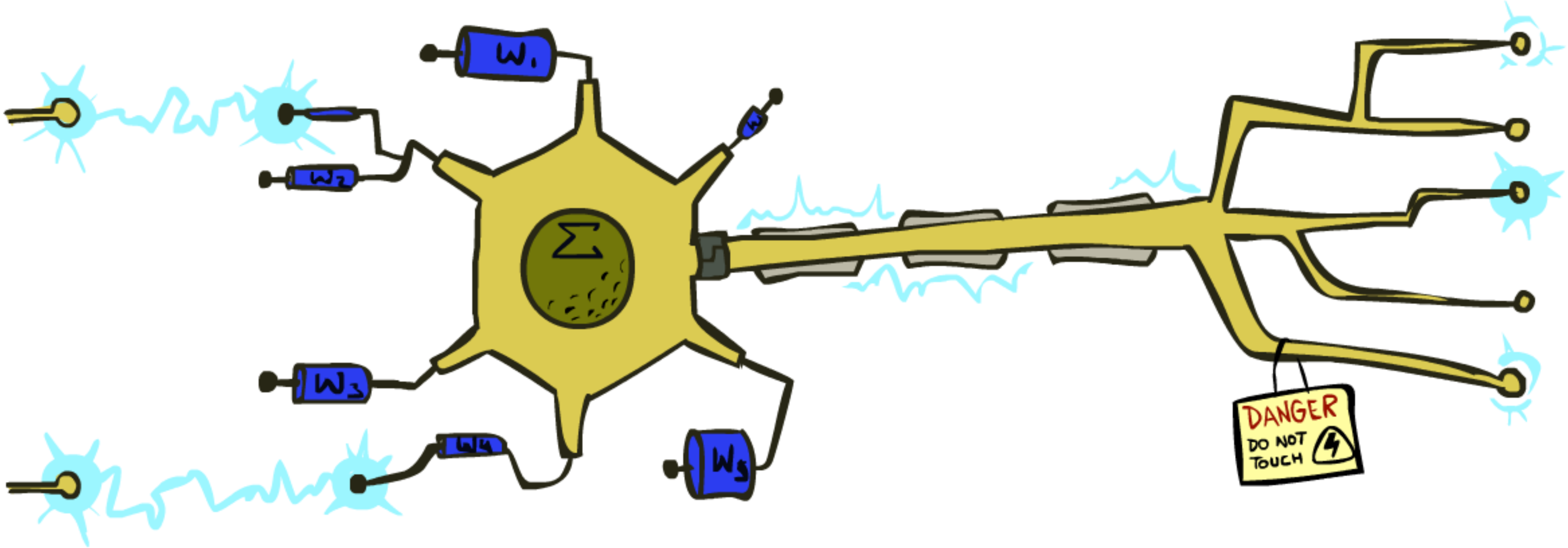
CS 188: Artificial Intelligence

Naïve Bayes and Perceptrons

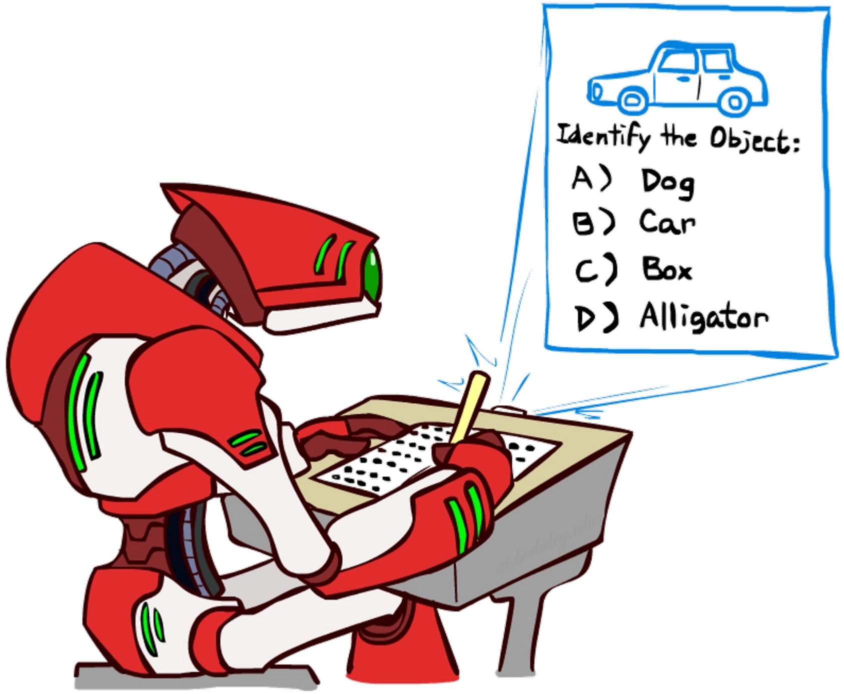


[These slides were created by Dan Klein, Pieter Abbeel, Anca Dragan, Sergey Levine. All CS188 materials are at http://ai.berkeley.edu.]

Last Time

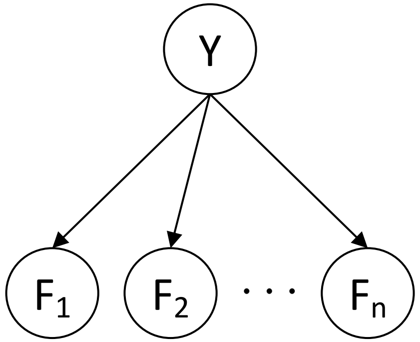
- **Classification:** given inputs x, predict labels (classes) y
  - Convert input x into a collection of *features* $f_1, \ldots, f_n$

$$\underset{\text{(input)}}{\mathbf{x}} \xrightarrow{\text{Feature extraction}} \underset{\text{(features)}}{\mathbf{f}} \xrightarrow{\text{Machine learning}} \underset{\text{(prediction)}}{\mathbf{y}}$$



Last Time

- *Naïve Bayes* model: $P(Y, F_1, \dots, F_n) = P(Y)\prod_i P(F_i|Y)$
  - Features and label are random variables
  - Input features $F_1, \dots, F_n$ are conditionally independent given label Y
  - Parameters θ: probability tables $P(Y), P(F_1|Y), \dots, P(F_n|Y)$



- Classification is inference in a Bayes Net:
  - Inference by enumeration
  - Given features $f_1, \dots, f_n$ probability over class labels is:

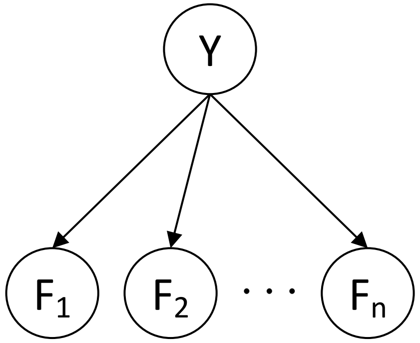
$$P(Y|f_1, \dots, f_n) \propto P(Y, f_1, \dots, f_n) = P(Y)\prod_i P(f_i|Y)$$

Enumerate over every label y:

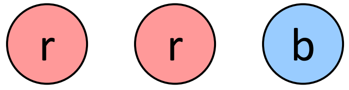
$$\begin{bmatrix} P(y_1)\prod_i P(f_i|y_1) \\ P(y_2)\prod_i P(f_i|y_2) \\ \vdots \\ P(y_k)\prod_i P(f_i|y_k) \end{bmatrix} \xrightarrow{\text{Normalize}} \begin{bmatrix} P(y_1|\, f_1 \dots f_n) \\ P(y_2|\, f_1 \dots f_n) \\ \vdots \\ P(y_k|\, f_1 \dots f_n) \end{bmatrix}$$

Last Time

- *Naïve Bayes* model: $P(Y, F_1, \ldots, F_n) = P(Y) \prod_i P(F_i|Y)$
  - Features and label are random variables
  - Input features $F_1, \ldots, F_n$ are conditionally independent given label Y
  - Parameters θ: probability tables $P(Y), P(F_1|Y), \ldots, P(F_n|Y)$



- Learn parameters by counting:
  - $P(\text{observing } x) = \dfrac{\#\text{ of times } x \text{ occured}}{\text{total }\#\text{ of events}}$

    For example: 

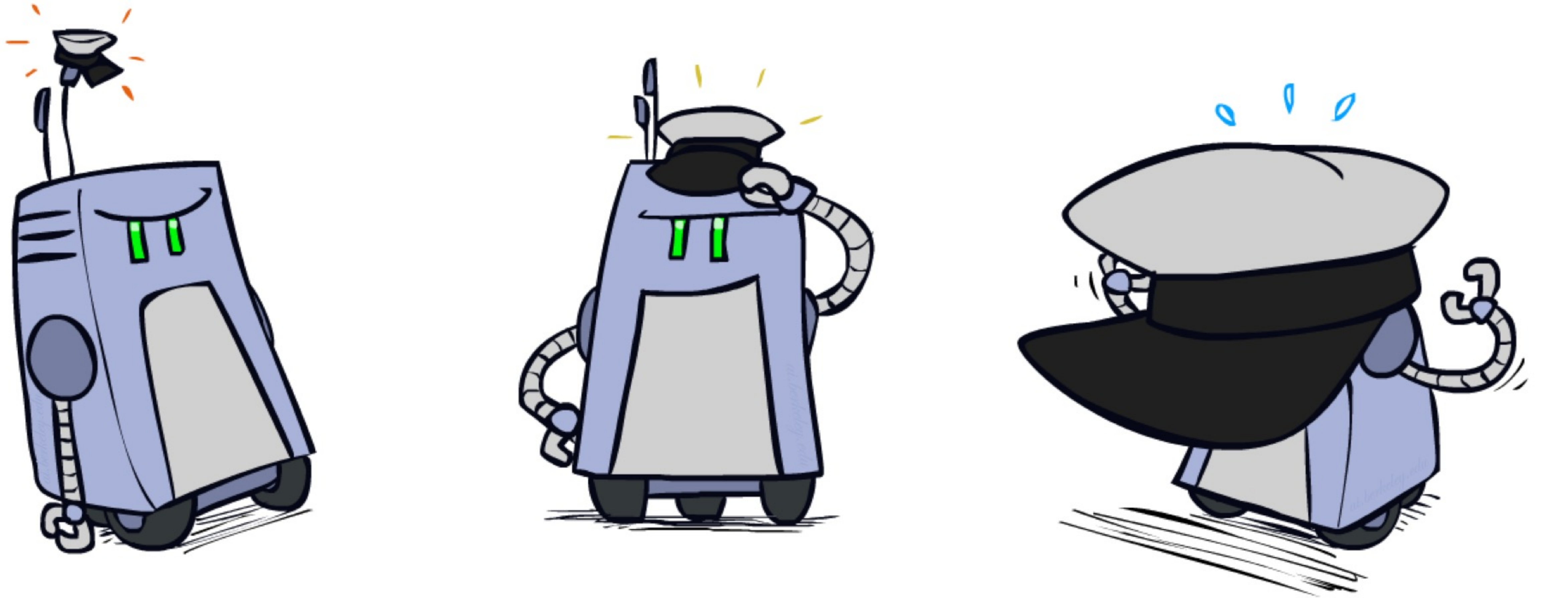
    $P(\text{red}) = \dfrac{2}{3}$

  - Comes from *Maximum Likelihood* estimation: find θ that maximizes $\mathrm{P}(\text{observations}|\theta)$
    - $\theta = \underset{\theta}{\text{argmax}}\, P(\text{observations}|\theta)$
    - Take derivative and set to 0
    - In practice, maximize log P instead because derivatives are easier

  - In general for Naïve Bayes maximum likelihood estimates of probability tables are:

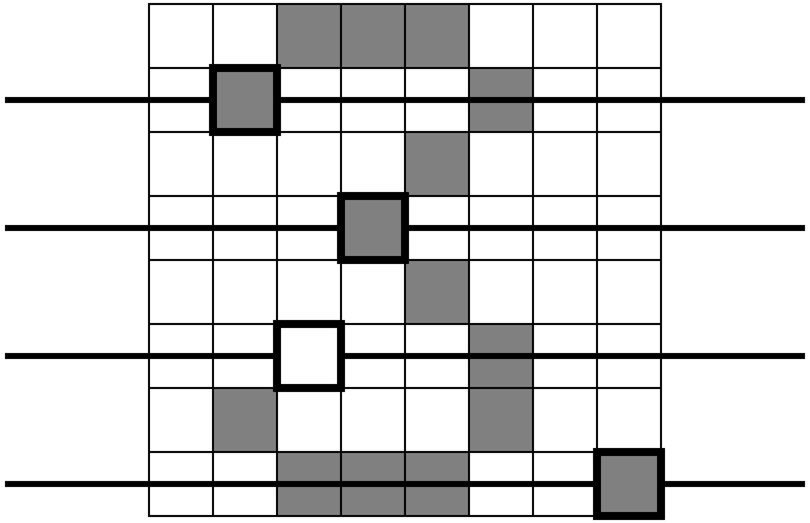
$$P(y) = \frac{\#\text{ of occurences of class } y}{\text{total }\#\text{ of observations}} \qquad P(f \mid y) = \frac{\#\text{ of occurences of feature } f \text{ and class } y}{\text{total }\#\text{ of occurences of class } y}$$

Underfitting and Overfitting

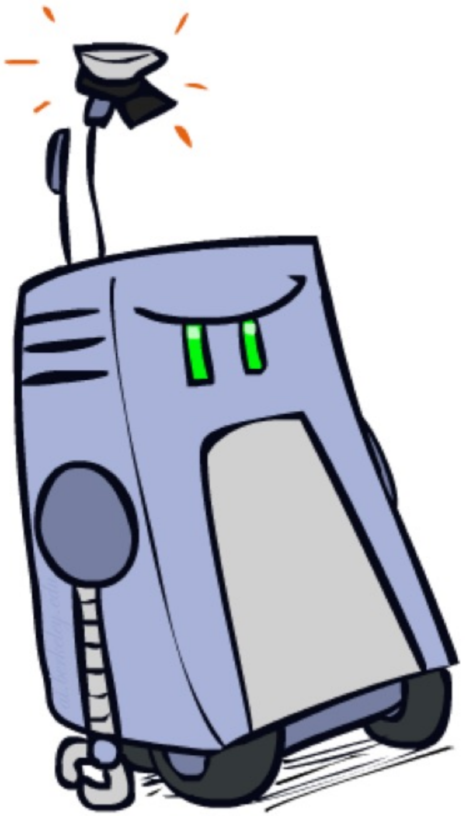


Example: Overfitting

$P(\text{features}, C = 2)$	$P(\text{features}, C = 3)$
$P(C = 2) = 0.1$	$P(C = 3) = 0.1$
$P(\text{on}\vert C = 2) = 0.8$	$P(\text{on}\vert C = 3) = 0.8$
$P(\text{on}\vert C = 2) = 0.1$	$P(\text{on}\vert C = 3) = 0.9$
$P(\text{off}\vert C = 2) = 0.1$	$P(\text{off}\vert C = 3) = 0.7$
$P(\text{on}\vert C = 2) = 0.01$	$P(\text{on}\vert C = 3) = 0.0$



2 wins!!



Example: Overfitting

- *relative* probabilities (odds ratios):

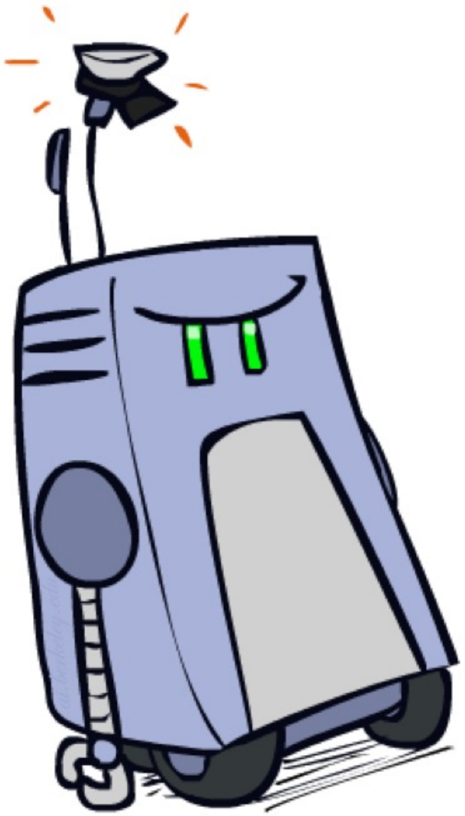
$$\frac{P(W|\text{ham})}{P(W|\text{spam})}$$

```
south-west  : inf
nation      : inf
morally     : inf
nicely      : inf
extent      : inf
seriously   : inf
...
```

$$\frac{P(W|\text{spam})}{P(W|\text{ham})}$$

```
screens    : inf
minute     : inf
guaranteed : inf
$205.00    : inf
delivery   : inf
signature  : inf
...
```

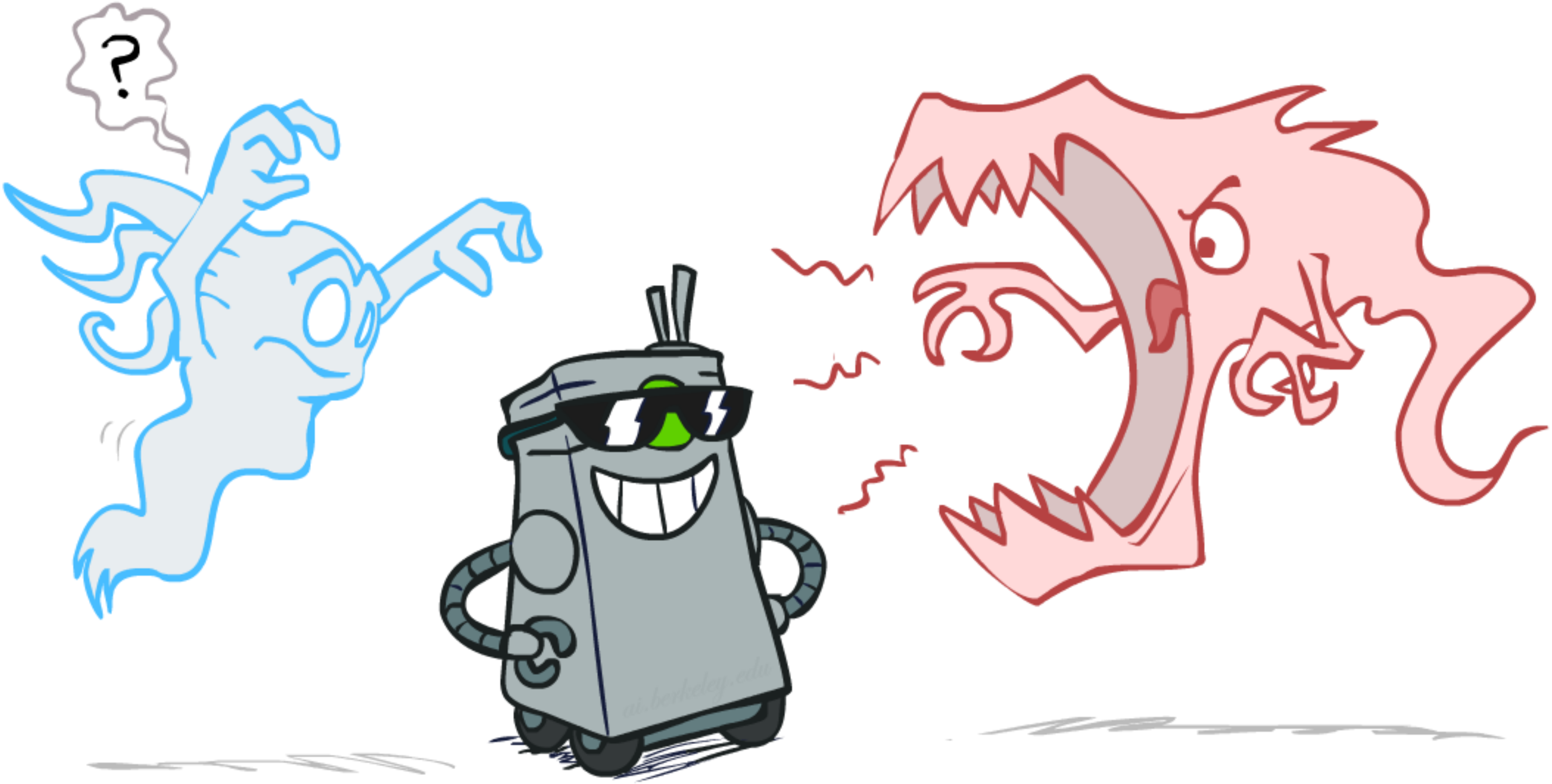
What went wrong here?



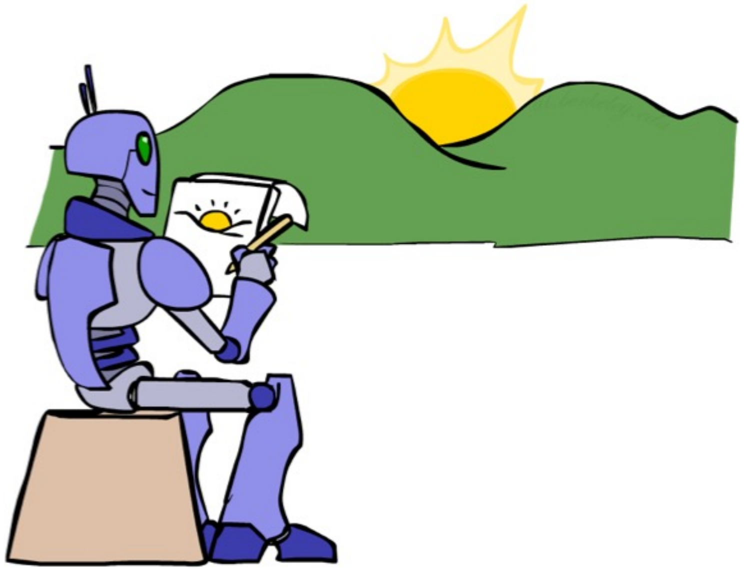
Generalization and Overfitting

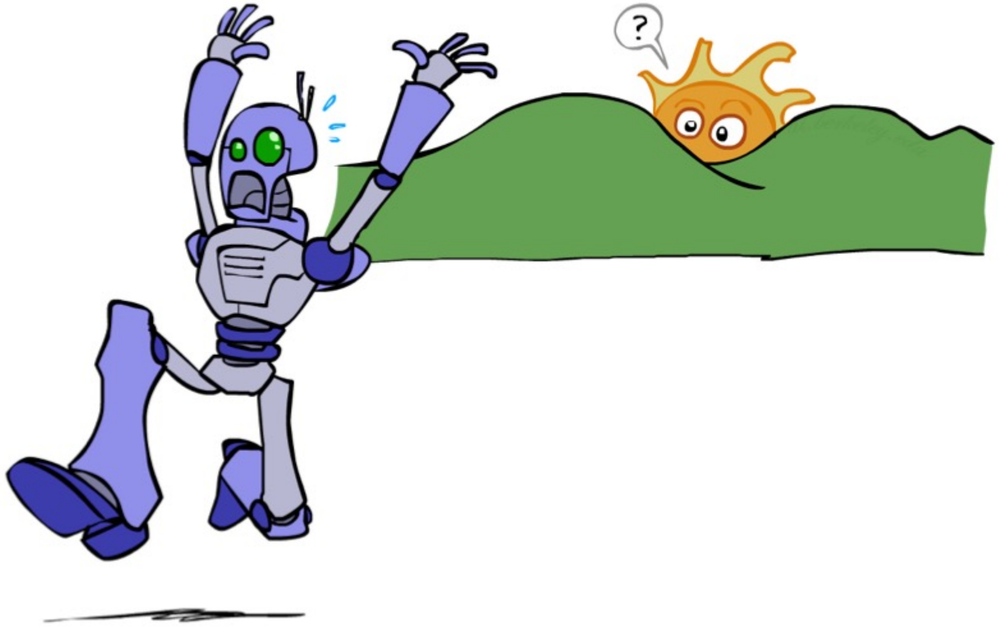
- Relative frequency parameters will overfit the training data!
  - Just because we never saw a 3 with pixel (15,15) on during training doesn't mean we won't see it at test time
  - Unlikely that every occurrence of "minute" is 100% spam
  - Unlikely that every occurrence of "seriously" is 100% ham
  - What about all the words that don't occur in the training set at all?
  - In general, we can't go around giving unseen events zero probability
- As an extreme case, imagine using the entire email as the only feature
  - Would get the training data perfect (if deterministic labeling)
  - Wouldn't *generalize* at all
  - Just making the bag-of-words assumption gives us some generalization, but isn't enough
- To generalize better: we need to smooth or regularize the estimates

Smoothing



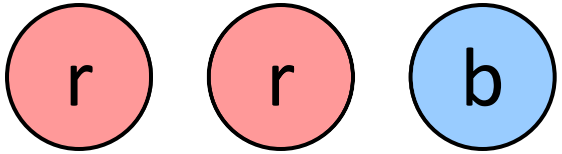
Unseen Events





Laplace Smoothing

- Laplace's estimate:
  - Pretend you saw every outcome once more than you actually did



$$P_{LAP}(x) = \frac{c(x)+1}{\sum_x [c(x)+1]}$$

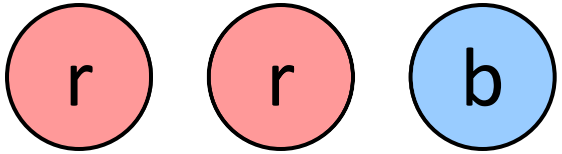
$$= \frac{c(x)+1}{N+|X|}$$

$$P_{ML}(X) =$$

$$P_{LAP}(X) =$$

Laplace Smoothing

- Laplace's estimate (extended):
  - Pretend you saw every outcome k extra times



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

  - What's Laplace with k = 0?
  - k is the strength of the prior

$$P_{LAP,0}(X) =$$

$$P_{LAP,1}(X) =$$

$$P_{LAP,100}(X) =$$

Real NB: Smoothing

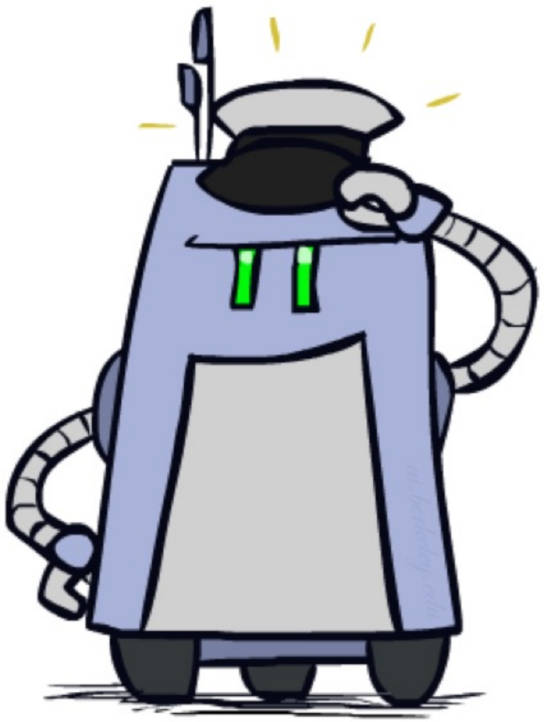
- For real classification problems, smoothing is critical
- New odds ratios:

$$\frac{P(W|\text{ham})}{P(W|\text{spam})}$$

helvetica	:	11.4
seems	:	10.8
group	:	10.2
ago	:	8.4
areas	:	8.3
...		

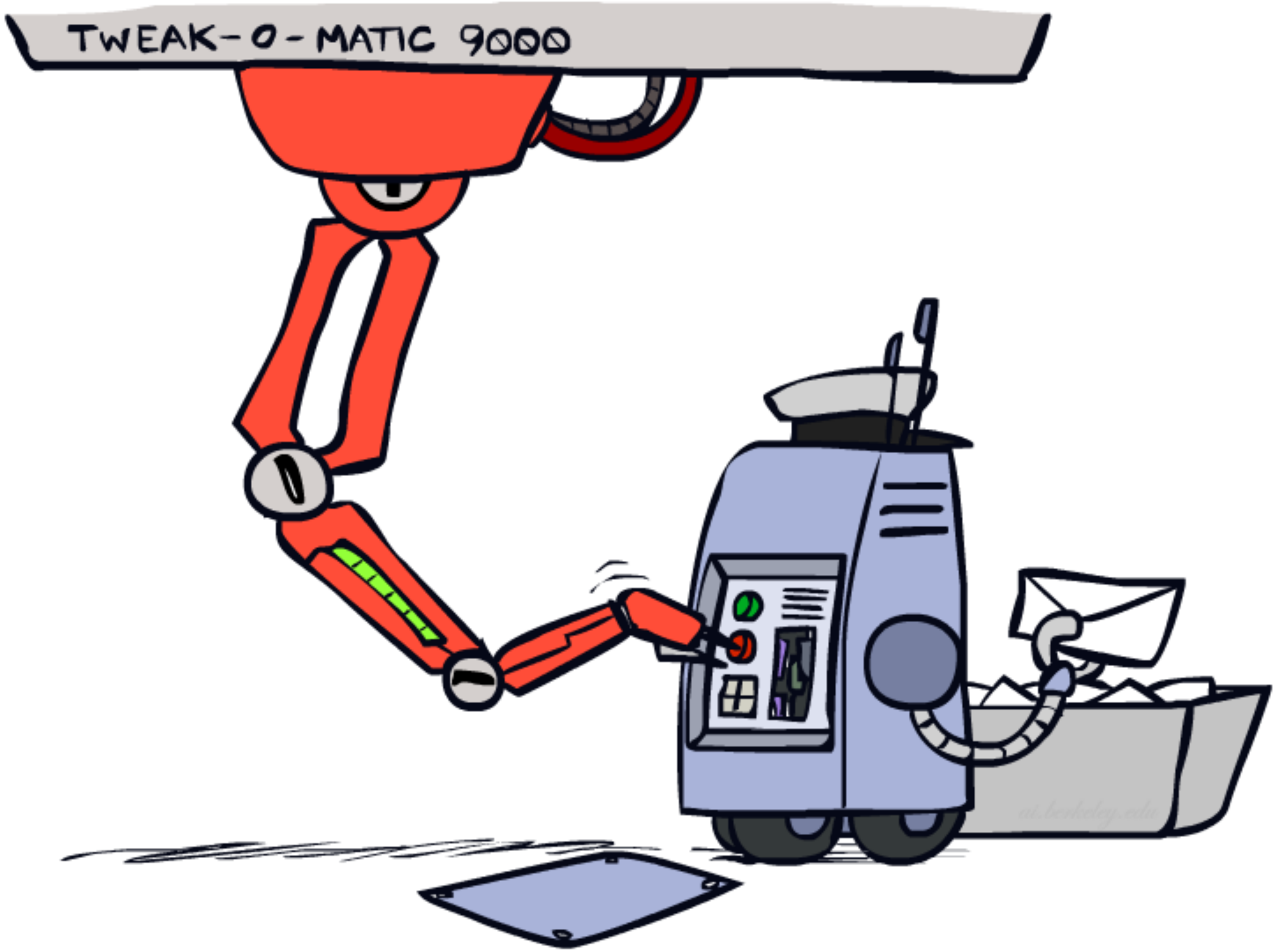
$$\frac{P(W|\text{spam})}{P(W|\text{ham})}$$

verdana	:	28.8
Credit	:	28.4
ORDER	:	27.2
<FONT>	:	26.9
money	:	26.5
...		



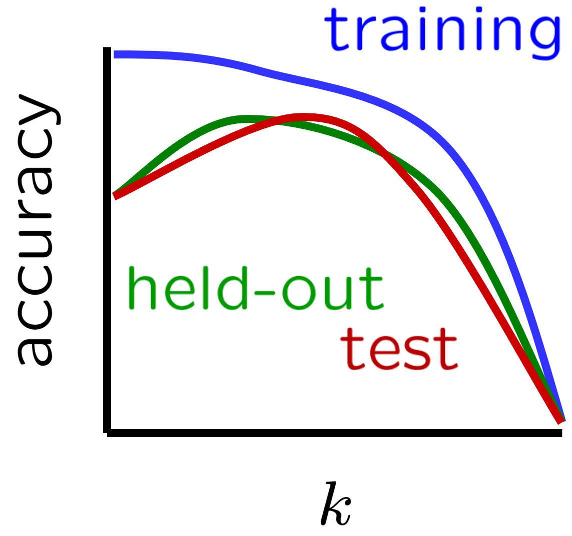
Do these make more sense?

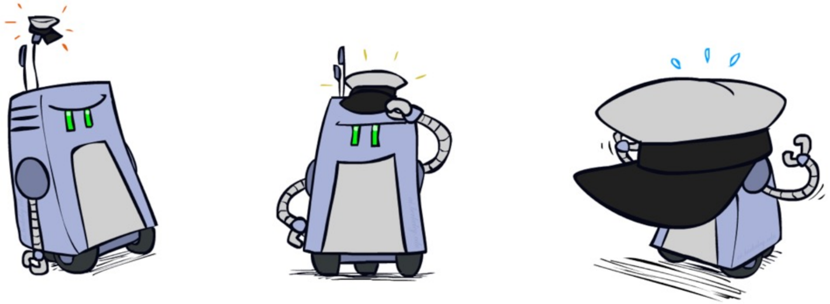
Tuning



Tuning on Held-Out Data

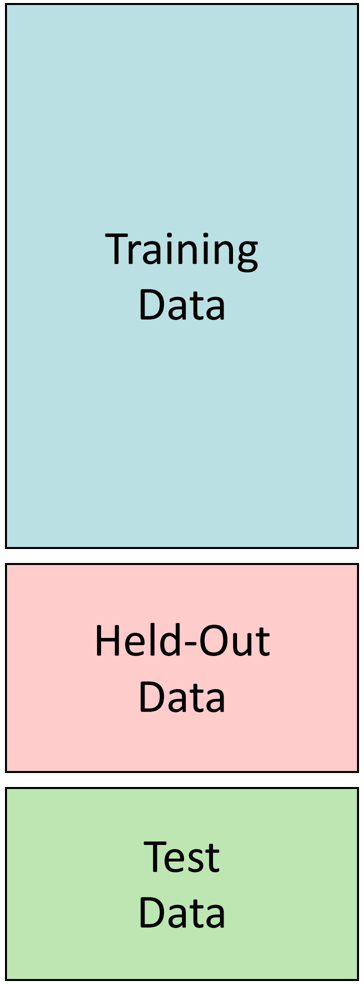
- Now we've got two kinds of unknowns
  - Parameters: the probabilities P(X|Y), P(Y)
  - Hyperparameters: e.g. the amount of smoothing *k*
- What should we learn where?
  - Learn parameters from training data
  - Tune hyperparameters on different data
    - Why?
  - For each value of the hyperparameters, train and test on the held-out data
  - Choose the best value and do a final test on the test data

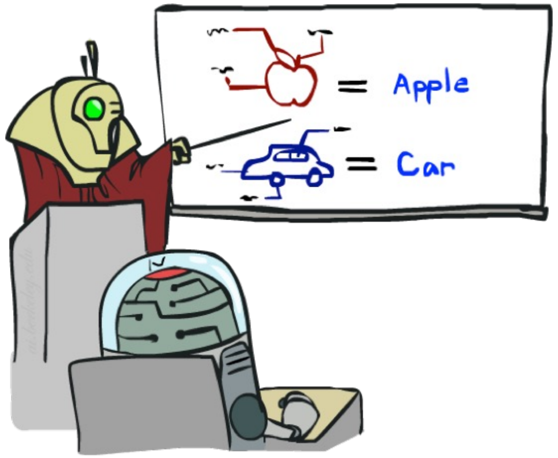


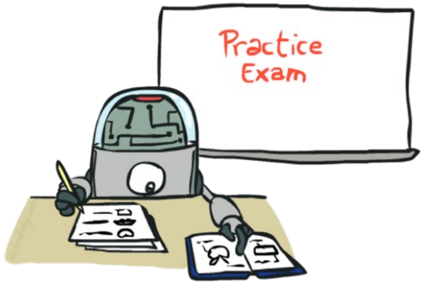


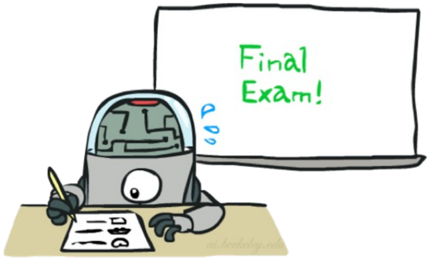
Important Concepts

- Data: labeled instances, e.g. emails marked spam/ham
  - Training set
  - Held out set
  - Test set
- Features: attribute-value pairs which characterize each input
- Experimentation cycle
  - Learn parameters (e.g. model probabilities) on training set
  - (Tune hyperparameters on held-out set)
  - Compute accuracy on test set
  - Very important: never “peek” at the test set!
- Evaluation
  - Accuracy: fraction of instances predicted correctly
- Overfitting and generalization
  - Want a classifier which does well on *test* data
  - <u>Overfitting</u>: fitting the training data very closely, but not generalizing well
  - <u>Underfitting</u>: fits the training set poorly





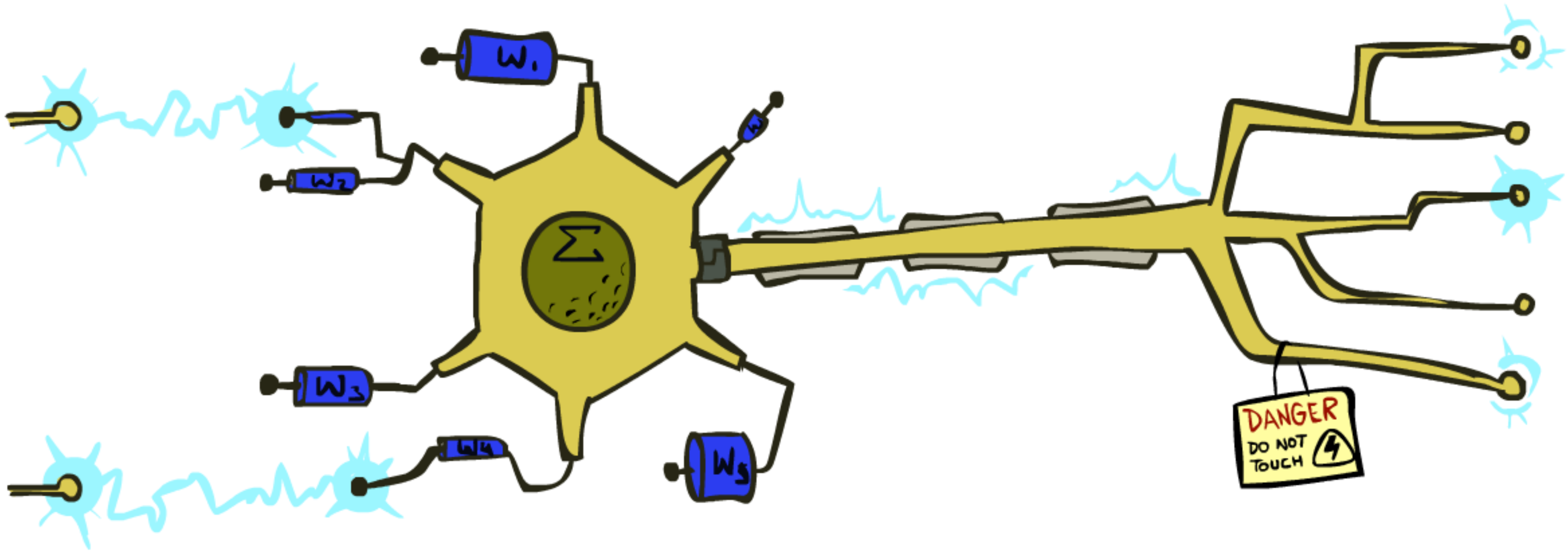




Practical Tip: Baselines

- 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

Perceptrons



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
...
```

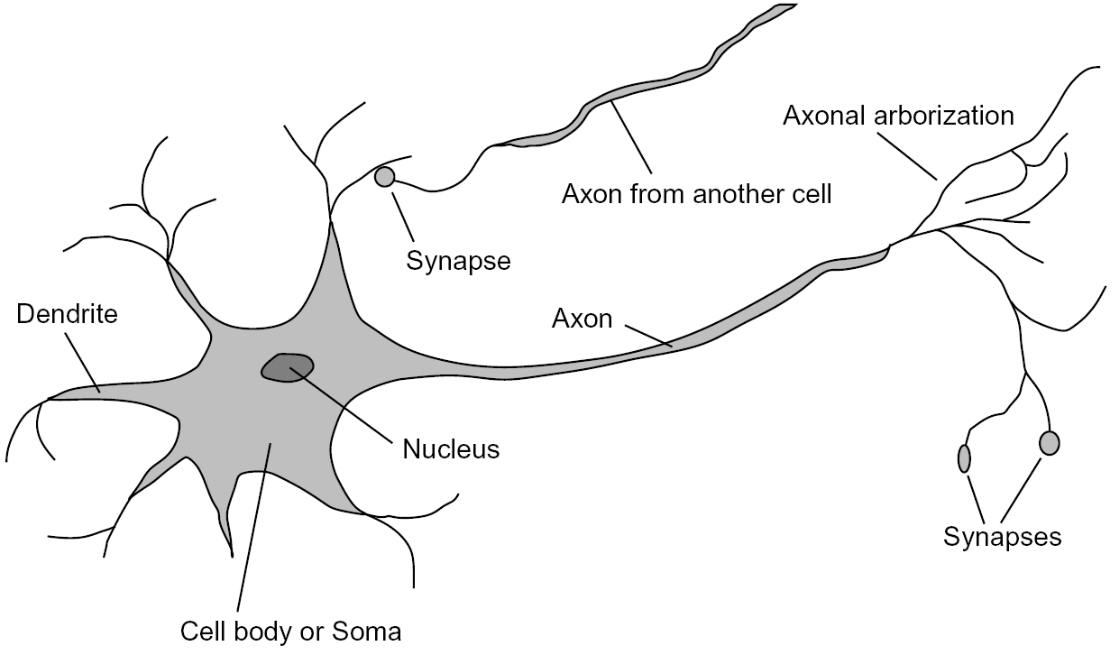
SPAM
or
HAM

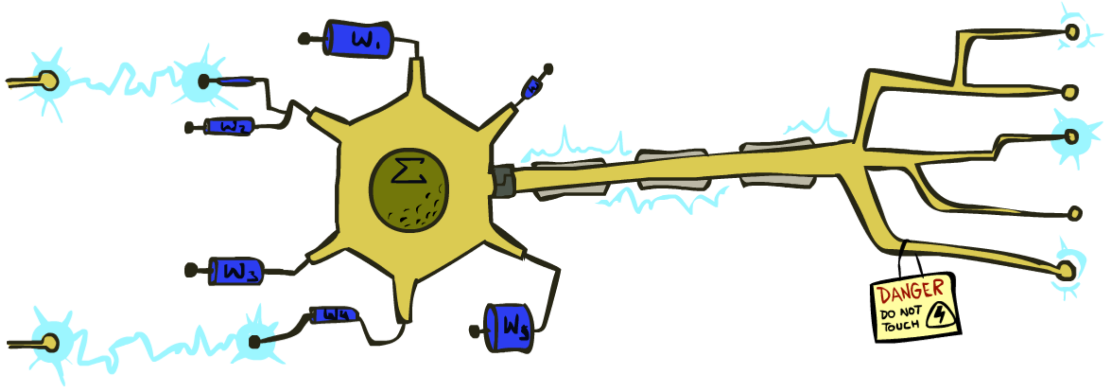
```
PIXEL-7,12  : 1
PIXEL-7,13  : 0
...
NUM_LOOPS   : 1
...
```

“2”

Some (Simplified) Biology

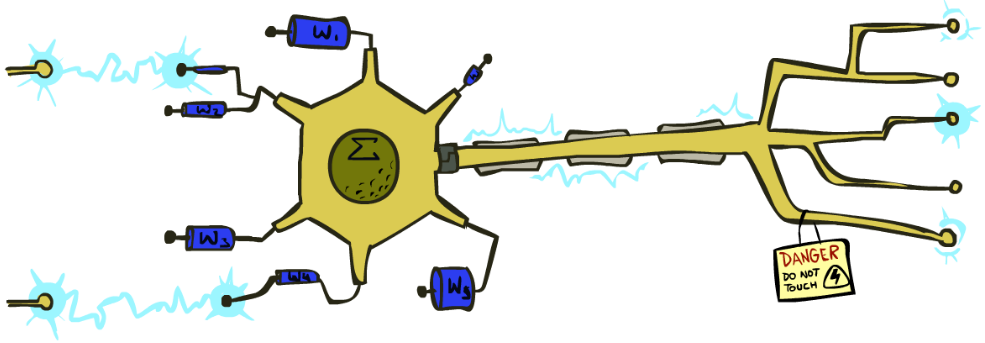
- Very loose inspiration: human neurons





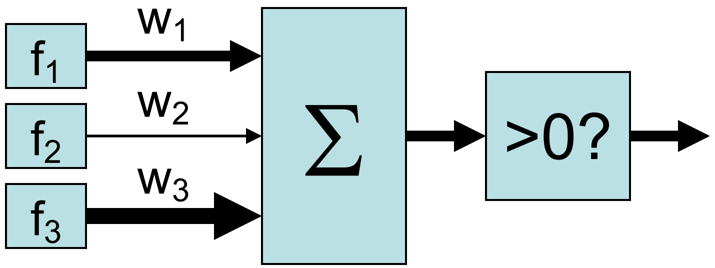
Linear Classifiers

- 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

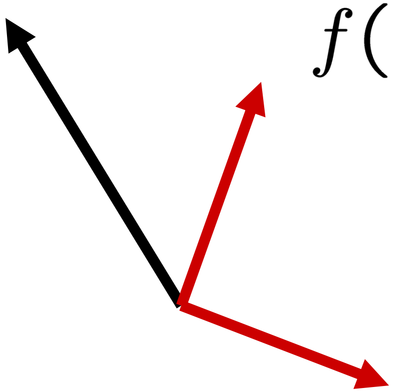


Weights

- 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
...
```

w



$f(x_1)$

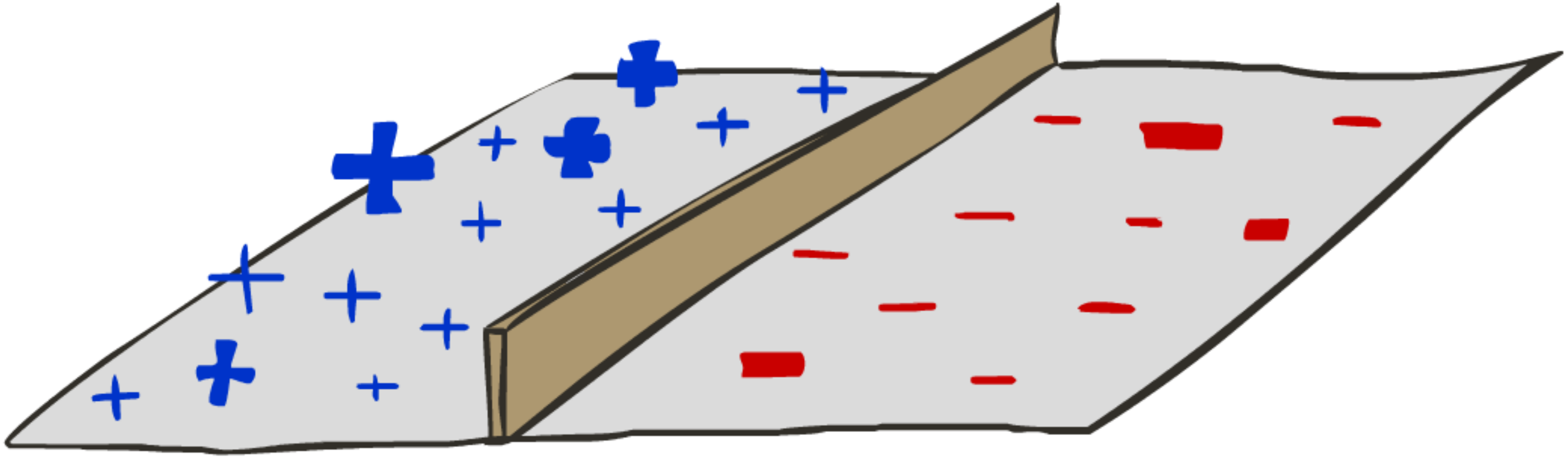
```
# free        :  2
YOUR_NAME     :  0
MISSPELLED    :  2
FROM_FRIEND   :  0
...
```

$f(x_2)$

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

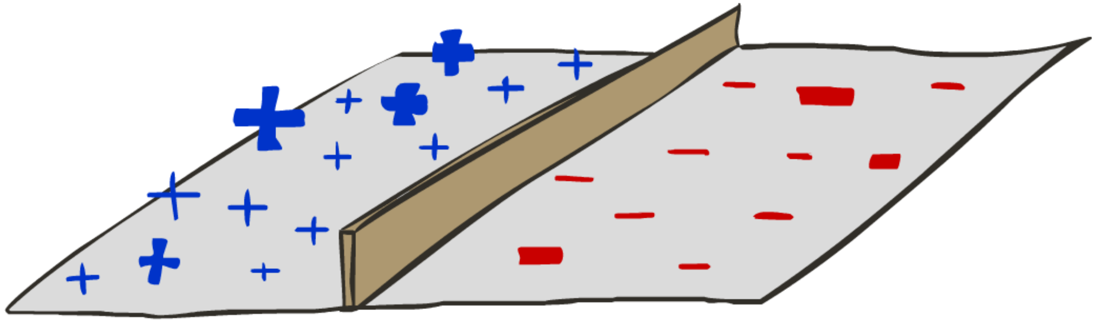
Dot product $w \cdot f$ *positive means the positive class*

Decision Rules



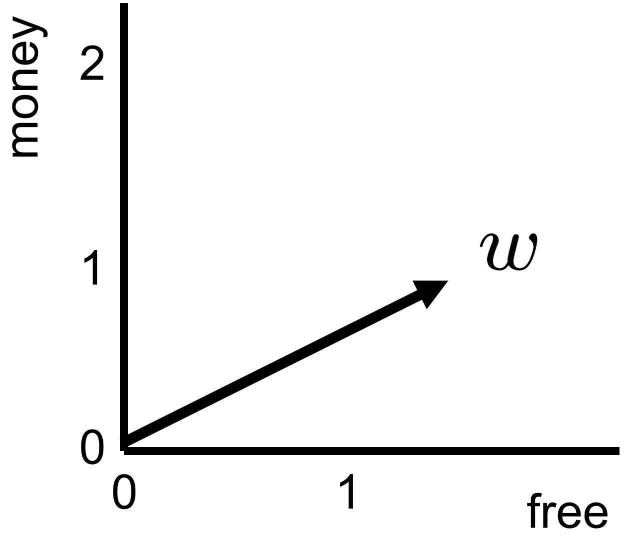
Binary Decision Rule

- 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



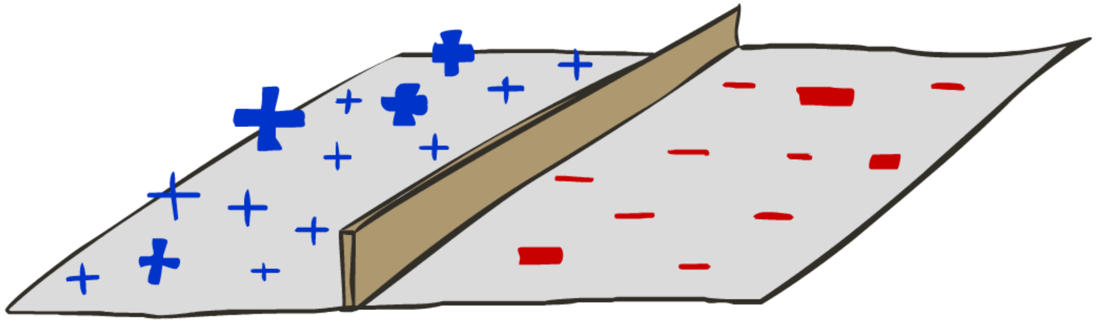
$$w$$

```
free  :  4
money :  2
```



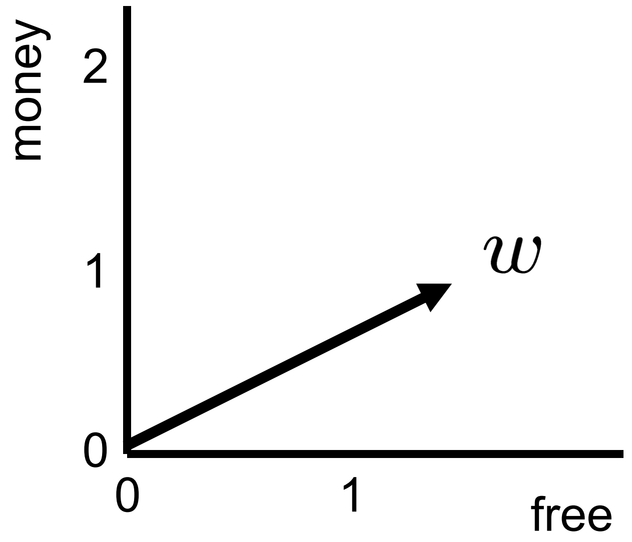
Binary Decision Rule

- 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



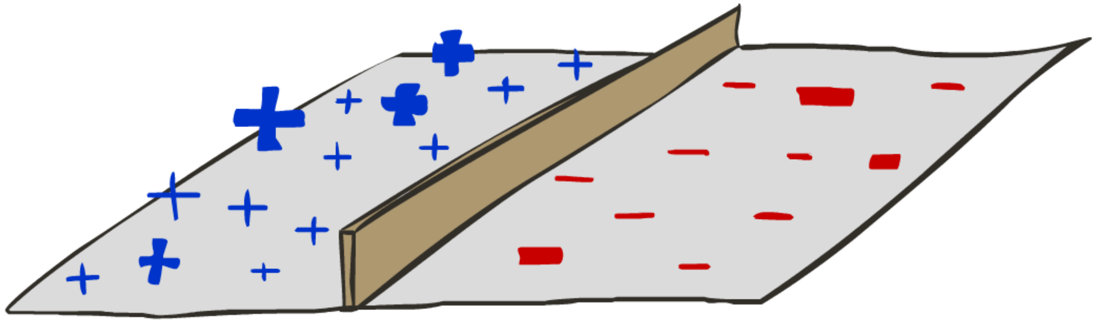
w

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



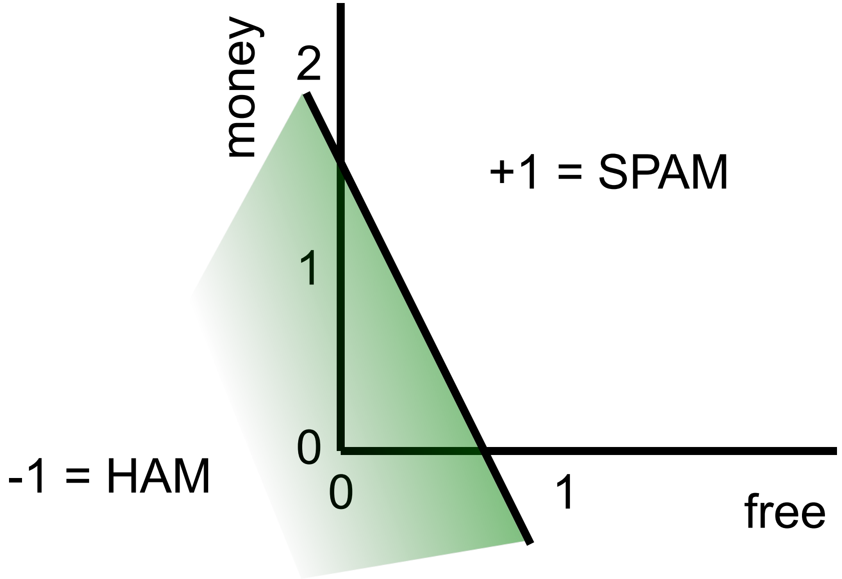
Binary Decision Rule

- 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
...
```

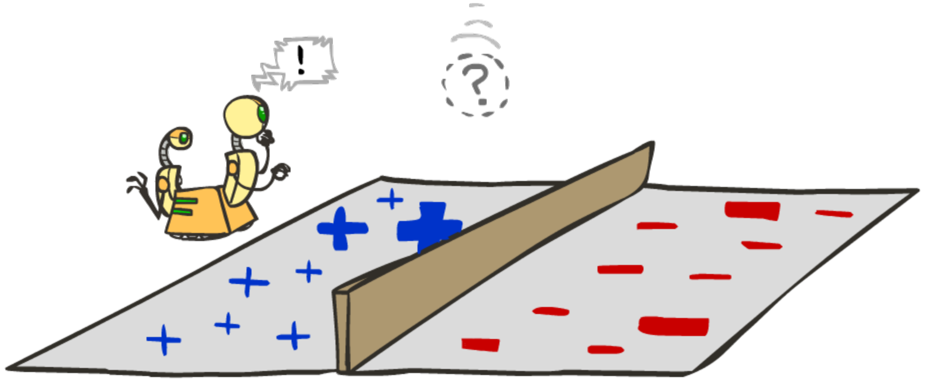


$f \cdot w = 0$

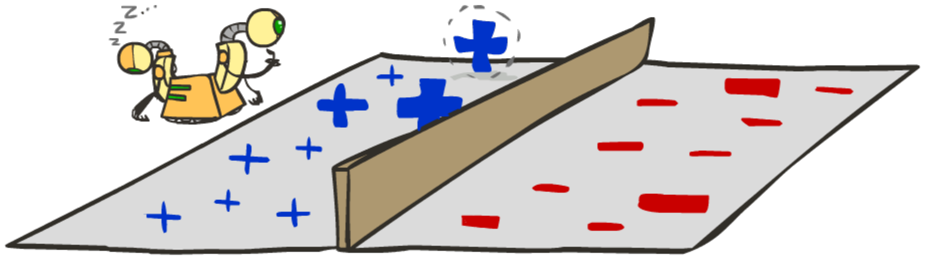
Weight Updates

Learning: Binary Perceptron

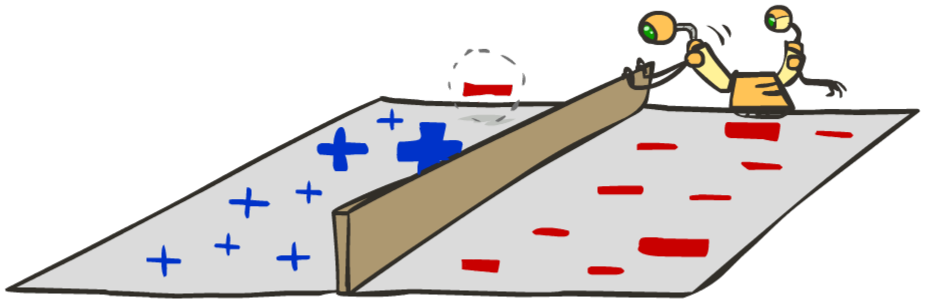
- Start with weights w = 0
- For each training instance f(x), y*:
  - Classify with current weights



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



  - If wrong: adjust the weight vector



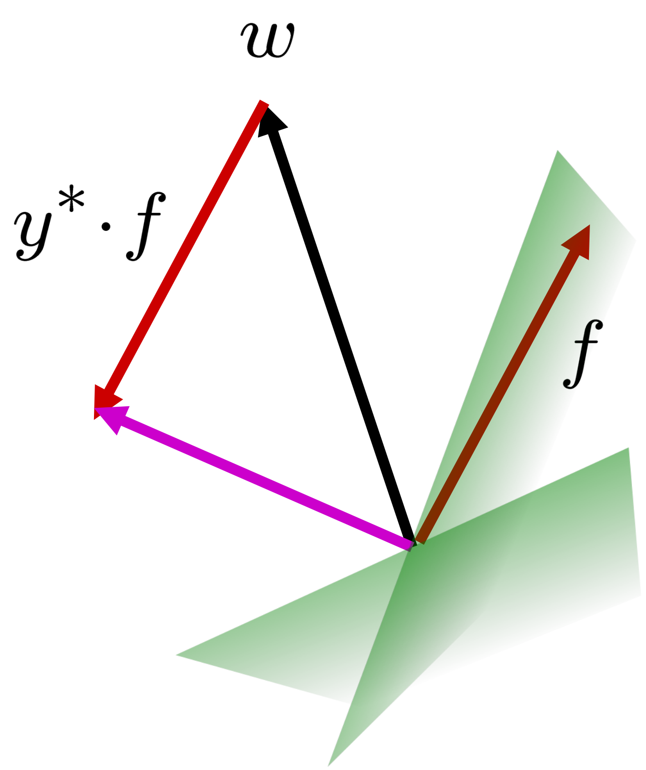
Learning: Binary Perceptron

- Start with weights w = 0
- For each training instance f(x), y*:
  - 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.

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



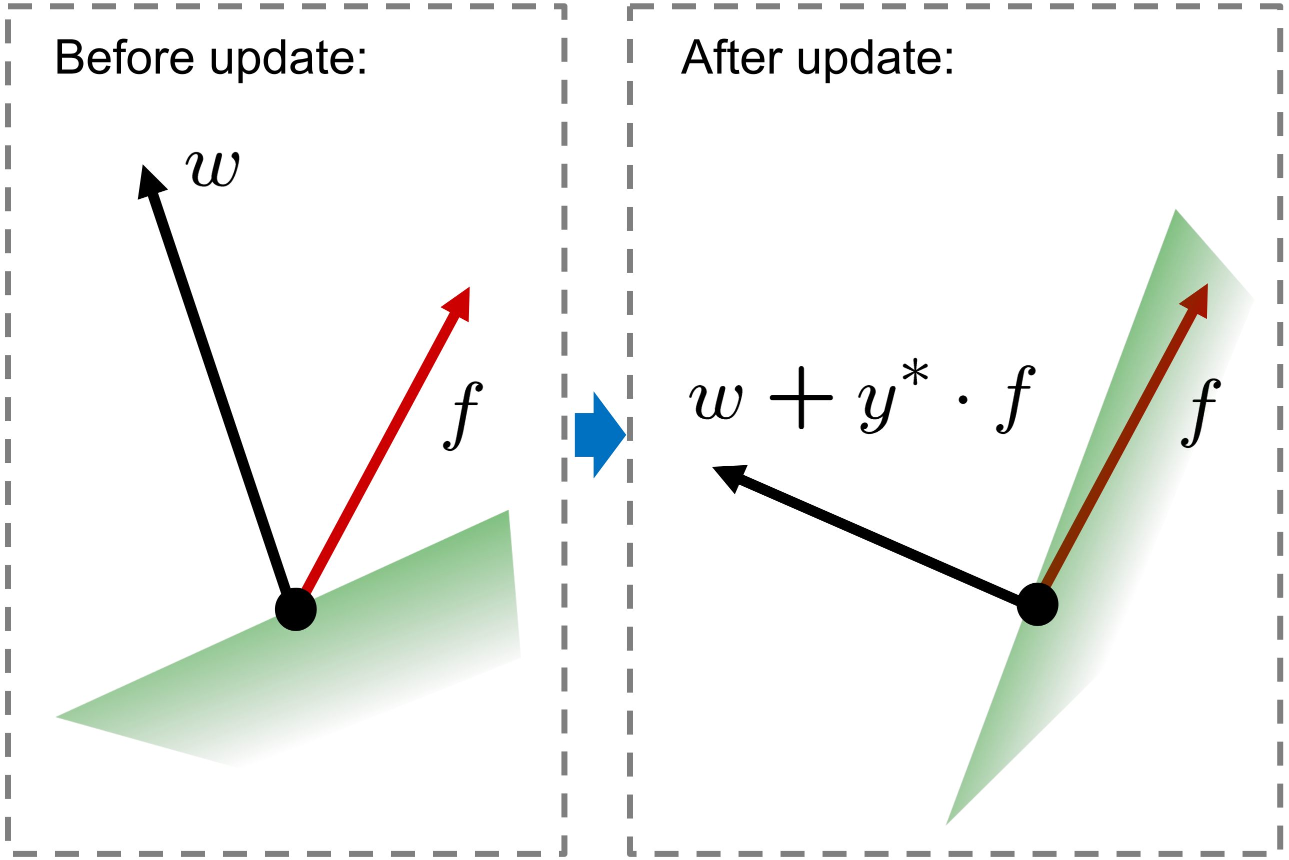
Learning: Binary Perceptron

- Start with weights w = 0
- For each training instance f(x), y*:
  - 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.

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



$w \cdot f$

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

Example: Perceptron

Iteration 0: x: "*win the vote*" f(x): [1 1 0 1 1] y*: -1

Iteration 1: x: "*win the election*" f(x): [1 1 0 0 1] y*: -1

Iteration 2: x: "*win the game*" f(x): [1 1 1 0 1] y*: +1

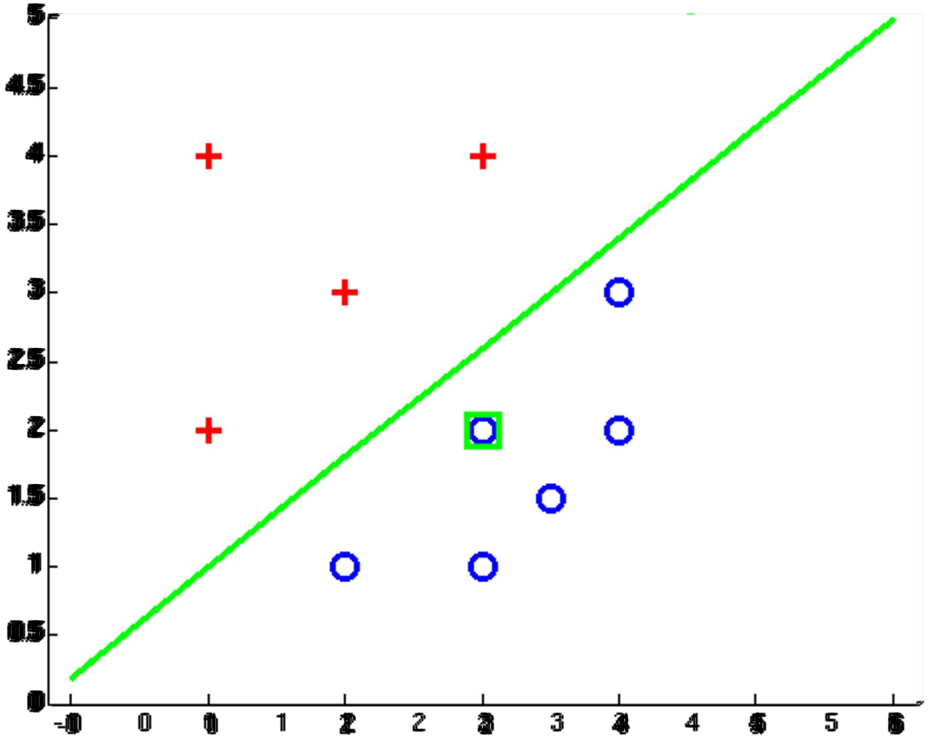
Iteration 3: x: "*win the game*" f(x): [1 1 1 0 1] y*: +1

w

BIAS	1	0	0	1
win	0	-1	-1	0
game	0	0	0	1
vote	0	-1	-1	-1
the	0	-1	-1	0
$w \cdot f(x)$:	1	-2	-2	2

Example: Perceptron

- Separable Case



Multiclass Decision Rule

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

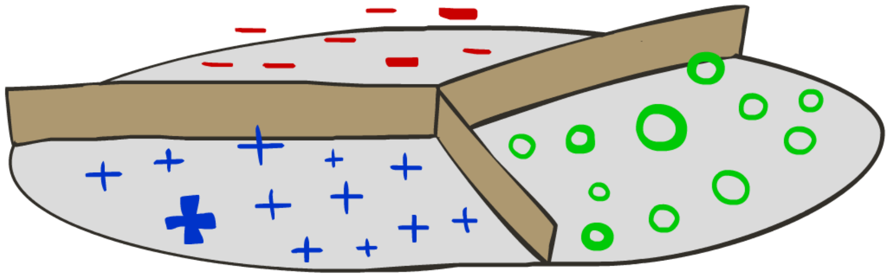
$$w_y$$

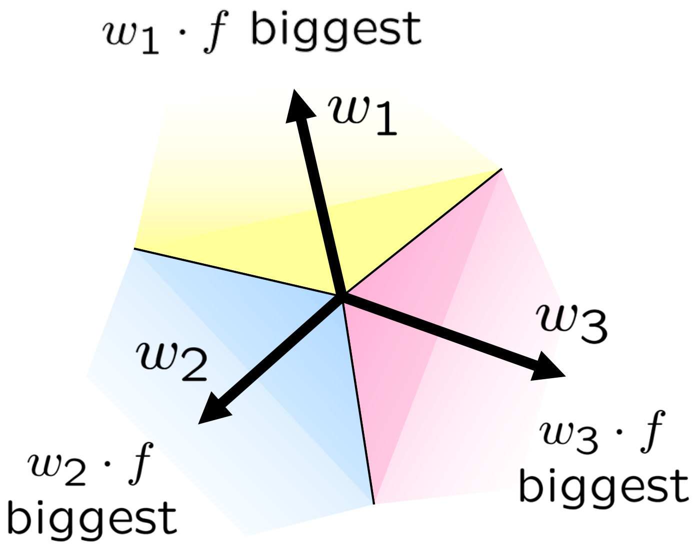
  - Score (activation) of a class y:

$$w_y \cdot f(x)$$

  - 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

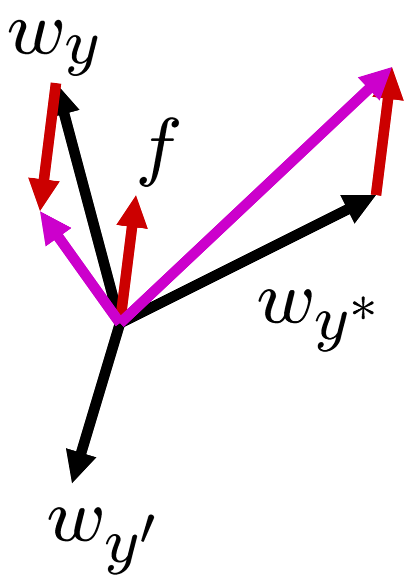
- Start with all weights = 0
- Pick up training examples f(x), y* one by one
- Predict with current weights

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

- 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

Iteration 0: x: "*win the vote*" f(x): [1 1 0 1 1] y*: politics

Iteration 1: x: "*win the election*" f(x): [1 1 0 0 1] y*: politics

Iteration 2: x: "*win the game*" f(x): [1 1 1 0 1] y*: sports

w_{SPORTS}

BIAS	1	0	0	1
win	0	-1	-1	0
game	0	0	0	1
vote	0	-1	-1	-1
the	0	-1	-1	0
$w \cdot f(x)$:	1	-2	-2	

$w_{POLITICS}$

BIAS	0	1	1	0
win	0	1	1	0
game	0	0	0	-1
vote	0	1	1	1
the	0	1	1	0
$w \cdot f(x)$:	0	3	3	

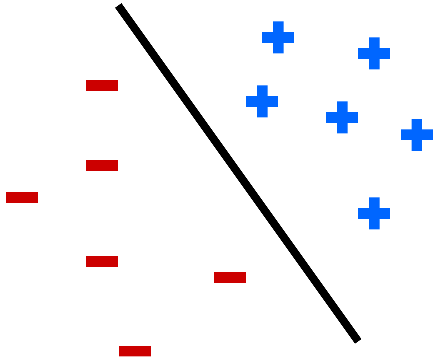
w_{TECH}

BIAS	0	0	0	0
win	0	0	0	0
game	0	0	0	0
vote	0	0	0	0
the	0	0	0	0
$w \cdot f(x)$:	0	0	0	

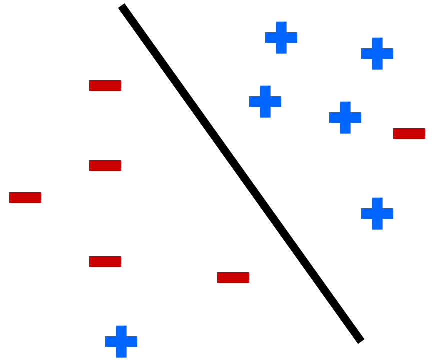
Properties of Perceptrons

- Separability: true if some parameters get the training set perfectly correct
- Convergence: if the training is separable, perceptron will eventually converge (binary case)
- Mistake Bound: the maximum number of mistakes (binary case) related to the *margin* or degree of separability

Separable

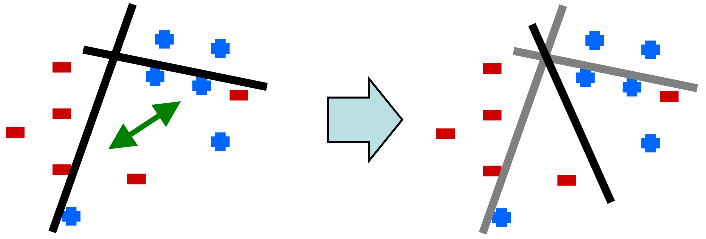


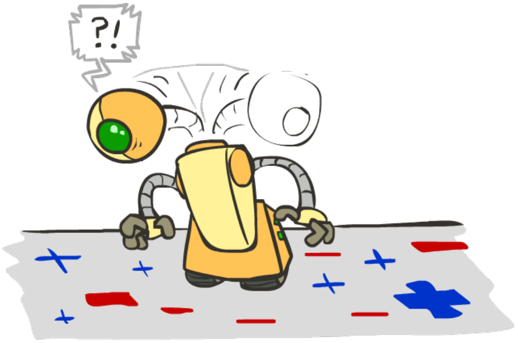
Non-Separable



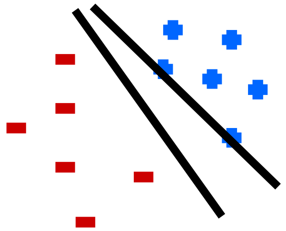
Problems with the Perceptron

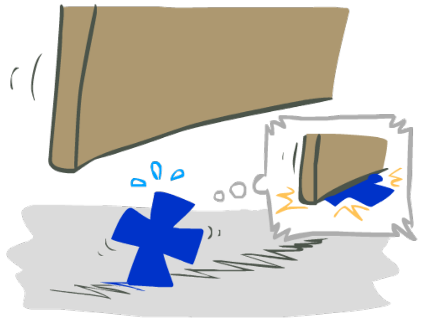
- Noise: if the data isn't separable, weights might thrash
  - Averaging weight vectors over time can help (averaged perceptron)



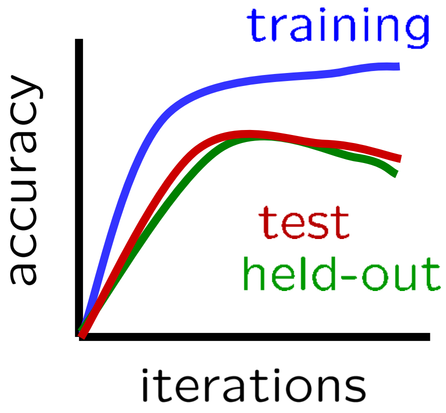


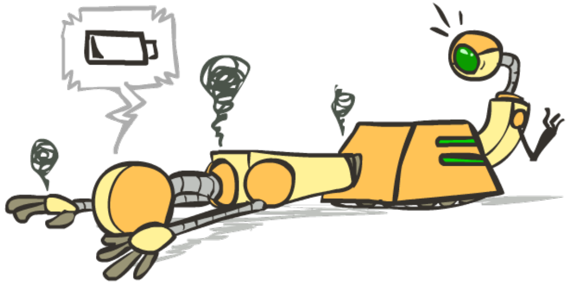
- Mediocre generalization: finds a "barely" separating solution





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





Next Lecture: Improving Perceptron & Optimization