Linear Classifiers



# Feature Vectors

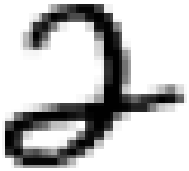
$x$ $f(x)$ $y$

```
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
+

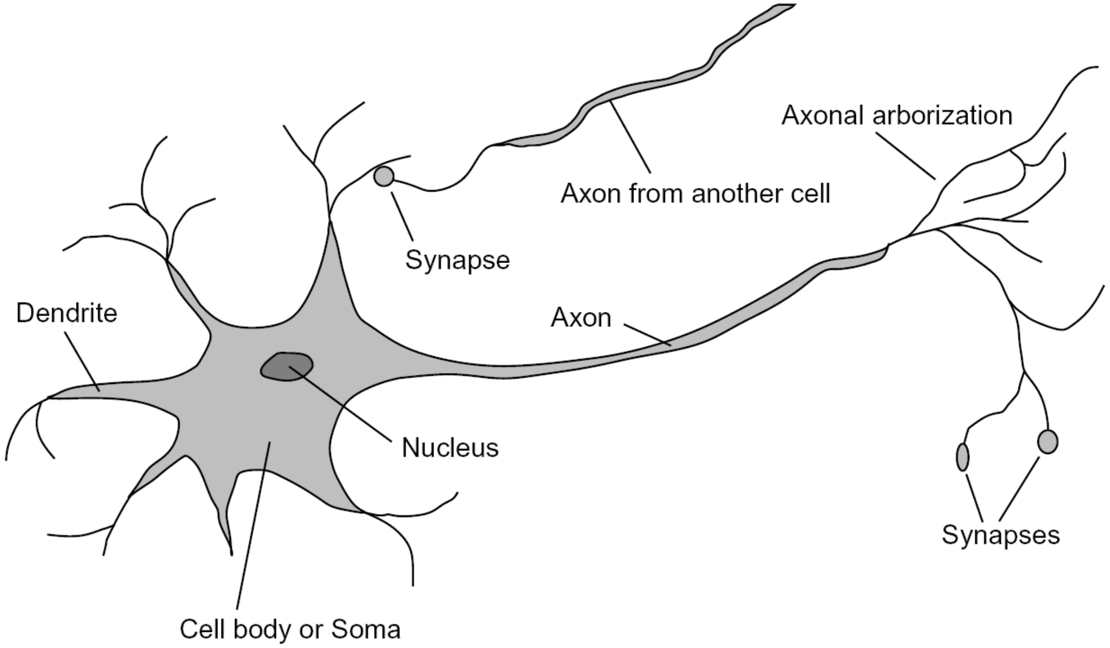


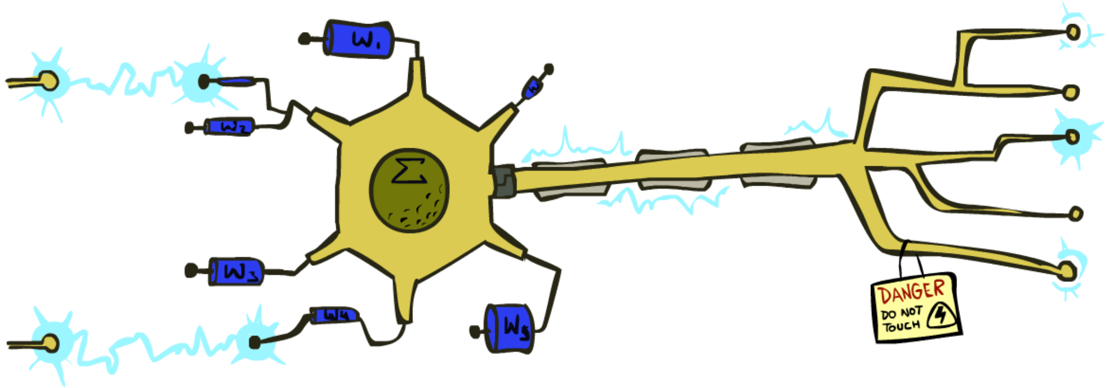
```
PIXEL-7,12   : 1
PIXEL-7,13   : 0
...
NUM_LOOPS    : 1
...
```

“2”

# Some (Simplified) Biology

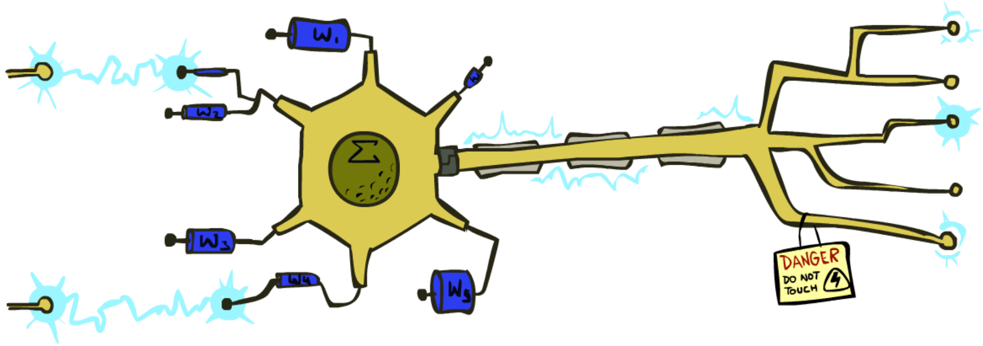
- Very loose inspiration: human neurons





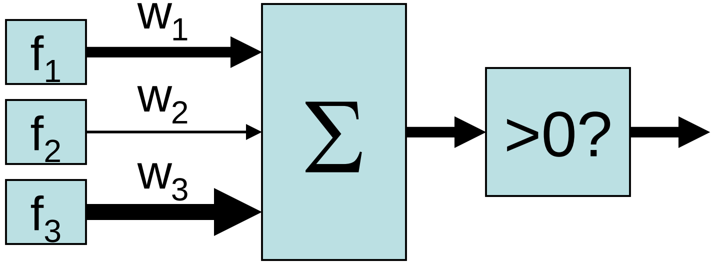
# Linear Classifiers

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



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

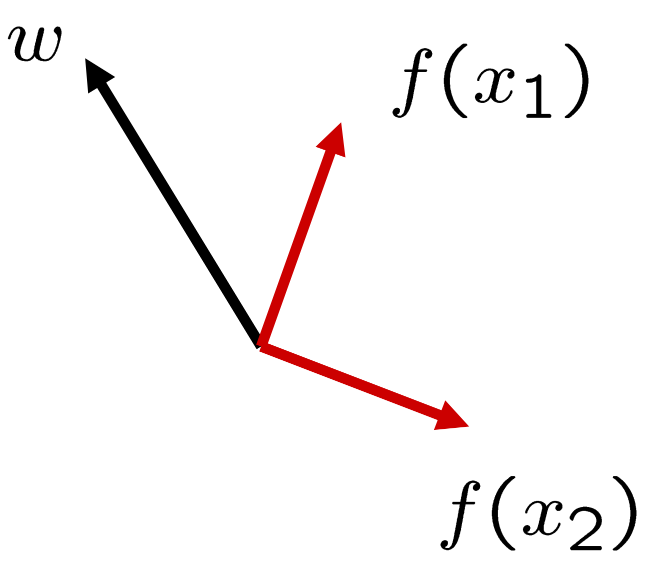
- If the activation is:
  - Positive, output +1
  - Negative, output -1



# Weights

- Binary case: compare features to a weight vector
- Learning: figure out the weight vector from examples

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

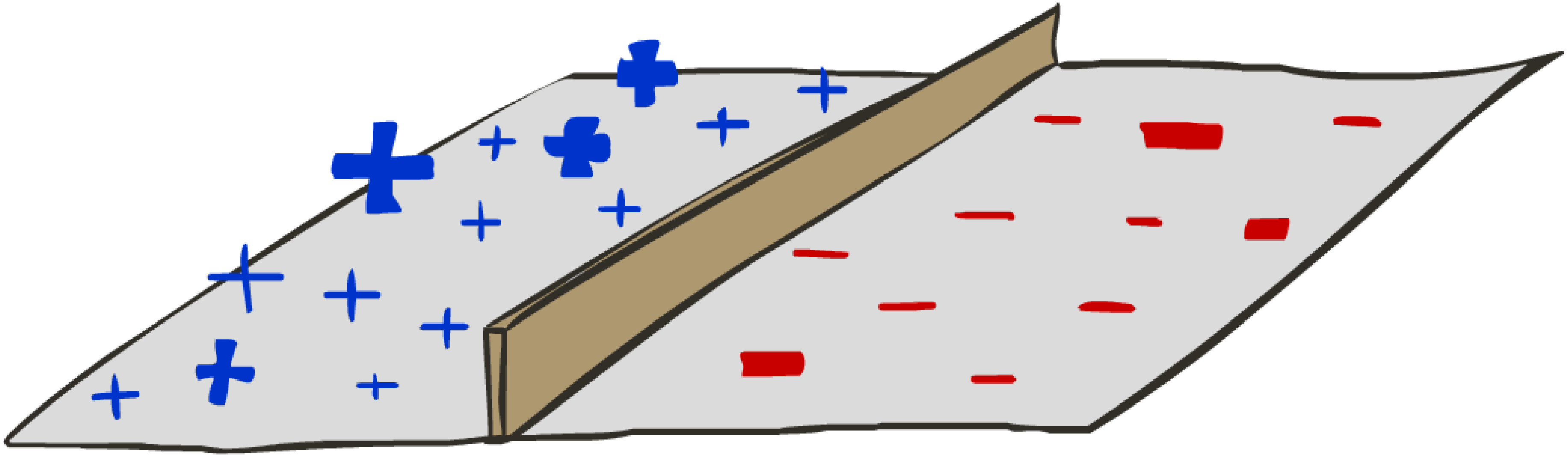


```
# free      : 2
YOUR_NAME   : 0
MISSPELLED  : 2
FROM_FRIEND : 0
...
```

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

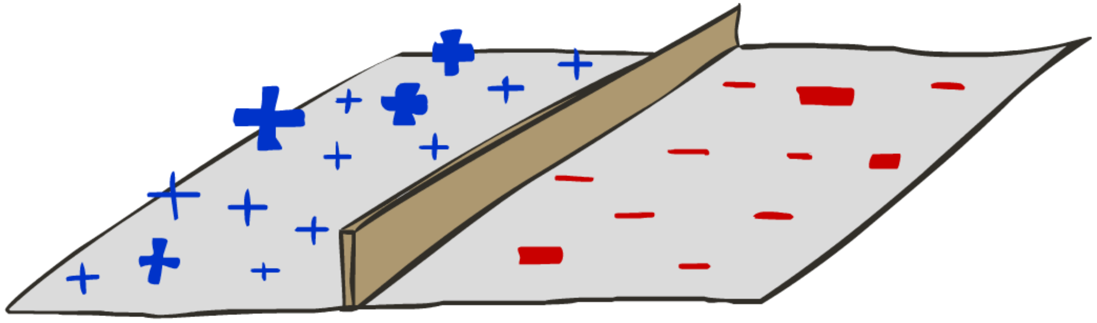
*Dot product* $w \cdot f$ *positive means the positive class*

# Decision Rules



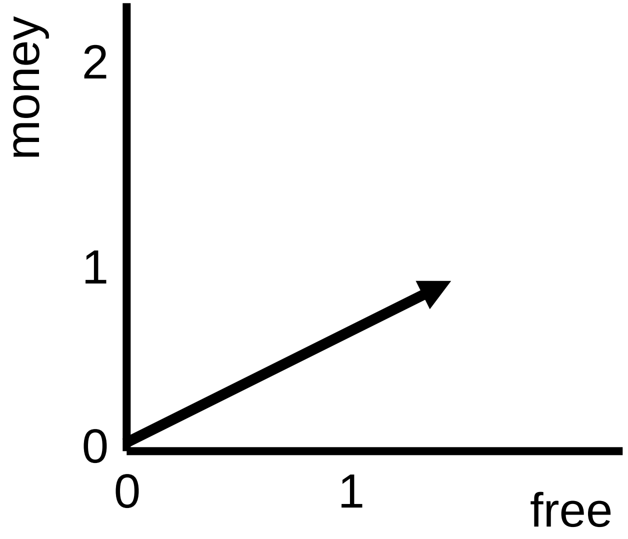
# Binary Decision Rule

- In the space of feature vectors
  - Examples are points
  - Any weight vector is a hyperplane
  - One side corresponds to Y=+1
  - Other corresponds to Y=-1



$$w$$

```
BIAS  : -3
free  :  4
money :  2
...
```

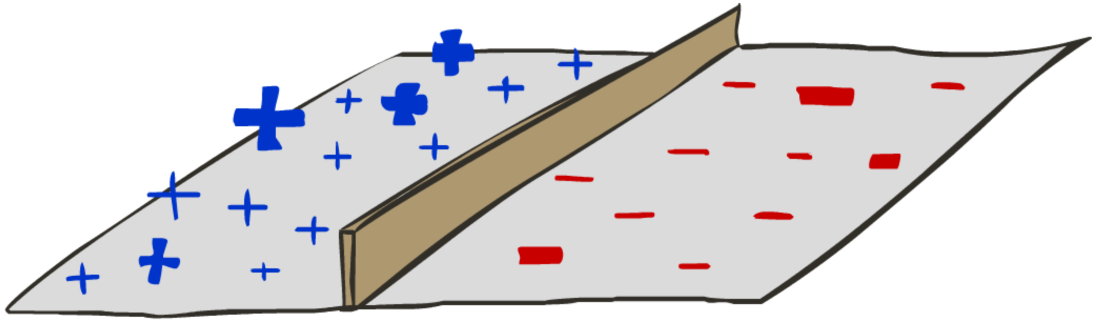


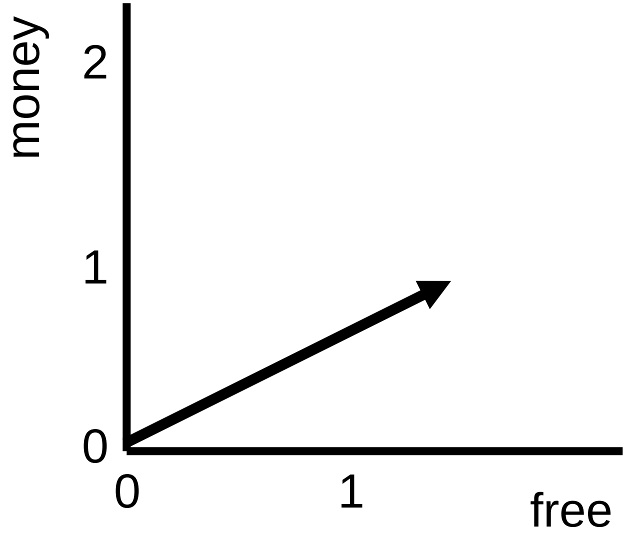
# Binary Decision Rule

- In the space of feature vectors
  - Examples are points
  - Any weight vector is a hyperplane
  - One side corresponds to Y=+1
  - Other corresponds to Y=-1

$$w$$

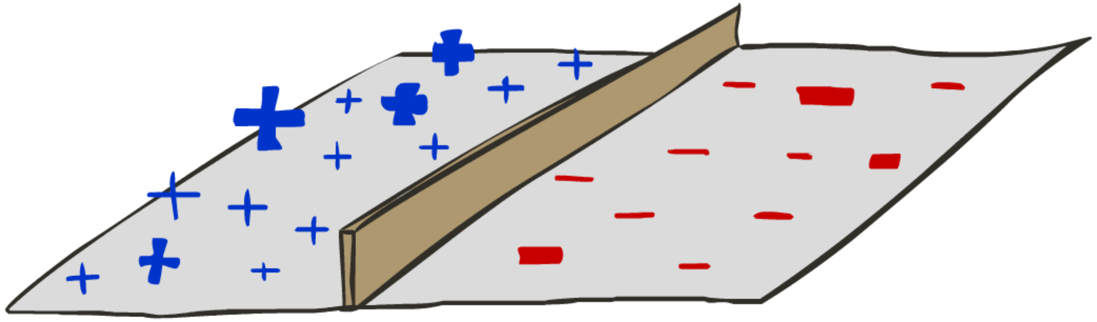
| | | |
|---|---|---|
| free | : | 4 |
| money | : | 2 |





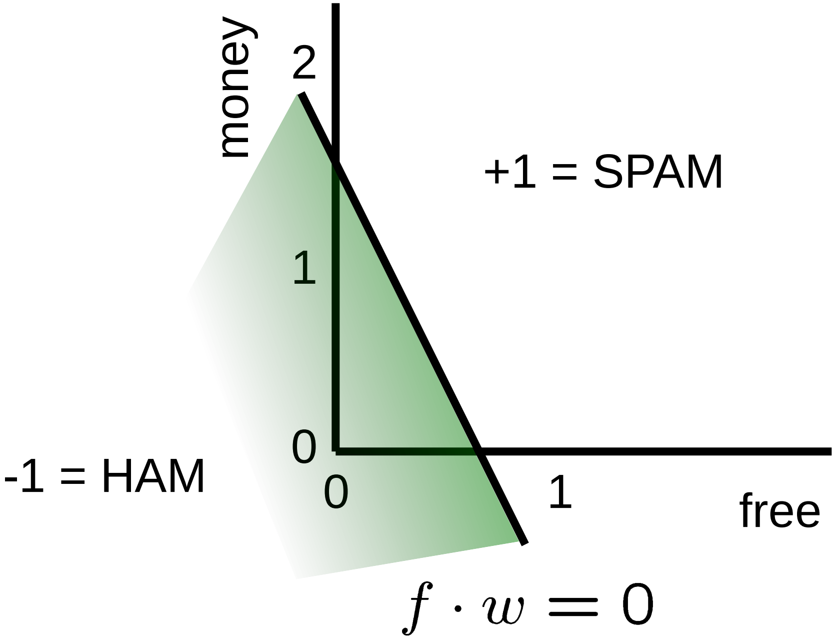
# Binary Decision Rule

- In the space of feature vectors
  - Examples are points
  - Any weight vector is a hyperplane
  - One side corresponds to Y=+1
  - Other corresponds to Y=-1



$w$

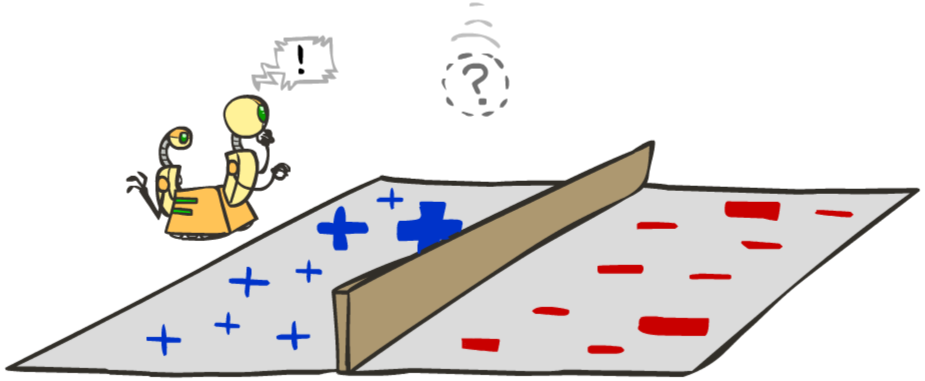
```
BIAS  : -3
free  :  4
money :  2
...
```



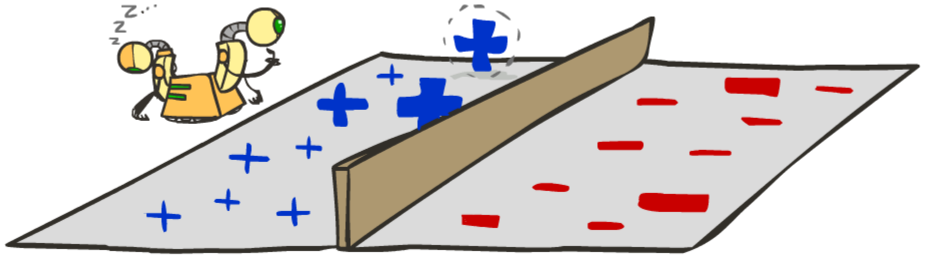
# Weight Updates

# Learning: Binary Perceptron

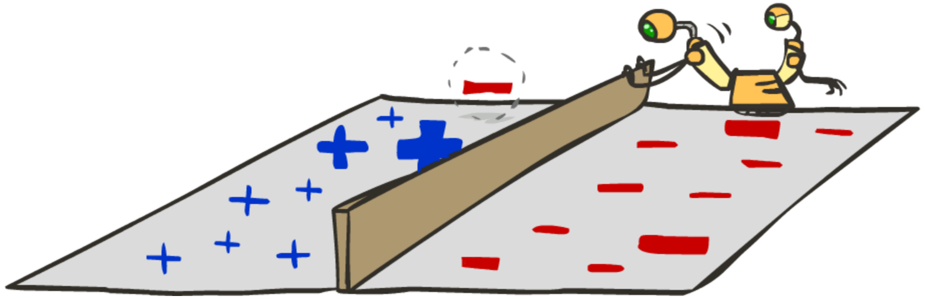
- Start with weights = 0
- For each training instance:
    - Classify with current weights



    - If correct (i.e., y=y*), no change!



    - If wrong: adjust the weight vector



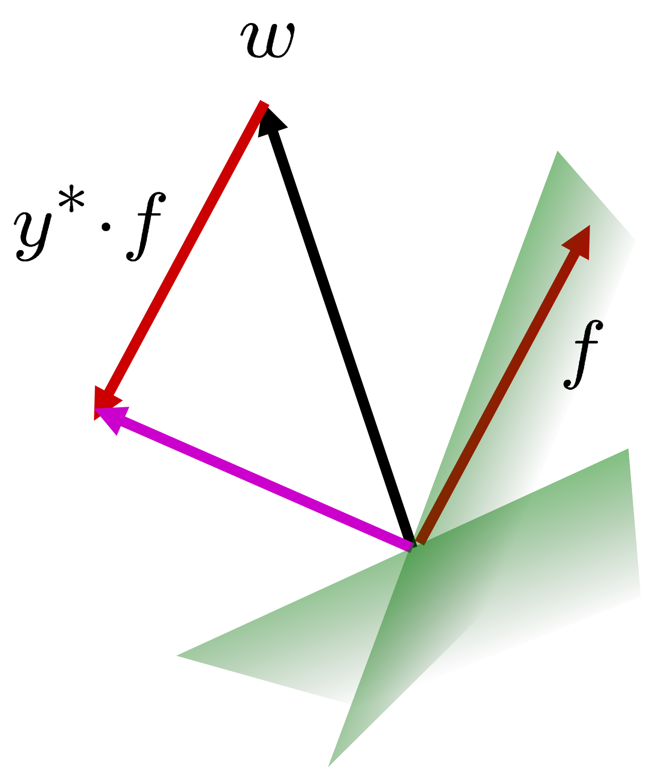
# Learning: Binary Perceptron

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

$$y = \begin{cases} +1 & \text{if } \; w \cdot f(x) \geq 0 \\ -1 & \text{if } \; w \cdot f(x) < 0 \end{cases}$$

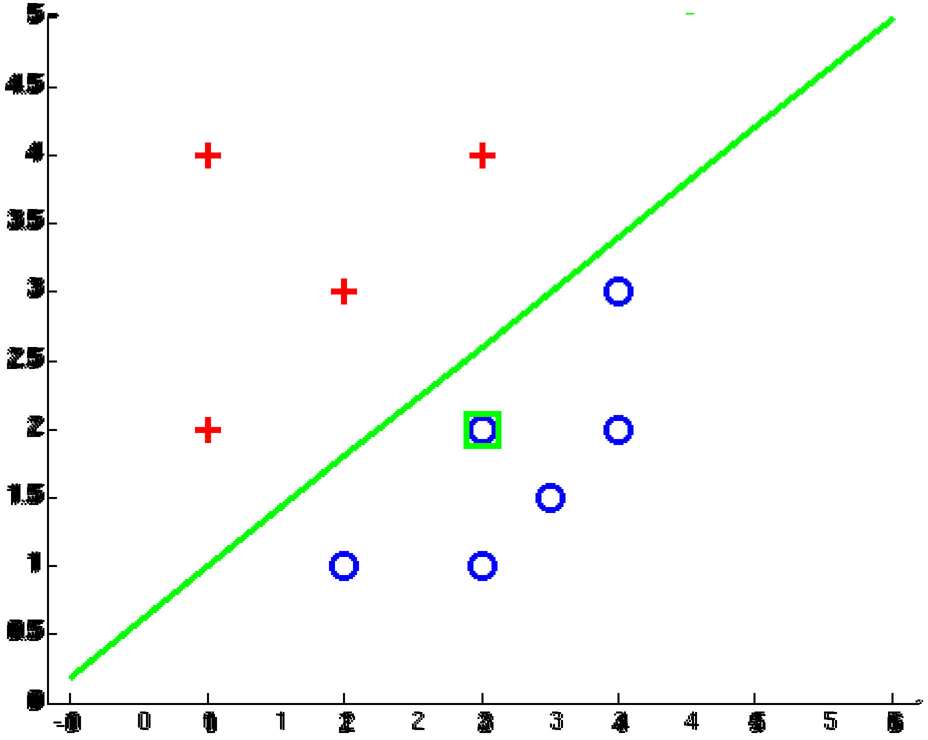
  - If correct (i.e., y=y*), no change!
  - If wrong: adjust the weight vector by adding or subtracting the feature vector. Subtract if y* is -1.

$$w = w + y^* \cdot f$$



# Examples: Perceptron

- Separable Case



# Multiclass Decision Rule

- If we have multiple classes:
  - A weight vector for each class:

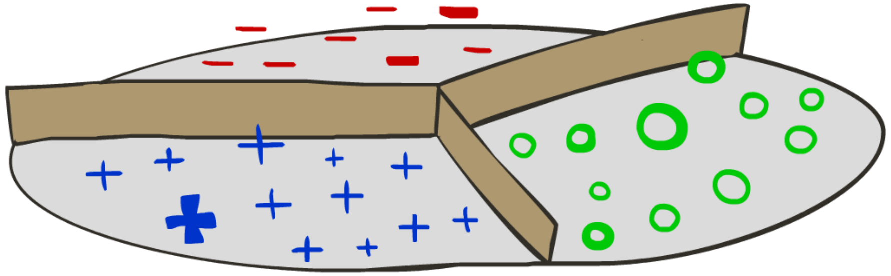
$$w_y$$

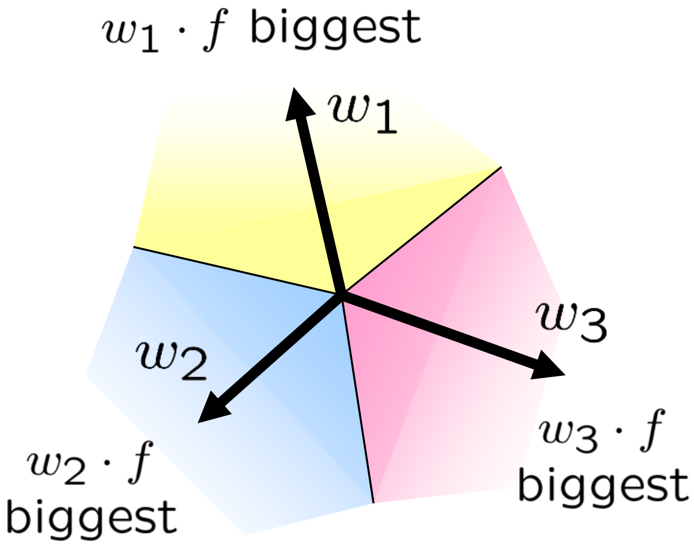
  - Score (activation) of a class y:

$$w_y \cdot f(x)$$

  - Prediction highest score wins

$$y = \arg\max_y \; w_y \cdot f(x)$$





*Binary = multiclass where the negative class has weight zero*

# Learning: Multiclass Perceptron

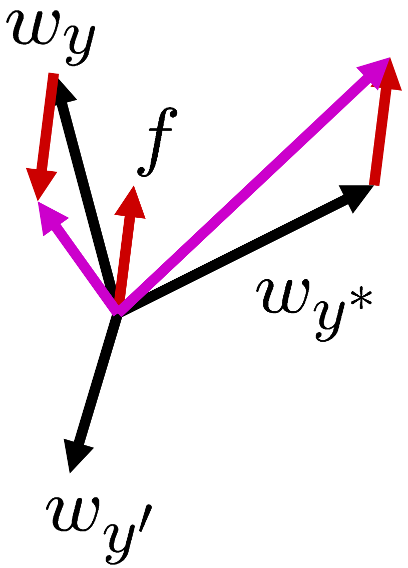
- Start with all weights = 0
- Pick up training examples one by one
- Predict with current weights

$$y \quad = \arg\max_y \; w_y \cdot f(x)$$

- If correct, no change!
- If wrong: lower score of wrong answer, raise score of right answer

$$w_y = w_y - f(x)$$

$$w_{y^*} = w_{y^*} + f(x)$$



# Example: Multiclass Perceptron

“win the vote” [1 1 0 1 1]

“win the election” [1 1 0 0 1]

“win the game” [1 1 1 0 1]

$w_{SPORTS}$

1 -2 -2

| | | |
|---|---|---|
| BIAS : 1 | 0 | 1 |
| win : 0 | -1 | 0 |
| game : 0 | 0 | 1 |
| vote : 0 | -1 | -1 |
| the : 0 | -1 | 0 |
| ... | | |

$w_{POLITICS}$

0 3 3

| | | |
|---|---|---|
| BIAS : 0 | 1 | 0 |
| win : 0 | 1 | 0 |
| game : 0 | 0 | -1 |
| vote : 0 | 1 | 1 |
| the : 0 | 1 | 0 |
| ... | | |

$w_{TECH}$

0 0

```
BIAS  : 0
win   : 0
game  : 0
vote  : 0
the   : 0
...
```

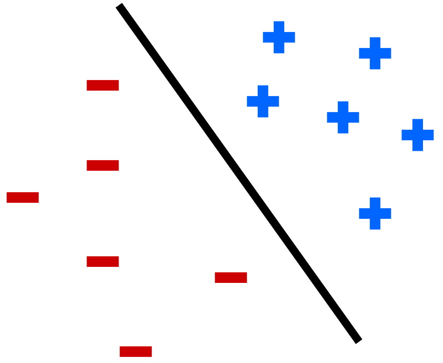
# Properties of Perceptrons

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

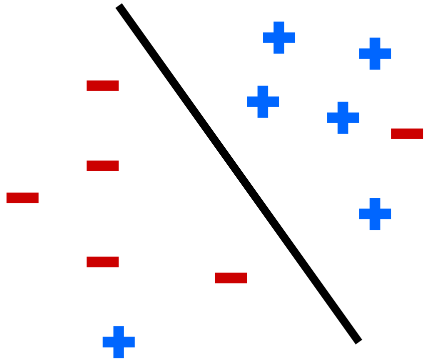
- Convergence: if the training is separable, perceptron will eventually converge (binary case)

- Non-separable?

Separable

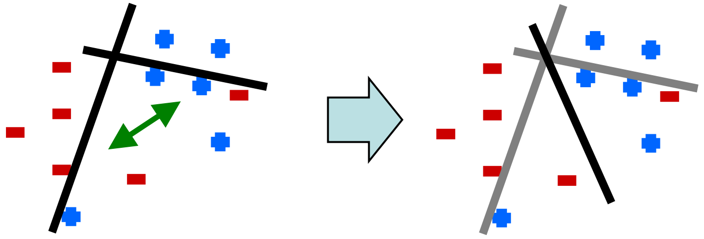


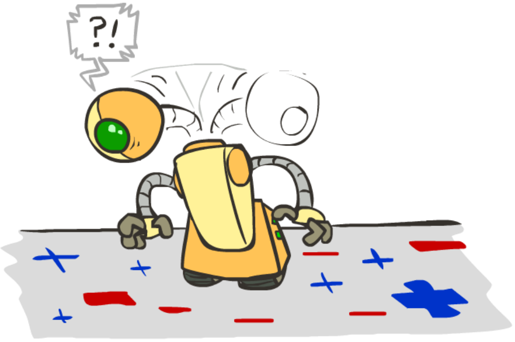
Non-Separable



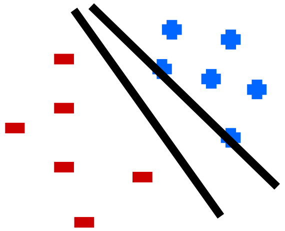
# Problems with the Perceptron

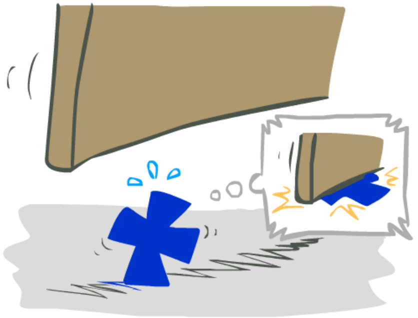
- Noise: if the data isn’t separable, weights might thrash
  - Averaging weight vectors over time can help (averaged perceptron)



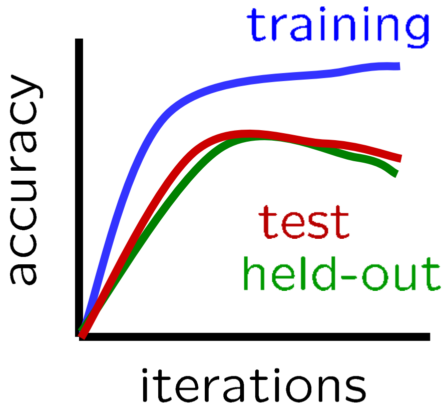


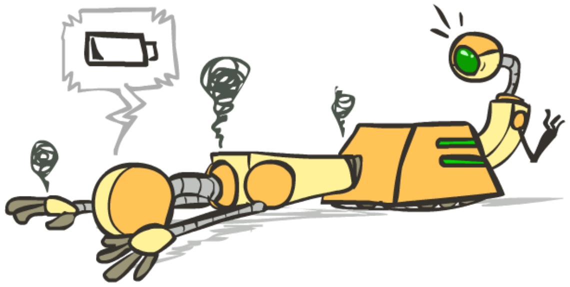
- Mediocre generalization: finds a “barely” separating solution



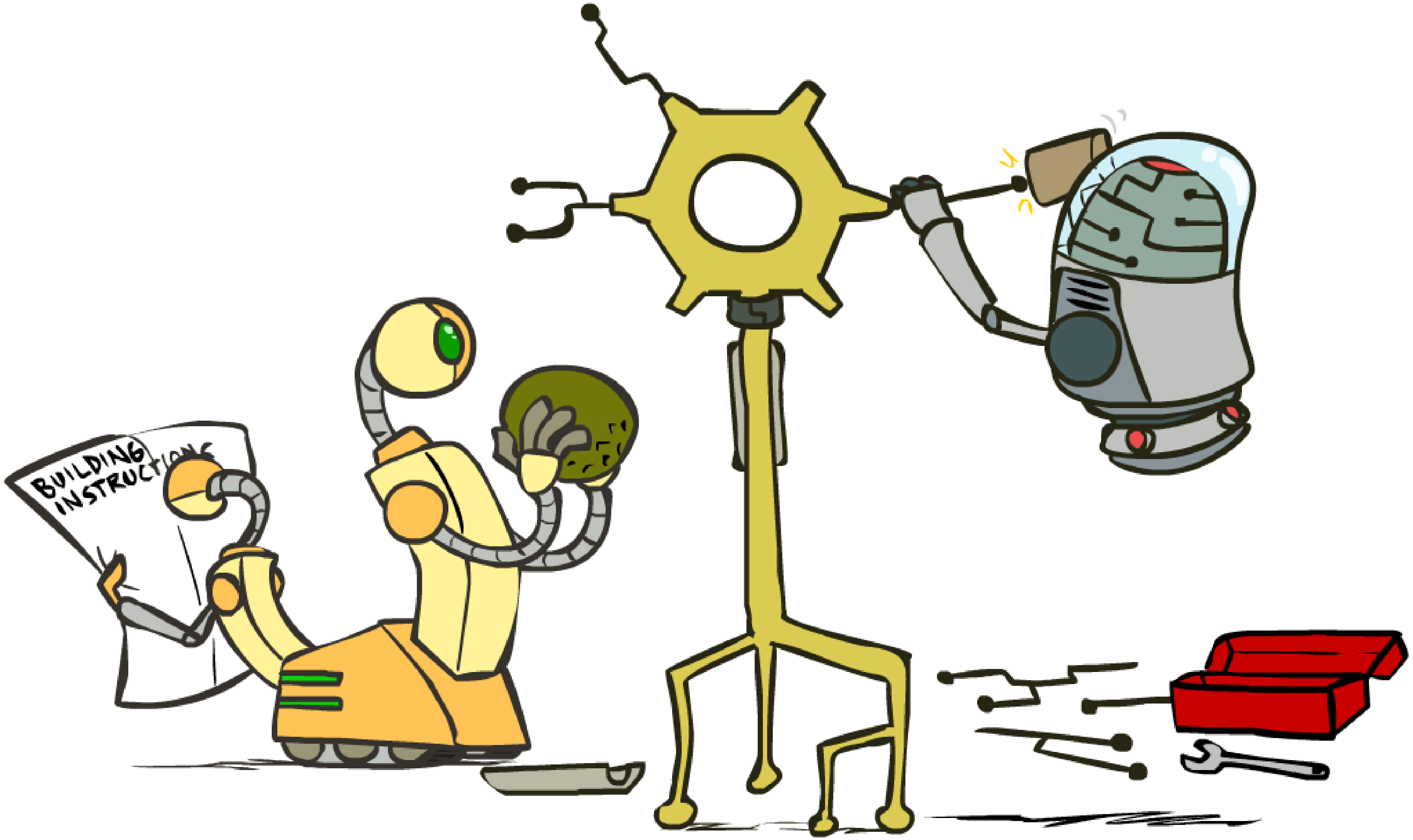


- Overtraining: test / held-out accuracy usually rises, then falls
  - Overtraining is a kind of overfitting



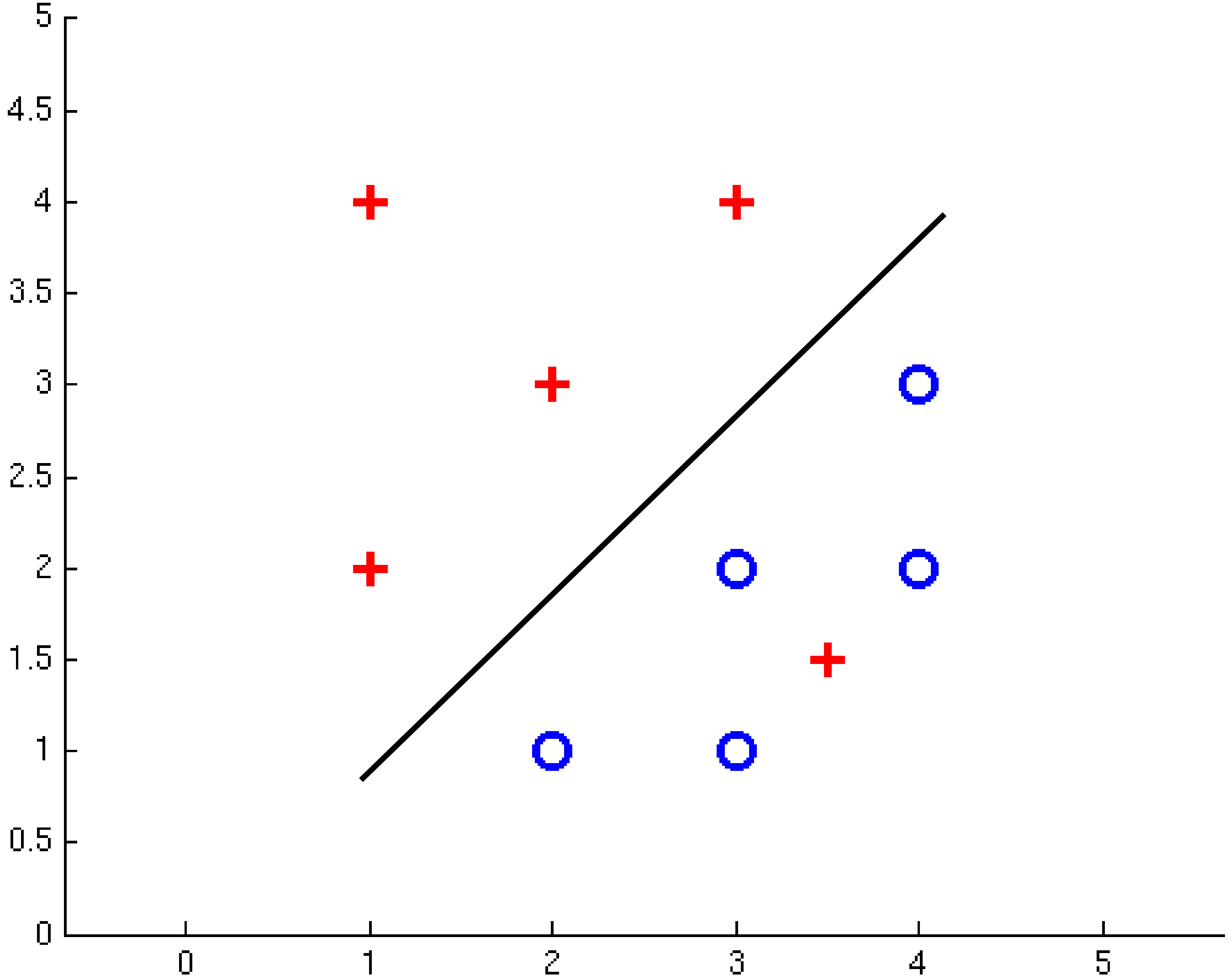


# Improving the Perceptron

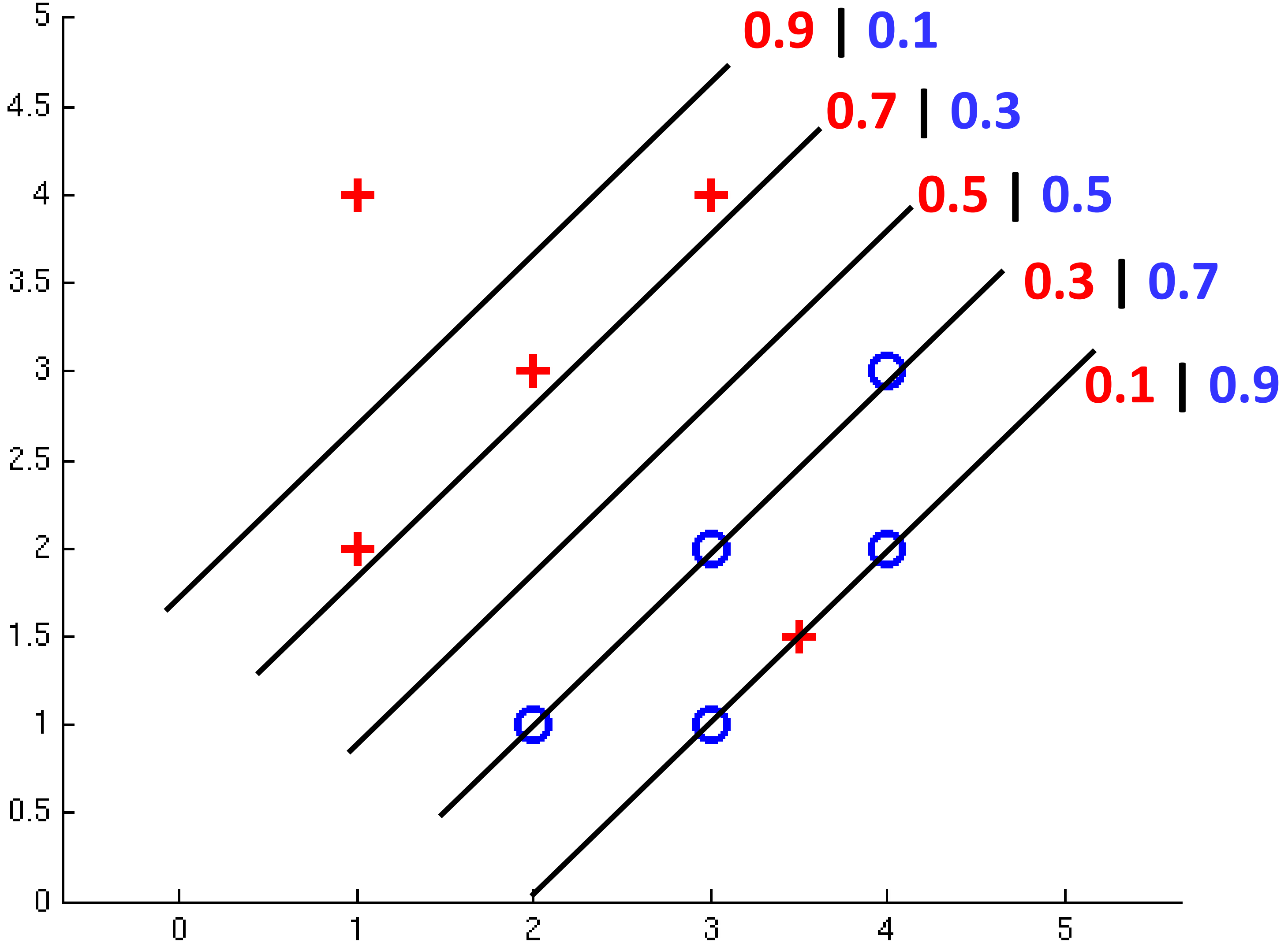


# Non-Separable Case: Deterministic Decision

Even the best linear boundary makes at least one mistake



# Non-Separable Case: Probabilistic Decision

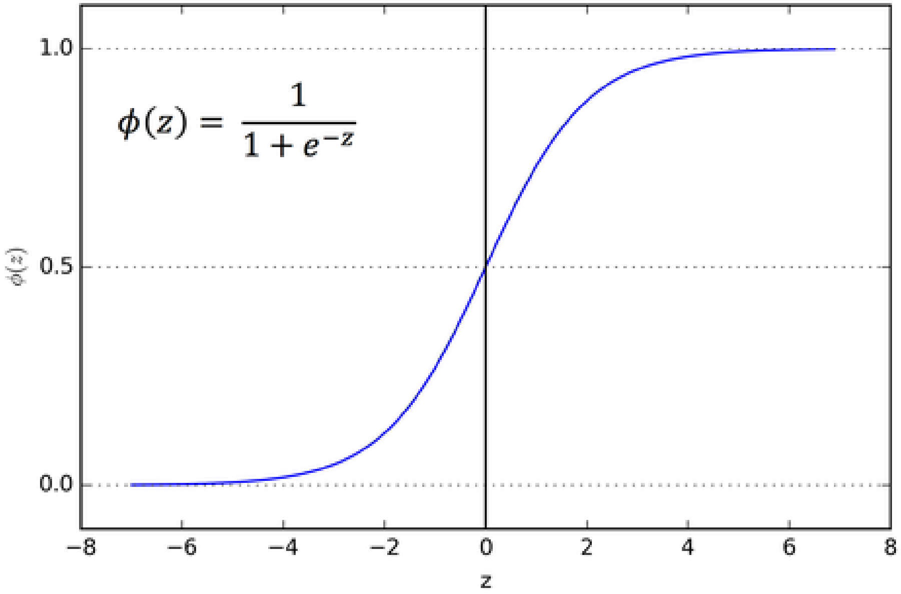


# How to get probabilistic decisions?

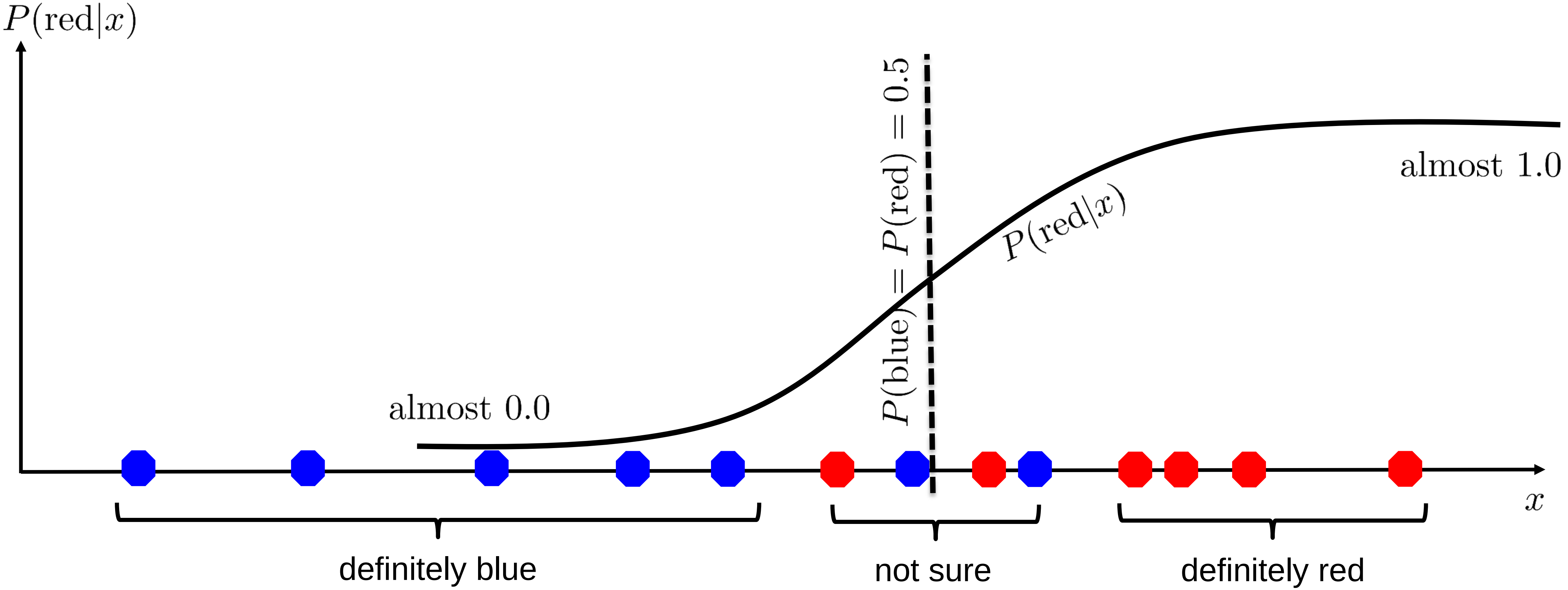
- Perceptron scoring: $z = w \cdot f(x)$
- If $z = w \cdot f(x)$ very positive → want probability going to 1
- If $z = w \cdot f(x)$ very negative → want probability going to 0

- Sigmoid function

$$\phi(z) = \frac{1}{1 + e^{-z}}$$



# A 1D Example

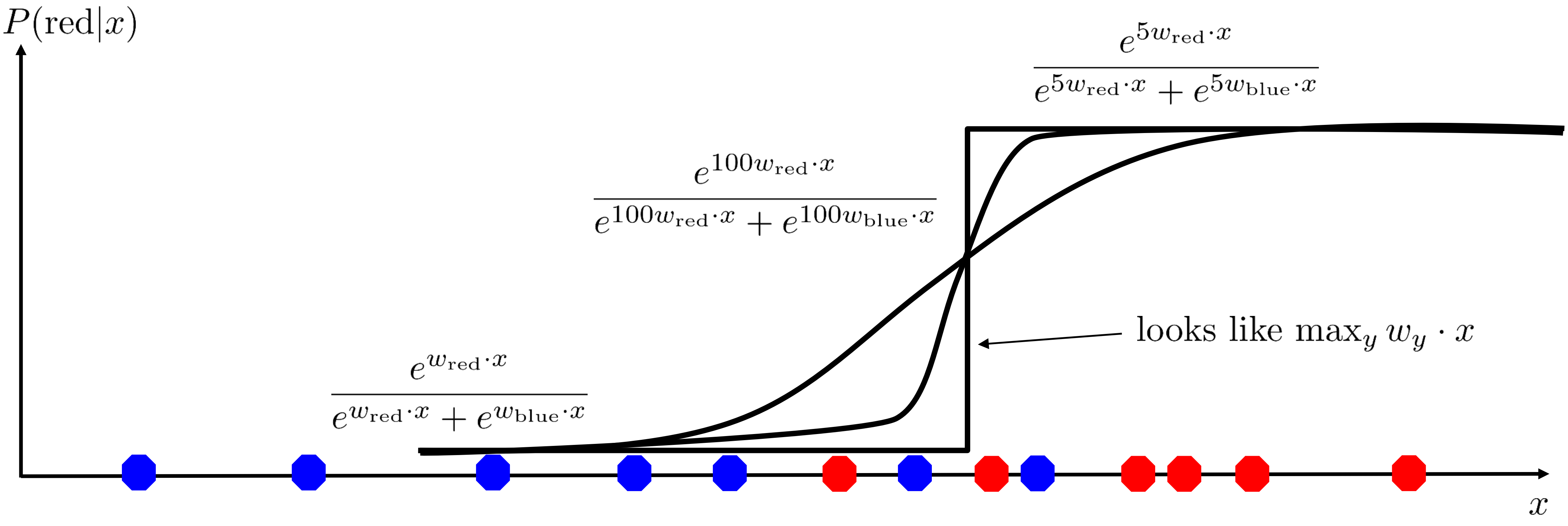


$$P(\text{red}|x) = \frac{e^{w_{\text{red}} \cdot x}}{e^{w_{\text{red}} \cdot x} + e^{w_{\text{blue}} \cdot x}}$$

probability increases exponentially
as we move away from boundary

normalizer

# The *Soft* Max



$$P(\text{red}|x) = \frac{e^{w_{\text{red}}\cdot x}}{e^{w_{\text{red}}\cdot x}+e^{w_{\text{blue}}\cdot x}}$$

# Best w?

- Maximum likelihood estimation:

$$\max_{w} \quad ll(w) = \max_{w} \quad \sum_{i} \log P(y^{(i)}|x^{(i)}; w)$$

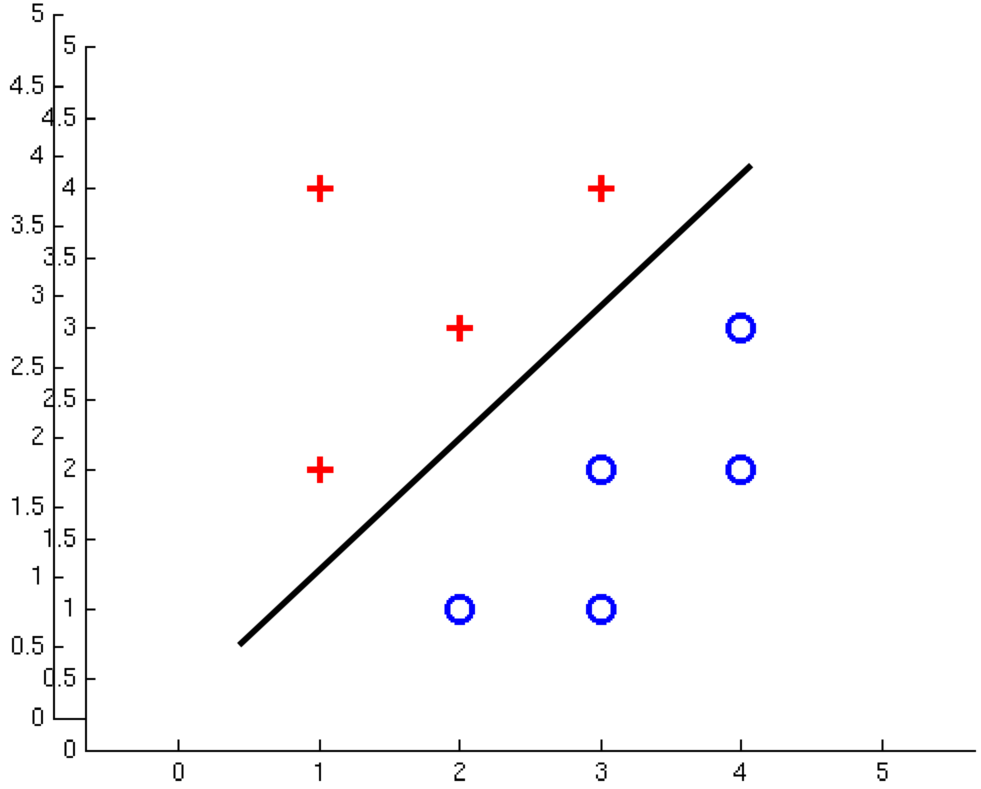
with:

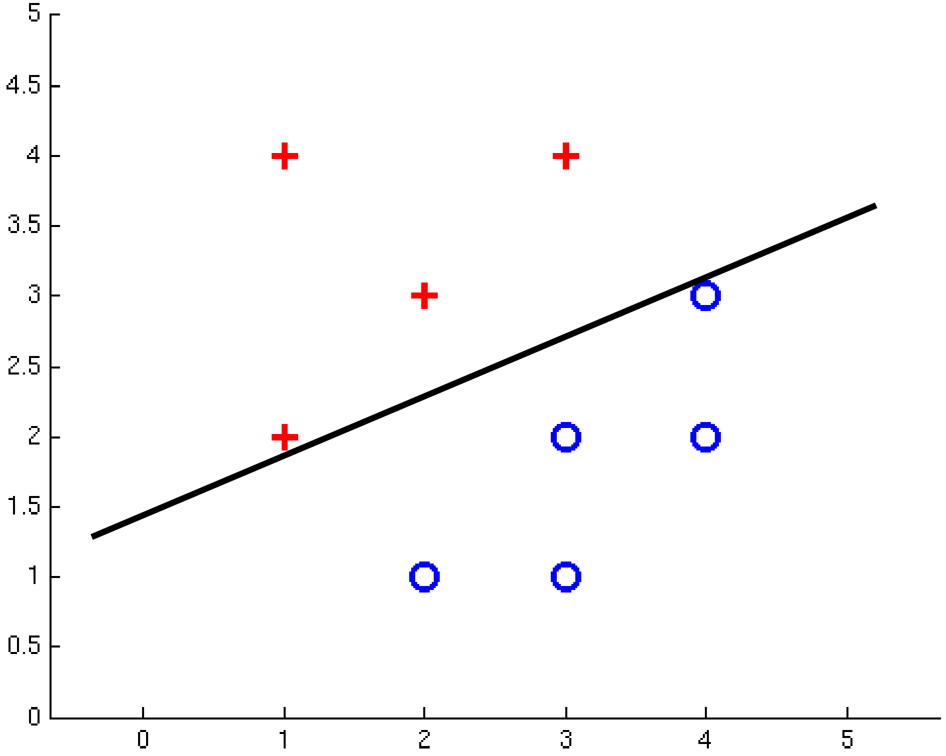
$$P(y^{(i)} = +1|x^{(i)}; w) = \frac{1}{1 + e^{-w \cdot f(x^{(i)})}}$$

$$P(y^{(i)} = -1|x^{(i)}; w) = 1 - \frac{1}{1 + e^{-w \cdot f(x^{(i)})}}$$

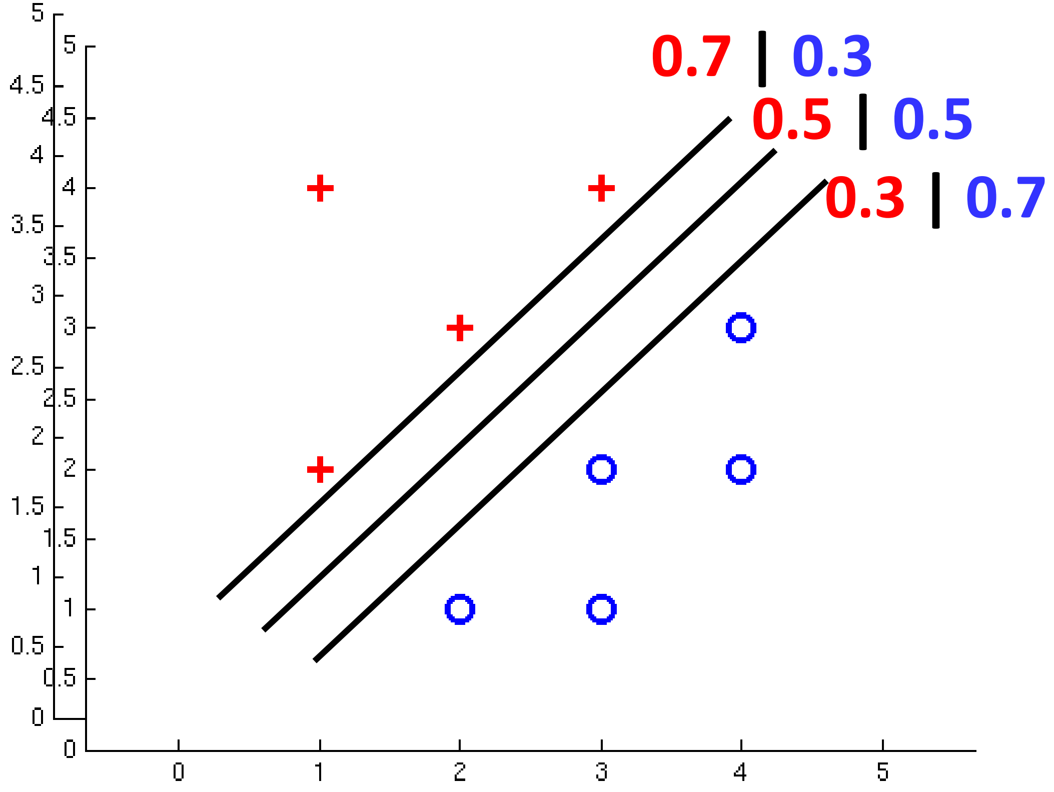
**= Logistic Regression**

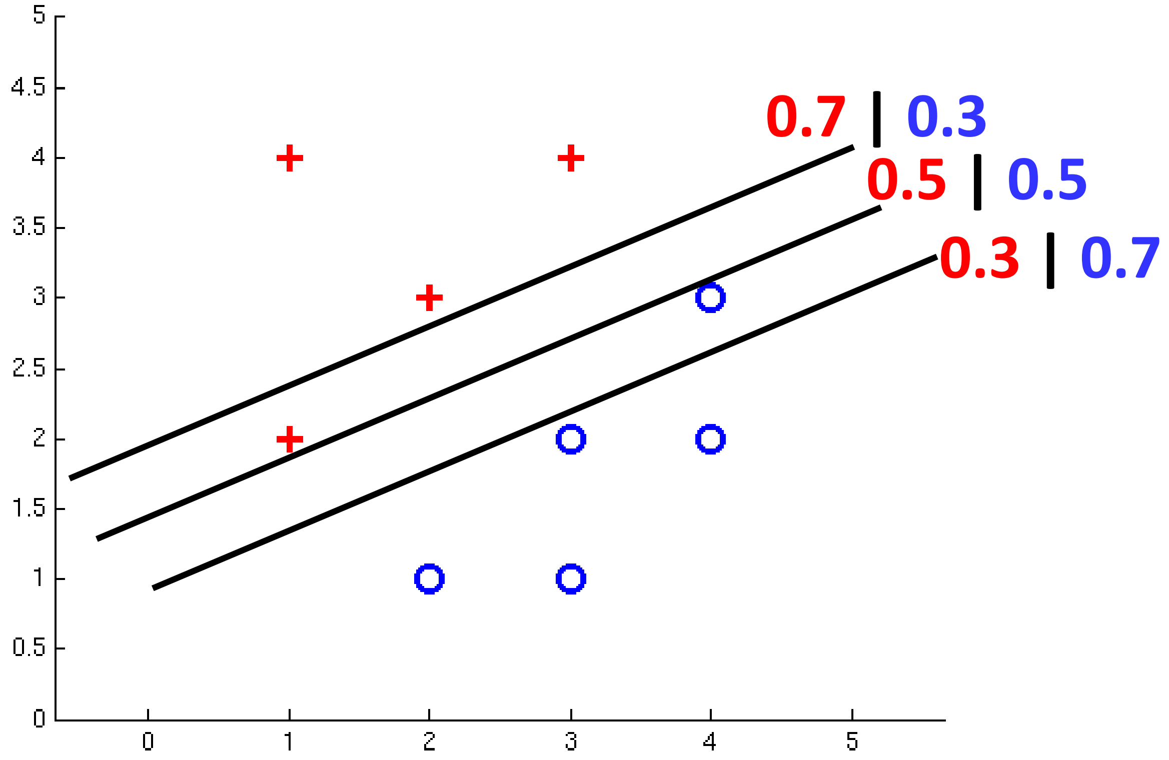
# Separable Case: Deterministic Decision – Many Options





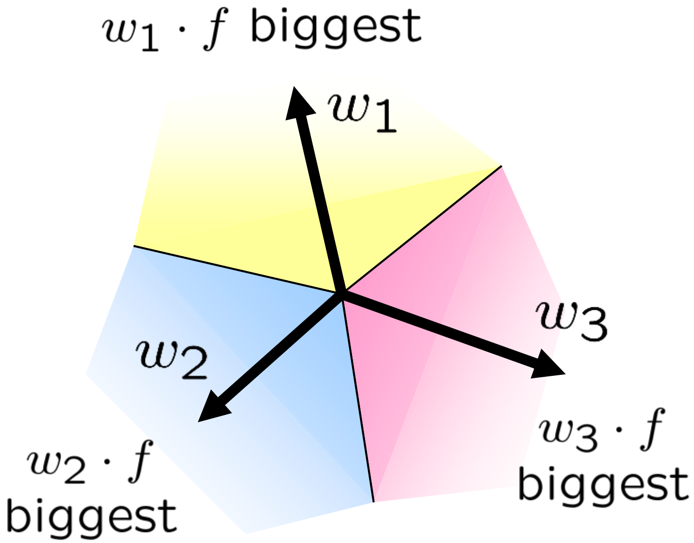
# Separable Case: Probabilistic Decision – Clear Preference





# Multiclass Logistic Regression

- Recall Perceptron:
  - A weight vector for each class: $w_y$
  - Score (activation) of a class y: $w_y \cdot f(x)$
  - Prediction highest score wins $y = \arg\max_y \; w_y \cdot f(x)$



- How to make the scores into probabilities?

$$\underbrace{z_1, z_2, z_3}_{\text{original activations}} \rightarrow \underbrace{\frac{e^{z_1}}{e^{z_1}+e^{z_2}+e^{z_3}}, \frac{e^{z_2}}{e^{z_1}+e^{z_2}+e^{z_3}}, \frac{e^{z_3}}{e^{z_1}+e^{z_2}+e^{z_3}}}_{\text{softmax activations}}$$

# Best w?

- Maximum likelihood estimation:

$$\max_{w} \quad ll(w) = \max_{w} \quad \sum_{i} \log P(y^{(i)}|x^{(i)}; w)$$

with: $$P(y^{(i)}|x^{(i)}; w) = \frac{e^{w_{y^{(i)}} \cdot f(x^{(i)})}}{\sum_{y} e^{w_y \cdot f(x^{(i)})}}$$

**= Multi-Class Logistic Regression**

# Best w?

- Optimization
  - i.e., how do we solve:

$$\max_w \quad ll(w) = \max_w \quad \sum_i \log P(y^{(i)}|x^{(i)}; w)$$

# Hill Climbing

- Simple, general idea
  - Start wherever
  - Repeat: move to the best neighboring state
  - If no neighbors better than current, quit

- What's particularly tricky when hill-climbing for multiclass logistic regression?
  - Optimization over a continuous space
    - Infinitely many neighbors!
    - How to do this efficiently?

# Gradient Ascent

- Perform update in uphill direction for each coordinate
- The steeper the slope (i.e. the higher the derivative) the bigger the step for that coordinate
- E.g., consider: $g(w_1, w_2)$
  - Updates:

    $$w_1 \leftarrow w_1 + \alpha * \frac{\partial g}{\partial w_1}(w_1, w_2)$$

    $$w_2 \leftarrow w_2 + \alpha * \frac{\partial g}{\partial w_2}(w_1, w_2)$$

  - Updates in vector notation:

    $$w \leftarrow w + \alpha * \nabla_w g(w)$$

    with: $\nabla_w g(w) = \begin{bmatrix} \frac{\partial g}{\partial w_1}(w) \\ \frac{\partial g}{\partial w_2}(w) \end{bmatrix}$ **= gradient**

# Gradient in n dimensions

$$\nabla g = \begin{bmatrix} \frac{\partial g}{\partial w_1} \\ \frac{\partial g}{\partial w_2} \\ \cdots \\ \frac{\partial g}{\partial w_n} \end{bmatrix}$$

# Optimization Procedure: Gradient Ascent

```
init $w$
for iter = 1, 2, …
        $w \leftarrow w + \alpha * \nabla g(w)$
```

- $\alpha$ : learning rate --- tweaking parameter that needs to be chosen carefully
- How? Try multiple choices
  - Crude rule of thumb: update changes $w$ about 0.1 – 1 %

# Batch Gradient Ascent on the Log Likelihood Objective

$$\max_w \; ll(w) = \max_w \; \underbrace{\sum_i \log P(y^{(i)}|x^{(i)};w)}_{g(w)}$$

```
init $w$
for iter = 1, 2, …
    $w \leftarrow w + \alpha * \sum_i \nabla \log P(y^{(i)}|x^{(i)};w)$
```

# Stochastic Gradient Ascent on the Log Likelihood Objective

$$\max_{w} \quad ll(w) = \max_{w} \quad \sum_{i} \log P(y^{(i)}|x^{(i)}; w)$$

**Observation:** 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 j
    $w \leftarrow w + \alpha * \nabla \log P(y^{(j)}|x^{(j)}; w)$
```

# Mini-Batch Gradient Ascent on the Log Likelihood Objective

$$\max_w \quad ll(w) = \max_w \quad \sum_i \log P(y^{(i)}|x^{(i)};w)$$

**Observation:** 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 J
      $w \leftarrow w + \alpha * \sum_{j \in J} \nabla \log P(y^{(j)}|x^{(j)};w)$
```

# How about computing all the derivatives?

- We'll talk about that in neural networks, which are a generalization of logistic regression