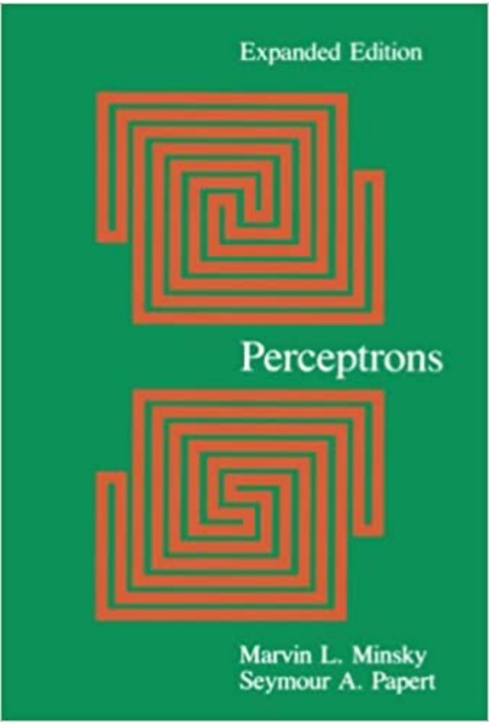
Discriminative Classification



LING83800: METHODS IN COMPUTATIONAL LINGUISTICS II

May 6, 2024

Spencer Caplan

Today

1. Logistic Regression

2. Cross-validation

3. Perceptrons and Neural Nets

Naïve Bayes Recap

- Bag of words (order independent)

- Feature are assumed independent given a class

$$P(x_1, \ldots, x_n|c) = P(x_1|c) \ldots P(x_n|c)$$

Q: Is this really true?

Problems with assuming conditional independence

- Correlated features → double counting evidence
  - Since parameters are estimated independently

- Example: Predicting test scores
  - Previous test score
  - Height
  - Age
  - Etc.

- **This hurts classifier accuracy and calibration**

Logistic Regression

- Doesn't assume features are independent

- Correlated features don't "double count"

Generative vs. Discriminative Models

Naive Bayes is a **generative** classifier

Logistic regression is a is a **discriminiative classifier**

Generative vs. Discriminative Models

A **generative model** uses the likelihood term, which expresses how to generate the features of a document *if we knew it was of class c*.

Generative models (like HMMs or Naïve Bayes) make use of the likelihood term

A **discriminative model** attempts to directly compute P(c|d).

$$\hat{c} = \underset{c \in C}{\operatorname{argmax}} \; \overbrace{P(d|c)}^{\text{likelihood}} \; \overbrace{P(c)}^{\text{prior}}$$

It may learn to assign a **high weight** to document features that directly improve its **ability to discriminate between classes**

Unlike the generative model, good parameter estimates for a discriminative model don't help it generate an example of one of the classes.

Discriminative models (like logistic regression) attempt to directly compute P(c|d)

Components of Classifiers

1. Feature representation of the input

2. Classification function

3. Objective function

4. Algorithm to optimize the objective function

Components of Classifiers

1. Feature representation of the input
2. Classification function
   - Used to compute our estimate of $p(y|x)$
   - For logistic regression this is the **sigmoid**
3. Objective function
   - Used during learning to minimize error on the training example.
   - For logistic regression this is **cross-entropy loss**
4. Algorithm to optimize the objective function
   - Here that's **stochastic gradient descent**

Sentiment Classifier

Input: "Spiraling away from narrative control as its first three episodes unreel, this series, about a post-apocalyptic future in which nearly everyone is blind, wastes the time of Jason Momoa and Alfre Woodard, among others, on a story that starts from a position of fun, giddy strangeness and drags itself forward at a lugubrious pace."

Output: positive (1) or **negative (0)**

Sentiment Classifier

For sentiment classification, consider an input observation x, represented by a vector of **features** $[x_1, x_2, \ldots, x_n]$. The classifier output y can be 1 (positive sentiment) or 0 (negative sentiment). We want to estimate $\boldsymbol{P(y=1|x)}$

Logistic regression solves this task by learning, from a training set, a vector of **weights** and a **bias term**

$$z = \sum_i w_i x_i + b$$

We can also write this as a dot product:

$$z = w \cdot x + b$$

30 second linear algebra

$$\text{dot-product}(\vec{v}, \vec{w}) = \vec{v} \cdot \vec{w} = \sum_{i=1}^{N} v_i w_i = v_1 w_1 + v_2 w_2 + \ldots + v_N w_N$$

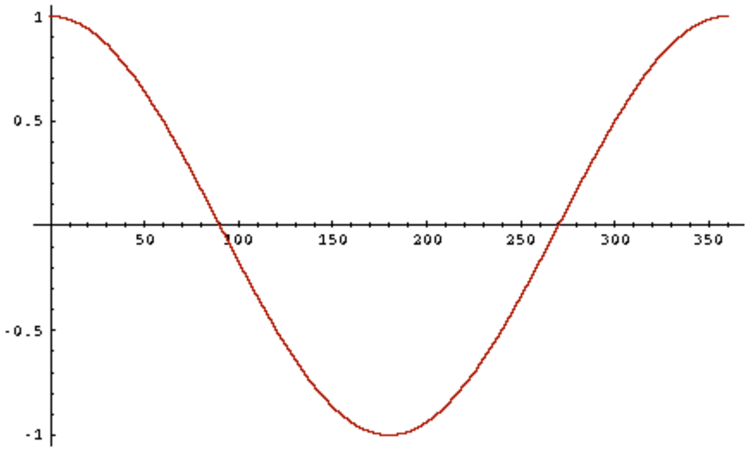
$$\text{vector length} \quad |\vec{v}| = \sqrt{\sum_{i=1}^{N} v_i^2}$$

30 second linear algebra: Cosine similarity

$$\text{cosine}(\vec{v},\vec{w}) = \frac{\vec{v}\cdot\vec{w}}{|\vec{v}||\vec{w}|} = \frac{\sum_{i=1}^{N} v_i w_i}{\sqrt{\sum_{i=1}^{N} v_i^2}\sqrt{\sum_{i=1}^{N} w_i^2}}$$

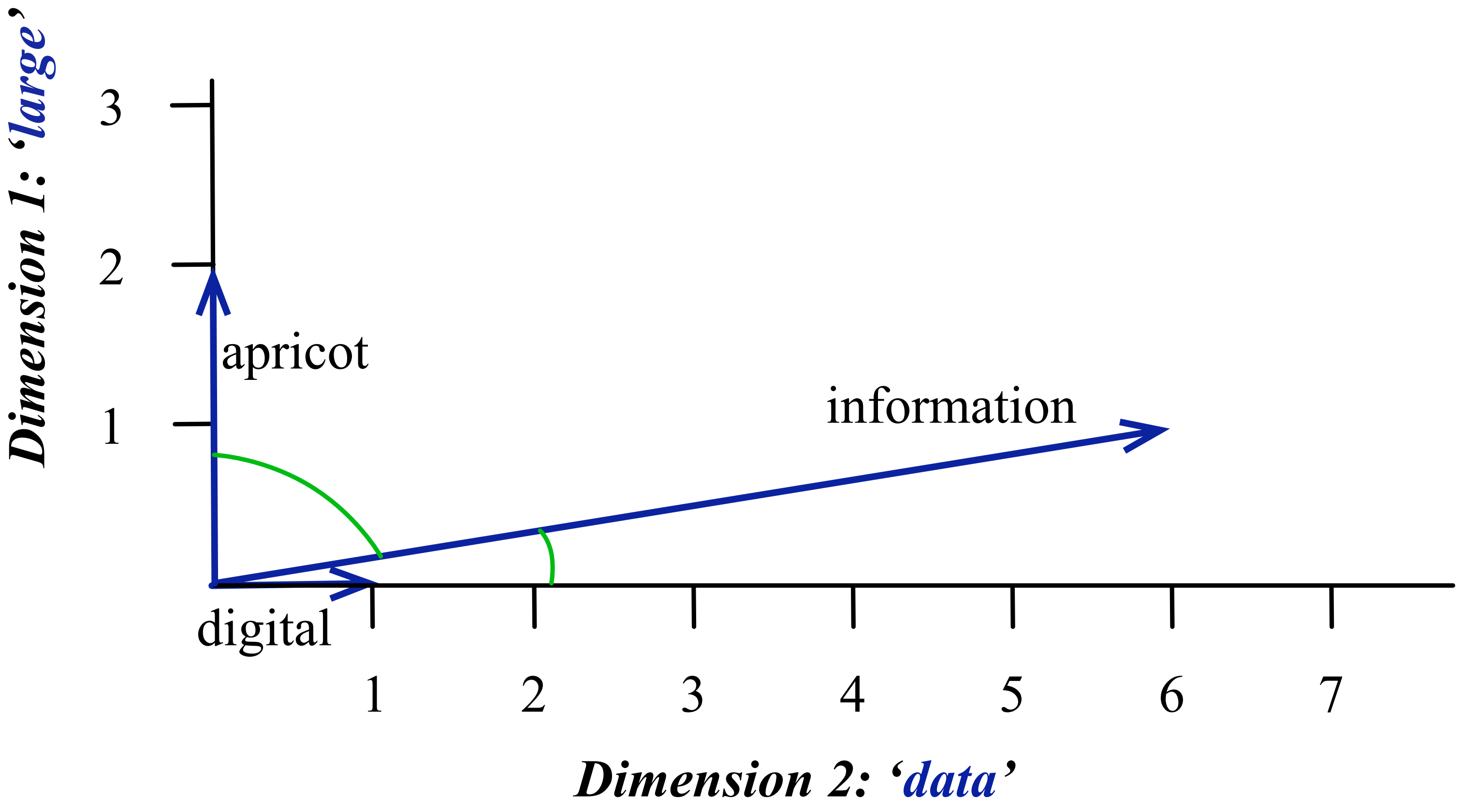
Cosine similarity

- -1: vectors point in opposite directions
- +1: vectors point in same directions
- 0: vectors are orthogonal



- Frequency is non-negative, so cosine range 0-1

Cosine similarity is just the difference in angle



Dot Product

$$a \cdot b = \sum_{i=1}^{n} a_i b_i$$

Sentiment Classifier

For sentiment classification, consider an input observation x, represented by a vector of **features** $[x_1, x_2, \ldots, x_n]$. The classifier output y can be 1 (positive sentiment) or 0 (negative sentiment). We want to estimate ***P*(*y*=1|*x*)**

Logistic regression solves this task by learning, from a training set, a vector of **weights** and a **bias term**

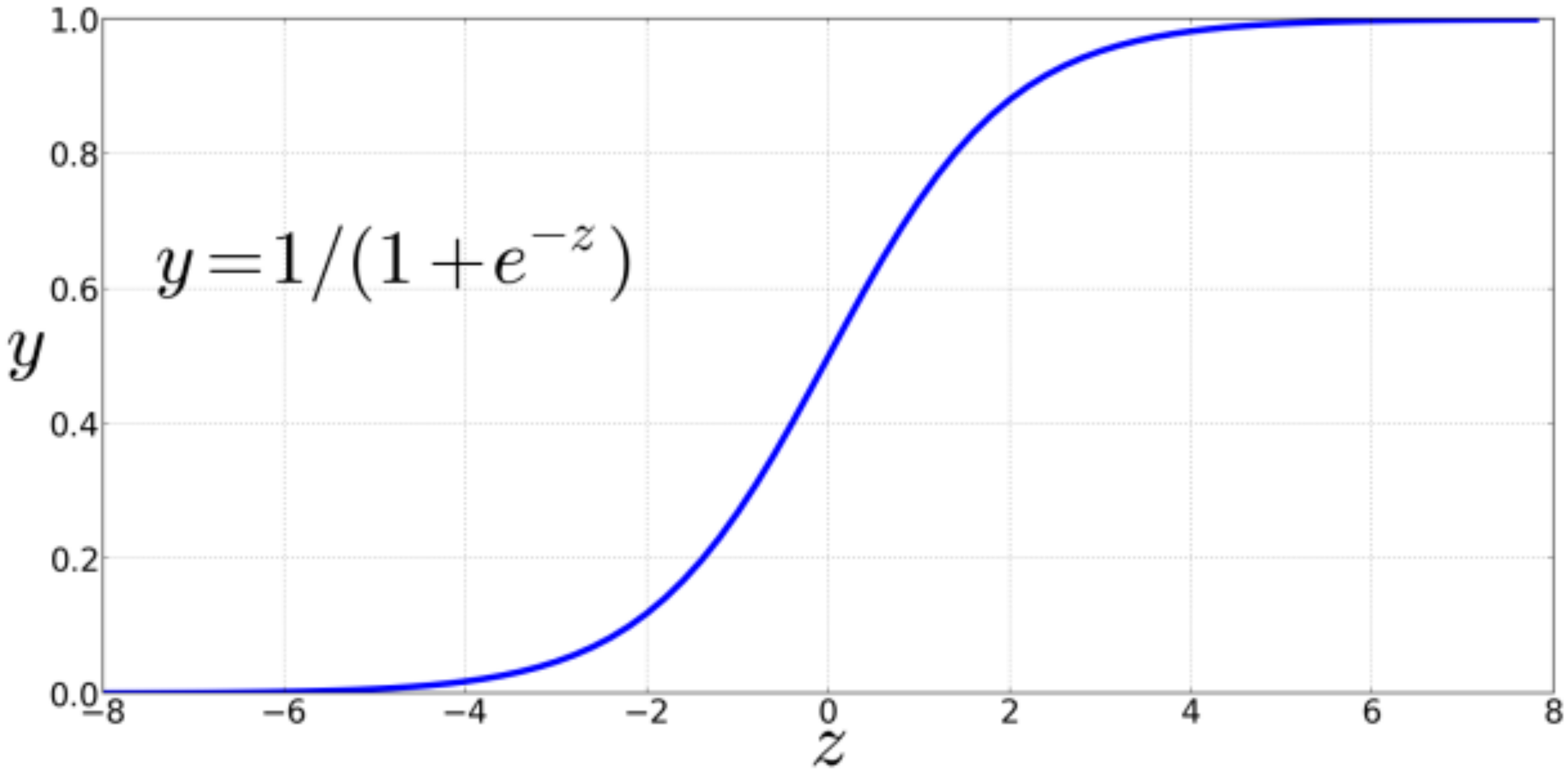
$z = \sum_i w_i x_i + b$

We can also write this as a dot product:

$z = w \cdot x + b$

But "z" here is not a probability

Sigmoid function



Probabilities in logistic regression

$$P(y=1) = \sigma(w \cdot x + b) = \frac{1}{1 + e^{-(w \cdot x + b)}}$$

Decision boundary

Now we have a function that -- given an instance x -- computes the probability $P(y=1|x)$. How do we make a decision?

$$\hat{y} = \begin{cases} 1 \ if \ P(y=1|x) > 0.5 \\ 0 \ otherwise \end{cases}$$

For a test instance x, we say **yes** if the probability $P(y=1|x)$ is more than .5, and **no** otherwise. We call .5 the decision boundary

Components of Classifiers

1. Feature representation of the input
2. Classification function
   - Used to compute our estimate of $p(y|x)$
   - For logistic regression this is the **sigmoid**
3. Objective function
   - Used during learning to minimize error on the training example.
   - For logistic regression this is **cross-entropy loss**
4. Algorithm to optimize the objective function
   - Here that's **stochastic gradient descent**

Feature Templates

- Typically “feature templates” are used to generate many features at once

- For each word *w*:
  - ${w}_count
  - ${w}_islowercase
  - ${w}_with_NOT_before_count

Extracting Features

It's hokey. There are virtually no surprises , and the writing is second-rate . So why was it so enjoyable? For one thing , the cast is great . Another nice touch is the music . I was overcome with the urge to get off the couch and start dancing . It sucked me in , and it'll do the same to you .

Var	Definition	Value
x_1	Count of positive lexicon words	3
x_2	Count of negative lexicon words	
x_3	Does no appear? (binary feature)	
x_4	Number of 1st and 2nd person pronouns	
x_5	Does ! appear? (binary feature)	
x_6	Log of the word count for the document	

Extracting Features

It's hokey. There are virtually no surprises , and the writing is second-rate . So why was it so enjoyable? For one thing , the cast is great . Another nice touch is the music . I was overcome with the urge to get off the couch and start dancing . It sucked me in , and it'll do the same to you .

Var	Definition	Value
x_1	Count of positive lexicon words	3
x_2	Count of negative lexicon words	2
x_3	Does no appear? (binary feature)	
x_4	Number of 1st and 2nd person pronouns	
x_5	Does ! appear? (binary feature)	
x_6	Log of the word count for the document	

Extracting Features

It's hokey. There are virtually no surprises , and the writing is second-rate . So why was it so enjoyable? For one thing , the cast is great . Another nice touch is the music . I was overcome with the urge to get off the couch and start dancing . It sucked me in , and it'll do the same to you .

Var	Definition	Value
x_1	Count of positive lexicon words	3
x_2	Count of negative lexicon words	2
x_3	Does no appear? (binary feature)	1
x_4	Number of 1st and 2nd person pronouns	
x_5	Does ! appear? (binary feature)	
x_6	Log of the word count for the document	

Extracting Features

It's hokey. There are virtually no surprises , and the writing is second-rate . So why was it so enjoyable? For one thing , the cast is great . Another nice touch is the music . I was overcome with the urge to get off the couch and start dancing . It sucked me in , and it'll do the same to you.

Var	Definition	Value
x_1	Count of positive lexicon words	3
x_2	Count of negative lexicon words	2
x_3	Does no appear? (binary feature)	1
x_4	Number of 1st and 2nd person pronouns	3
x_5	Does ! appear? (binary feature)	
x_6	Log of the word count for the document	

Extracting Features

It's hokey. There are virtually no surprises , and the writing is second-rate . So why was it so enjoyable? For one thing , the cast is great . Another nice touch is the music . I was overcome with the urge to get off the couch and start dancing . It sucked me in , and it'll do the same to you.

Word count = 64, ln(64) = 4.15

Var	Definition	Value
x_1	Count of positive lexicon words	3
x_2	Count of negative lexicon words	2
x_3	Does no appear? (binary feature)	1
x_4	Number of 1st and 2nd person pronouns	3
x_5	Does ! appear? (binary feature)	0
x_6	Log of the word count for the document	4.15

Components of Classifiers

1. Feature representation of the input
2. Classification function
   - Used to compute our estimate of p(y|x)
   - For logistic regression this is the **sigmoid**
3. Objective function
   - Used during learning to minimize error on the training example.
   - For logistic regression this is **cross-entropy loss**
4. Algorithm to optimize the objective function
   - Here that’s **stochastic gradient descent**

Computing Z

Var	Definition	Value	Weight	Product
x_1	Count of positive lexicon words	3	2.5	7.5
x_2	Count of negative lexicon words	2	-5.0	-10
x_3	Does no appear? (binary feature)	1	-1.2	-1.2
x_4	Num 1st and 2nd person pronouns	3	0.5	1.5
x_5	Does ! appear? (binary feature)	0	2.0	0
x_6	Log of the word count for the doc	4.15	0.7	2.905
b	bias	1	0.1	.1

$$z = \sum_i w_i x_i + b$$

Z=0.805

Sigmoid(Z)

Var	Definition	Value	Weight	Product
x_1	Count of positive lexicon words	3	2.5	7.5
x_2	Count of negative lexicon words	2	-5.0	-10
x_3	Does no appear? (binary feature)	1	-1.2	-1.2
x_4	Num 1st and 2nd person pronouns	3	0.5	1.5
x_5	Does ! a		2.0	0
x_6	Log of th		0.7	2.905
b	bias		0.1	.1

$\sigma(0.805)$

$= 0.69$

Components of Classifiers

1. Feature representation of the input
2. Classification function
   - Used to compute our estimate of p(y|x)
   - For logistic regression this is the **sigmoid**
3. Objective function
   - Used during learning to minimize error on the training example.
   - For logistic regression this is **cross-entropy loss**
4. Algorithm to optimize the objective function
   - Here that's **stochastic gradient descent**

Learning in Logistic Regression

How do we get the weights of the model? We need to learn the parameters (weights + bias).

This requires 2 components:

1. An objective function or **loss function** that tells us the ***distance*** between the system output and the gold output. We will use **cross-entropy loss**.

2. An algorithm for optimizing the objective function. We will use stochastic gradient descent to **minimize** the **loss function.**

Loss functions

We need to determine for some observation *x* how close the classifier output ($\hat{y}$= σ (*w* · *x* + *b*)) is to the correct output (*y*, which is 0 or 1).

$$L(\hat{y}, y) = \text{how much } \hat{y} \text{ differs from the true } y$$

One example is mean squared error

$$L_{MSE}(\hat{y}, y) = \frac{1}{2}(\hat{y} - y)^2$$

Loss functions for probabilistic classification

- We use a loss function that prefers the correct class labels of the training example to be more likely.

- Conditional MLE: Choose parameters *w, b* that maximize the (log) probabilities of the true labels in the training data.

- The resulting loss function is the negative log likelihood loss, more commonly called the **cross entropy loss**.

Loss functions for probabilistic classification

For one observation x, let's **maximize** the probability of the correct label p(y|x).

$$p(y|x) = \hat{y}^{y}(1 - \hat{y})^{1-y}$$

If y = 1, then $\mathbf{p(y|x)} = \hat{y}$.

If y = 0, then $\mathbf{p(y|x)} = 1 - \hat{y}$.

Loss functions for probabilistic classification

Change to logs (still maximizing)

$$\log p(y|x) = \log[\hat{y}^{y}(1-\hat{y})^{1-y}]$$

This tells us what **log likelihood should be maximized. But for loss functions, we want to minimize things, so we'll flip the sign.**

Cross-entropy loss

The result is cross-entropy loss:

$$L_{CE}(\hat{y}, y) = -\log p(y|x) = -[y \log \hat{y} + (1-y)\log(1-\hat{y})]$$

Finally, plug in the definition for $\hat{\boldsymbol{y}} = \sigma\,(w \cdot x) + b$

$$L_{CE}(\hat{y}, y) = -[y \log \sigma(w \cdot x + b) \; + (1-y)\log(1-\sigma(w \cdot x + b))]$$

Cross-entropy loss: example

Why does minimizing this negative log probability do what we want? We want the loss to be smaller if the model's estimate is close to correct, and we want the loss to be bigger if it is confused.

It's hokey. There are virtually no surprises , and the writing is second-rate . So why was it so enjoyable? For one thing , the cast is great . Another nice touch is the music . I was overcome with the urge to get off the couch and start dancing . It sucked me in , and it'll do the same to you .

P(sentiment=1|It's hokey...) = 0.69. Let's say y=1.

$$L_{CE}(\hat{y}, y) = -[y \log \sigma(w\cdot x+b) + (1-y)\log(1-\sigma(w\cdot x+b))]$$

$$= -[\log \sigma(w\cdot x+b)]$$

$$= -\log(0.69) = \mathbf{0.37}$$

Cross-entropy loss: example

Why does minimizing this negative log probability do what we want? We want the loss to be smaller if the model's estimate is close to correct, and we want the loss to be bigger if it is confused.

It's hokey. There are virtually no surprises , and the writing is second-rate . So why was it so enjoyable? For one thing , the cast is great . Another nice touch is the music . I was overcome with the urge to get off the couch and start dancing . It sucked me in , and it'll do the same to you.

P(sentiment=1|It's hokey...) = 0.69. Let's **pretend** y=0.

$$L_{CE}(\hat{y}, y) = -[y \log \sigma(w \cdot x + b) + (1-y) \log(1 - \sigma(w \cdot x + b))]$$

$$= -[\log(1 - \sigma(w \cdot x + b))]$$

$$= -\log(0.31) = \mathbf{1.17}$$

Cross-entropy loss: example

Why does minimizing this negative log probability do what we want? We want the loss to be smaller if the model's estimate is close to correct, and we want the loss to be bigger if it is confused.

It's hokey. There are virtually no surprises , and the writing is second-rate . So why was it so enjoyable? For one thing , the cast is great . Another nice touch is the music . I was overcome with the urge to get off the couch and start dancing . It sucked me in , and it'll do the same to you .

If our prediction is **correct**, then our CE loss is **lower**

$$= -\log(0.69) = \mathbf{0.37}$$

If our prediction is **incorrect**, then our CE loss is **higher**

$$-\log(0.31) = \mathbf{1.17}$$

Components of Classifiers

1. Feature representation of the input
2. Classification function
   - Used to compute our estimate of p(y|x)
   - For logistic regression this is the **sigmoid**
3. Objective function
   - Used during learning to minimize error on the training example.
   - For logistic regression this is **cross-entropy loss**
4. Algorithm to optimize the objective function
   - Here that's **stochastic gradient descent**

NB vs. LR: Parameter Learning

- Naïve Bayes:
  - Learn conditional probabilities **independently** by counting
- Logistic Regression:
  - Learn weights **jointly**

Closed Form Solution

- a Closed Form Solution is a simple solution that works instantly without any loops, functions etc

- e.g. the sum of integer from 1 to n

```
s= 0
for i in 1 to n
    s = s + i
end for
print s
```

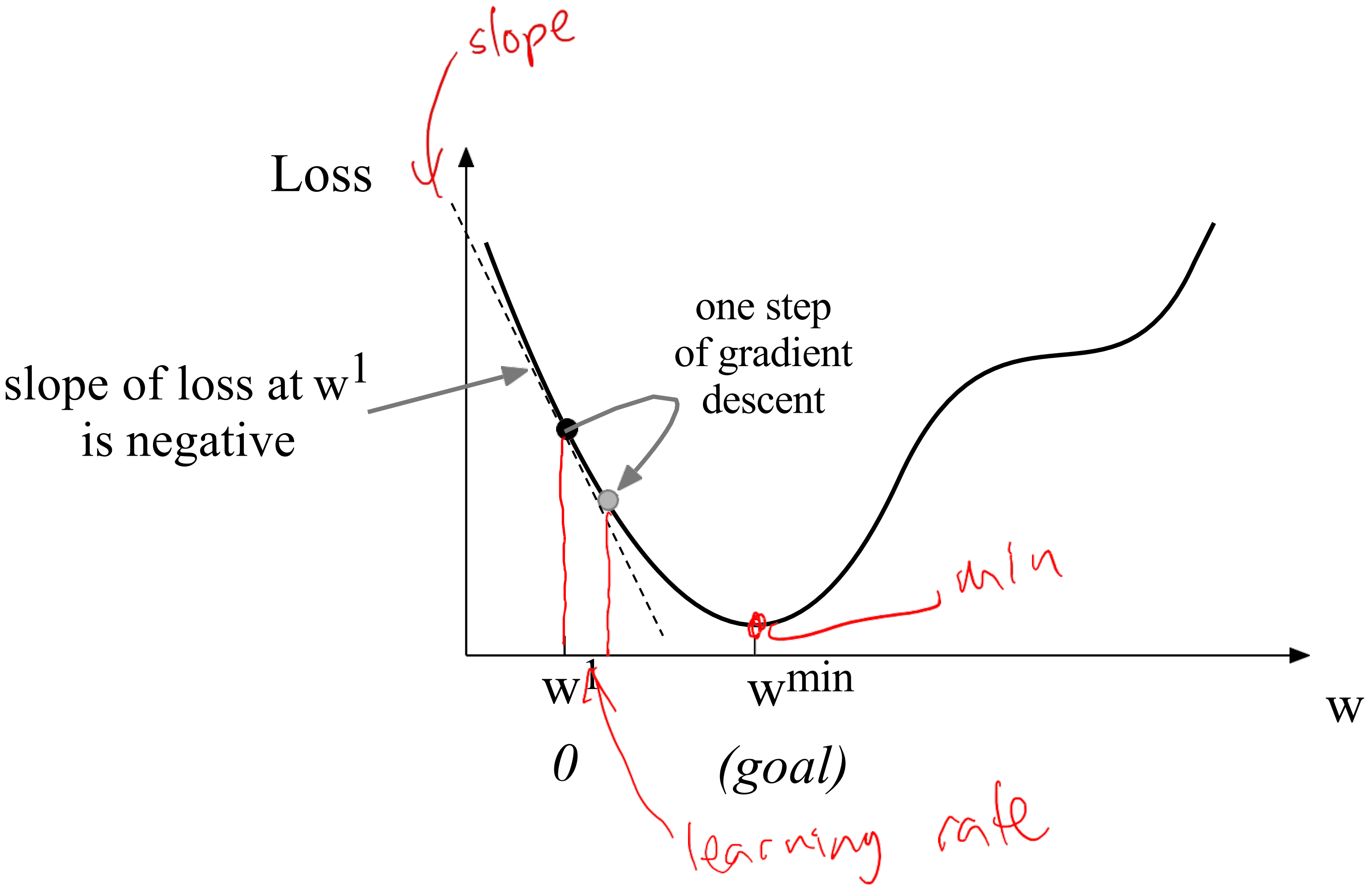
Iterative Algorithm

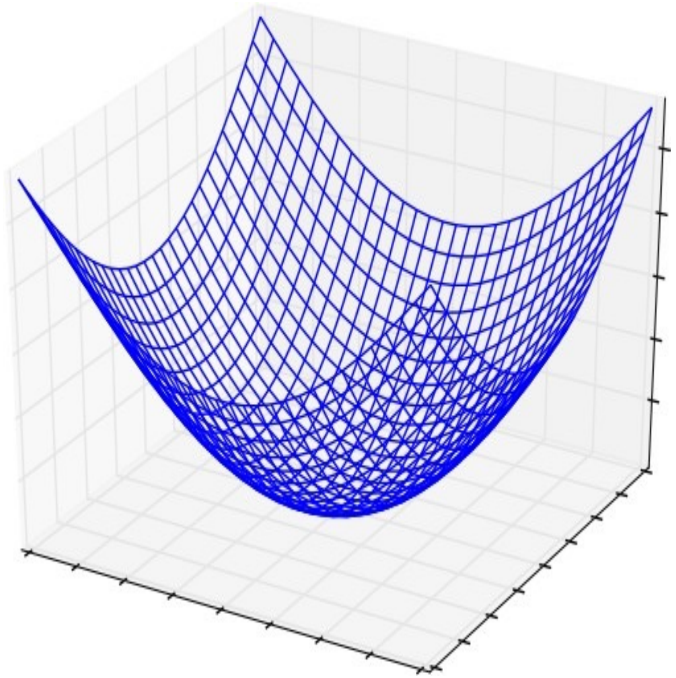
$s = n(n+1)/2$

Closed Form Solution

Gradient descent

Iteratively find minimum

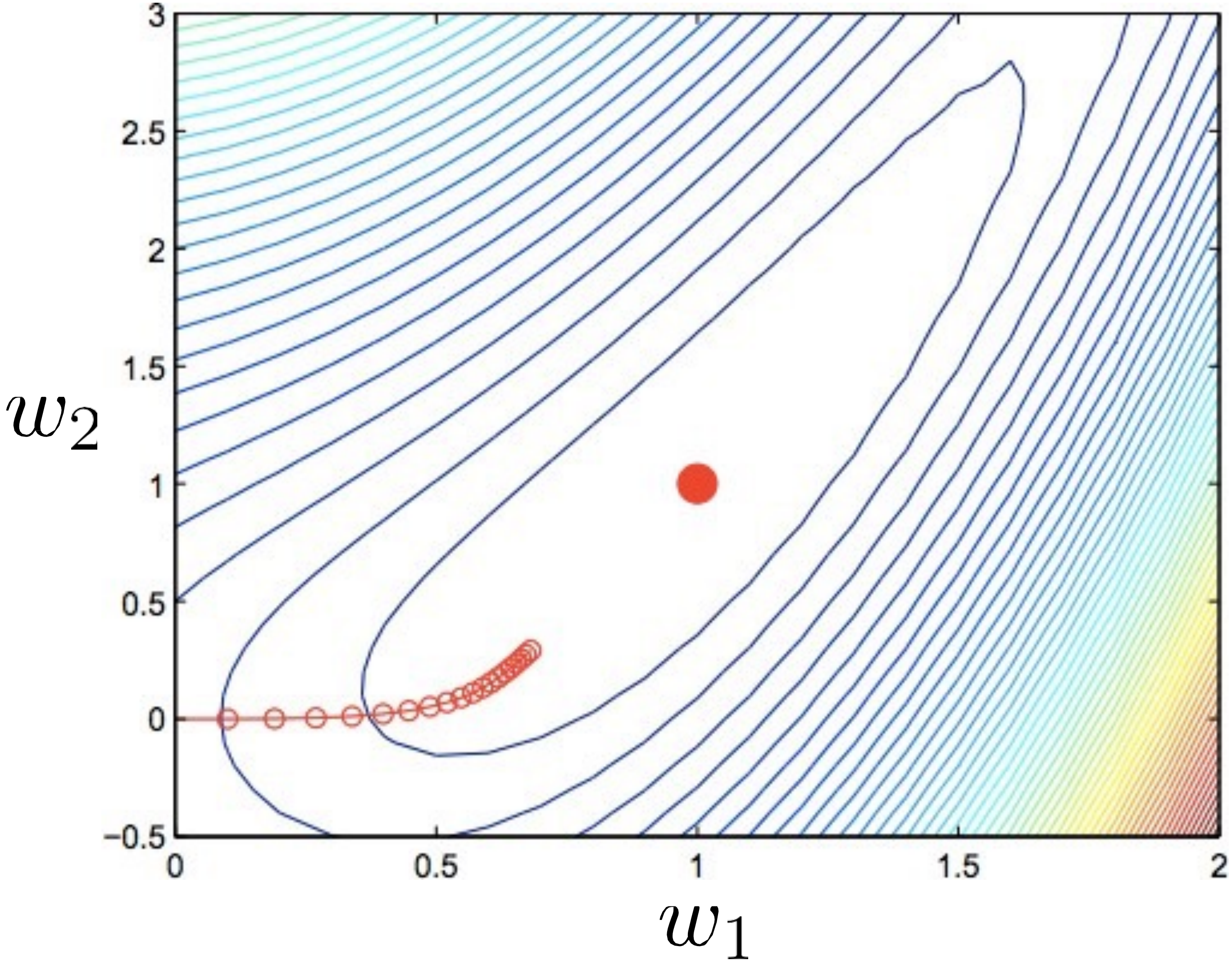




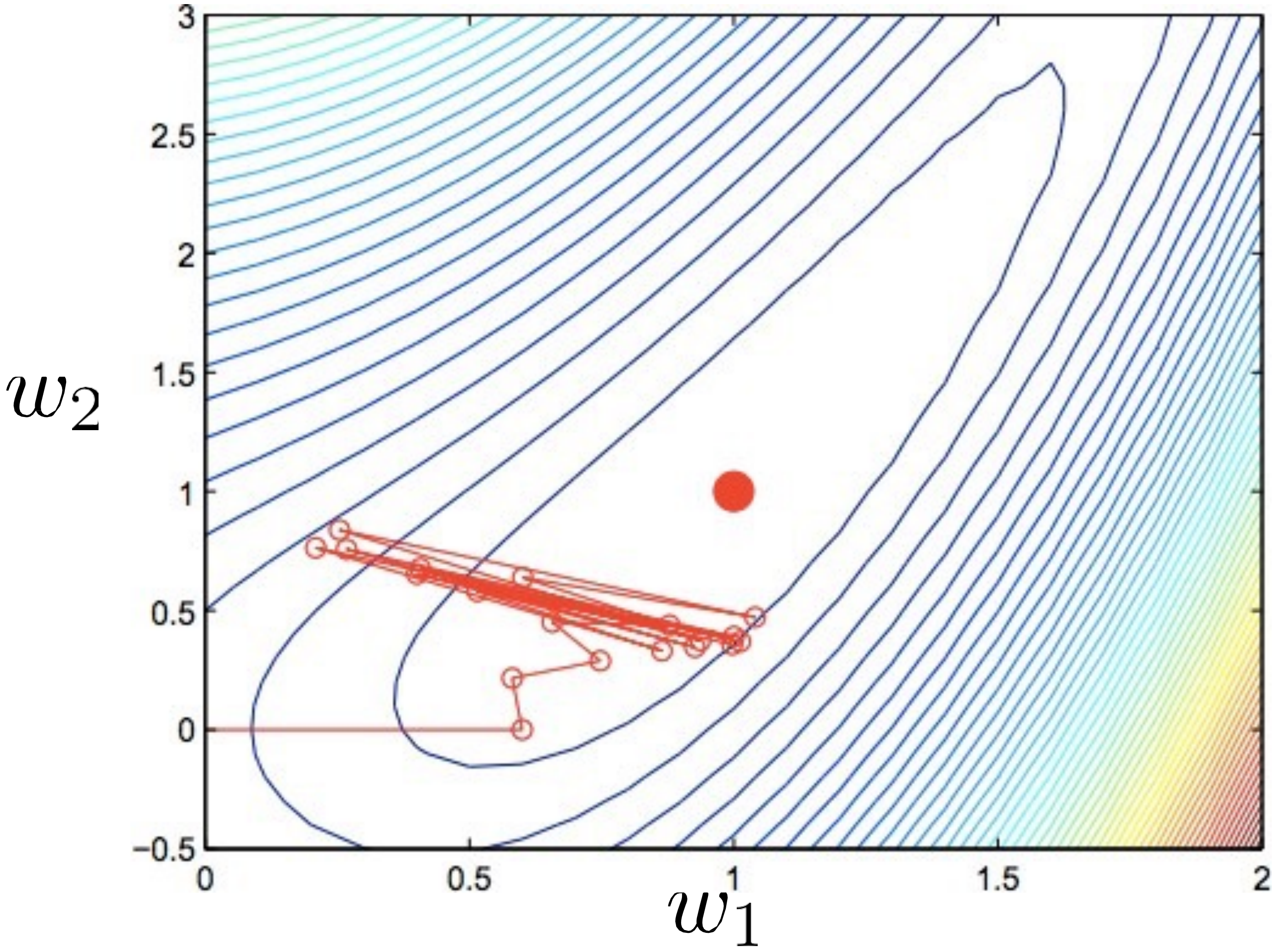
There are many other issues to consider with learning regression models like this

- But this is not a machine learning course!

Gradient Descent



Gradient Descent



Logistic Regression: Pros and Cons

- Doesn't assume conditional independence of features
  - Better calibrated probabilities
  - Can handle highly correlated or overlapping features

- But NB is faster to train, less likely to overfit

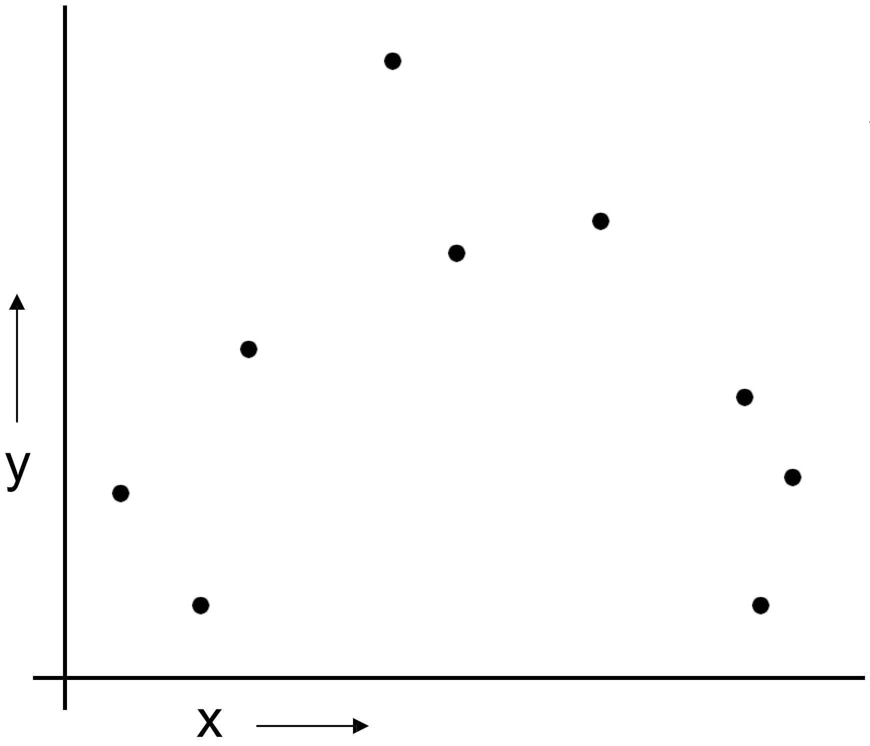
Cross-Validation

Cross-validation slides adapted from:

Andrew W. Moore Carnegie Mellon

www.cs.cmu.edu/~awm

A regression problem

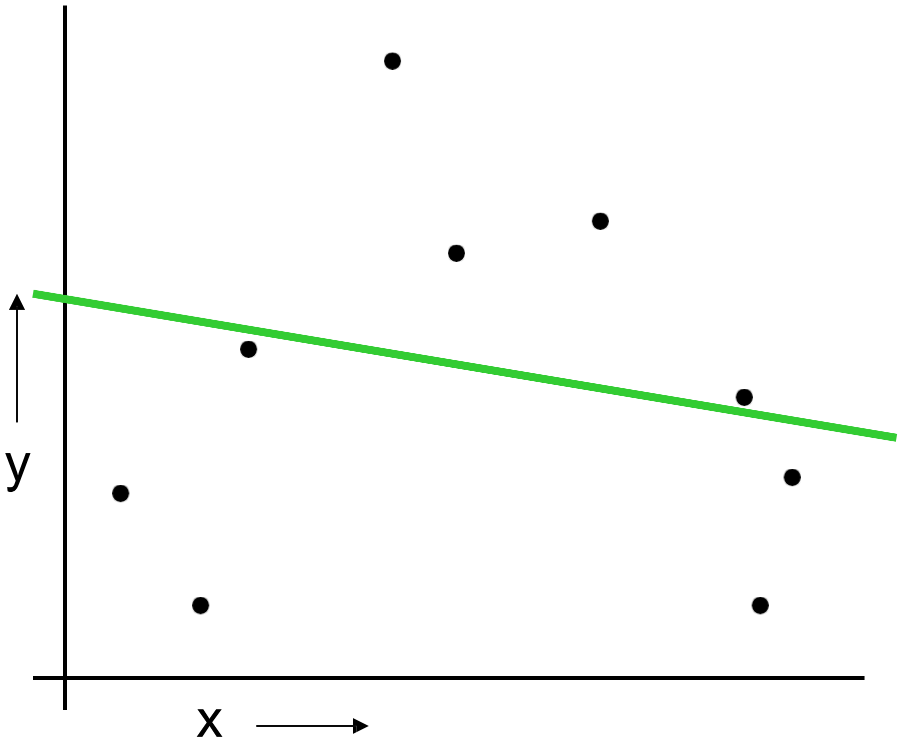


y = f(x) + noise

Can we learn f from this data?

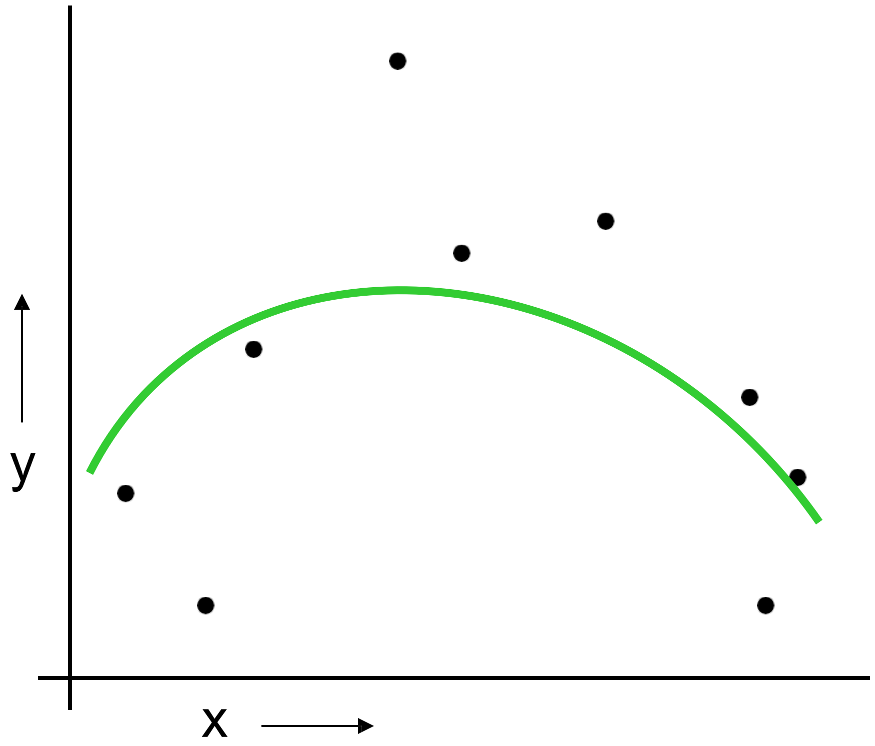
Let’s consider three methods…

Liner Regression



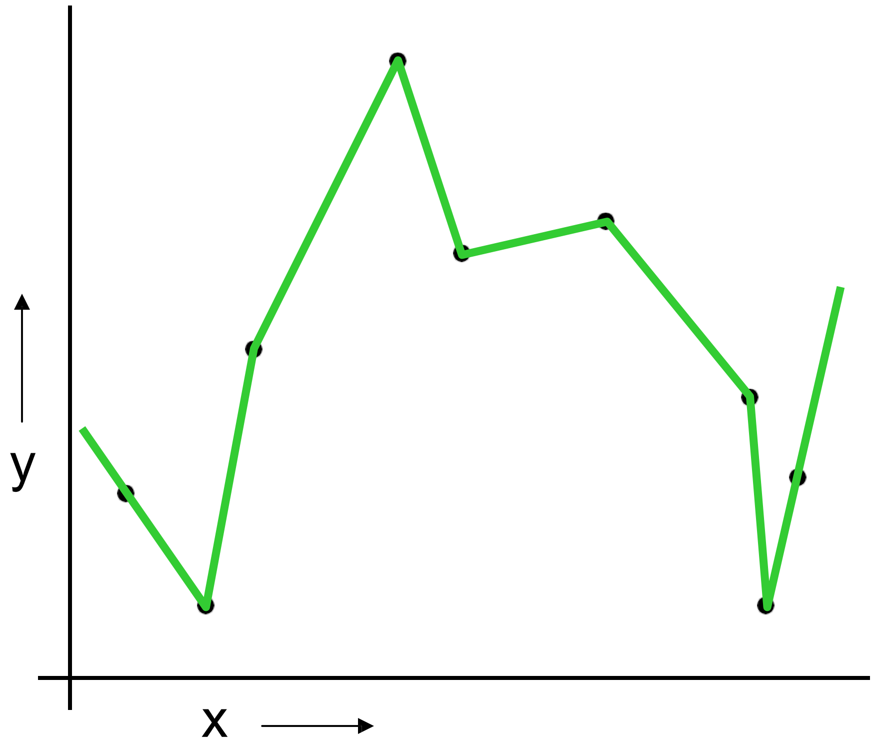
$$y^{est} = \beta_0 + \beta_1 x$$

Quadratic Regression



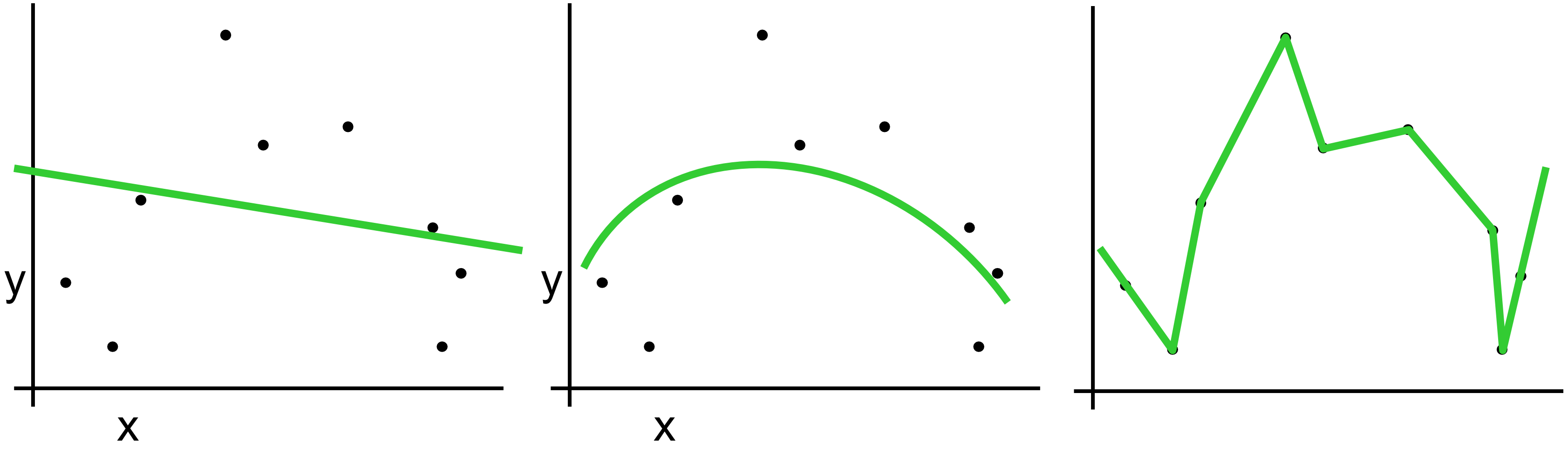
$$y^{est} = \beta_0 + \beta_1 x + \beta_2 x^2$$

”Join-the-dots”



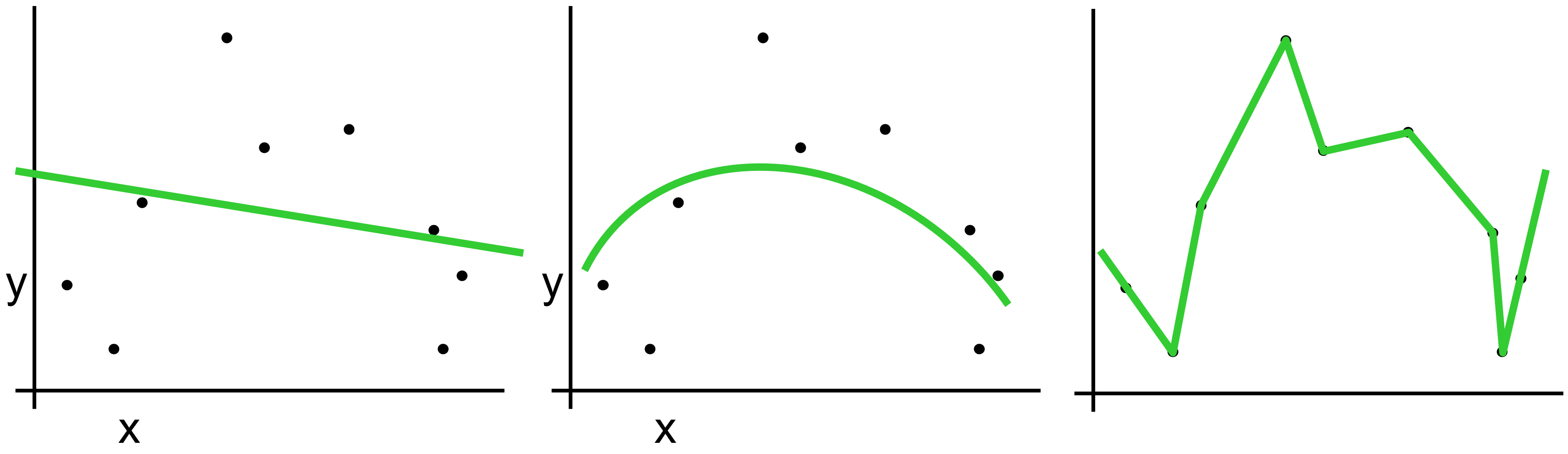
Also known as piecewise linear nonparametric regression if you want to feel fancy

Which is best?



Why not choose the method with the
best fit to the data?

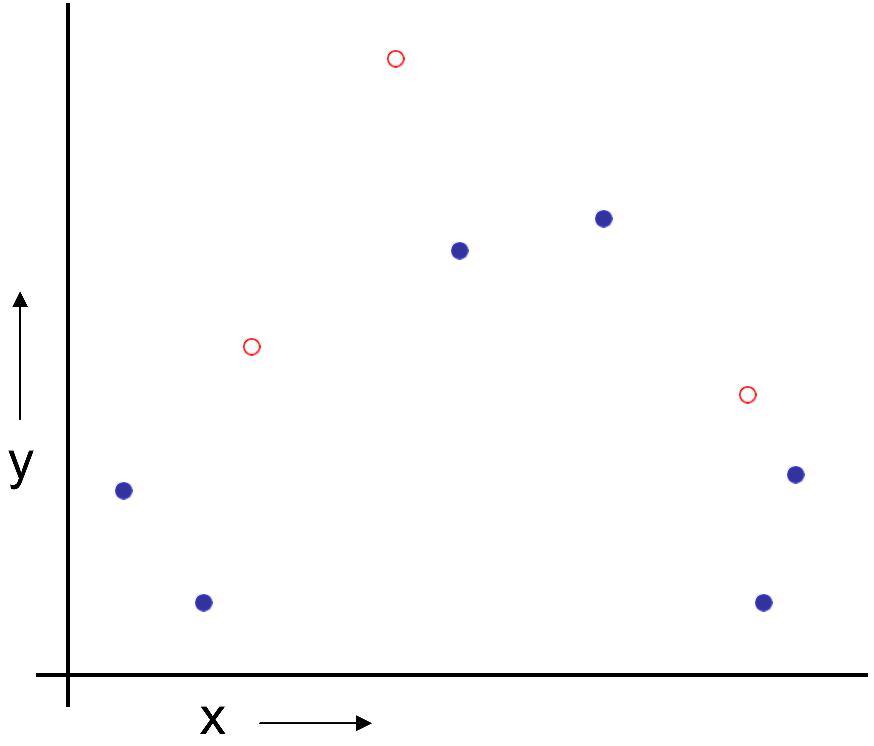
What do we really want?



Why not choose the method with the best fit to the data?

"How well are you going to predict future data drawn from the same distribution?"

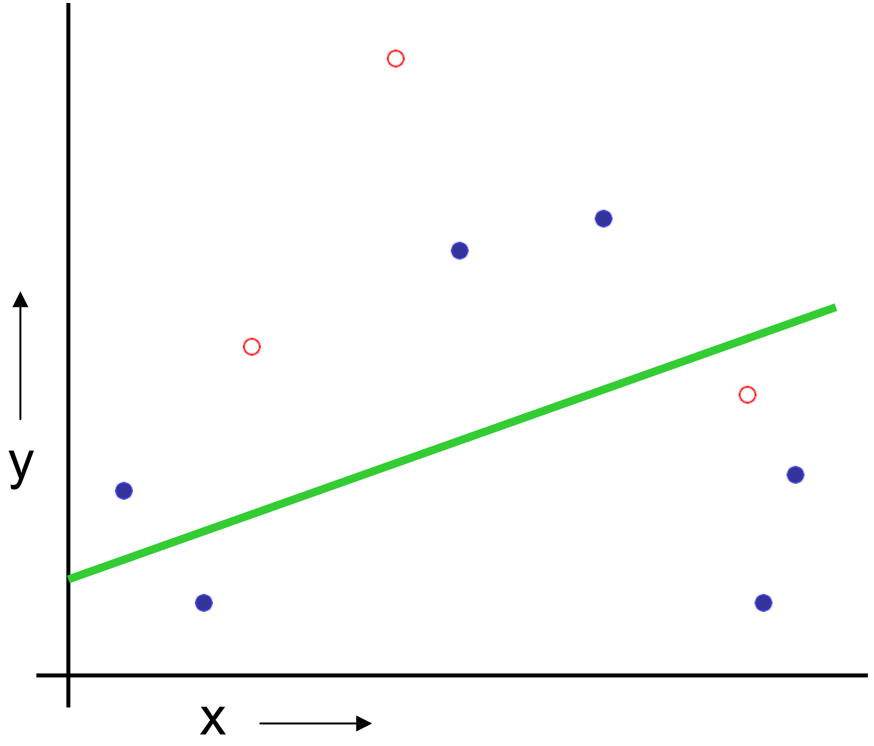
The test set method



1. Randomly choose 30% of the data to be in a test set

2. The remainder is a training set

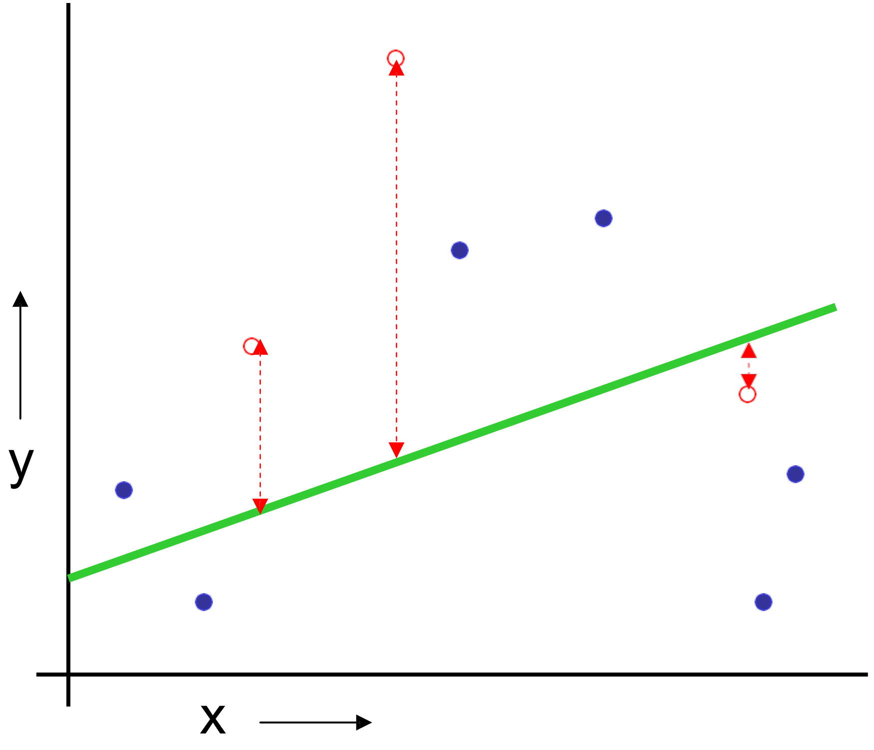
The test set method



(Linear regression example)

1. Randomly choose 30% of the data to be in a test set
2. The remainder is a training set
3. Perform your regression on the training set

The test set method

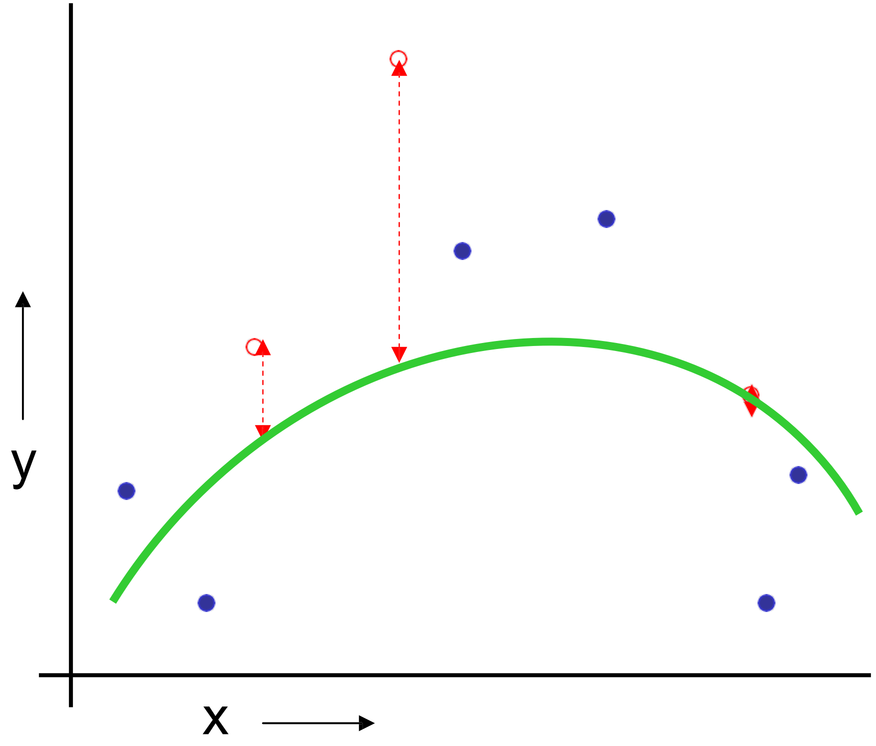


(Linear regression example)

Mean Squared Error = 2.4

1. Randomly choose 30% of the data to be in a test set
2. The remainder is a training set
3. Perform your regression on the training set
4. Estimate your future performance with the test set

The test set method

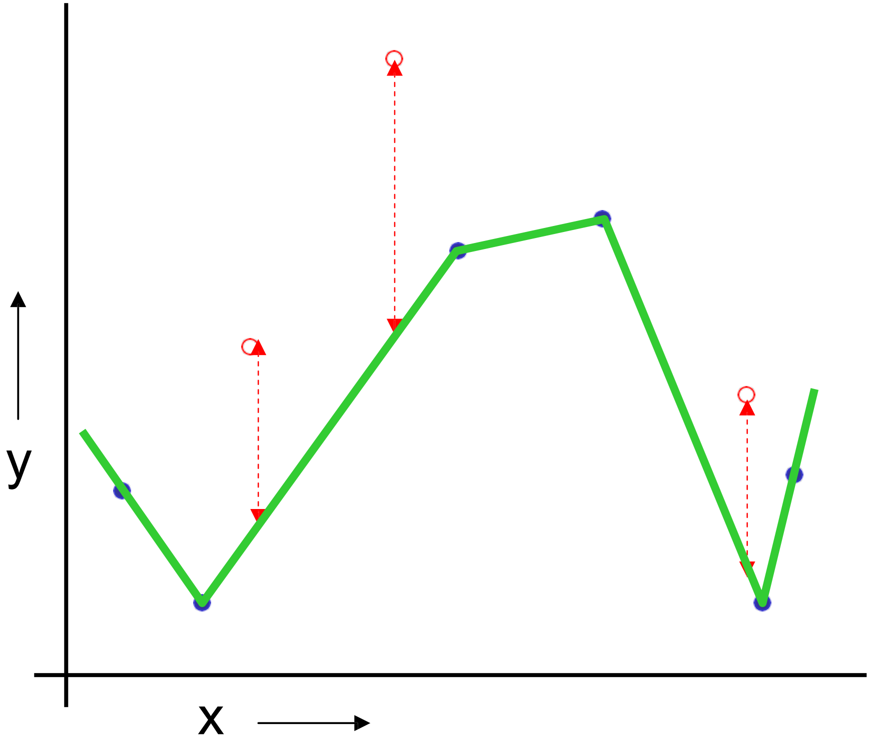


(Quadratic regression example)

Mean Squared Error = 0.9

1. Randomly choose 30% of the data to be in a test set
2. The remainder is a training set
3. Perform your regression on the training set
4. Estimate your future performance with the test set

The test set method



(Join the dots example)

Mean Squared Error = 2.2

1. Randomly choose 30% of the data to be in a test set
2. The remainder is a training set
3. Perform your regression on the training set
4. Estimate your future performance with the test set

The test set method

Good news:

- Very very simple
- Can then simply choose the method with the best test-set score

Bad news:

- What’s the downside?

The test set method

Good news:

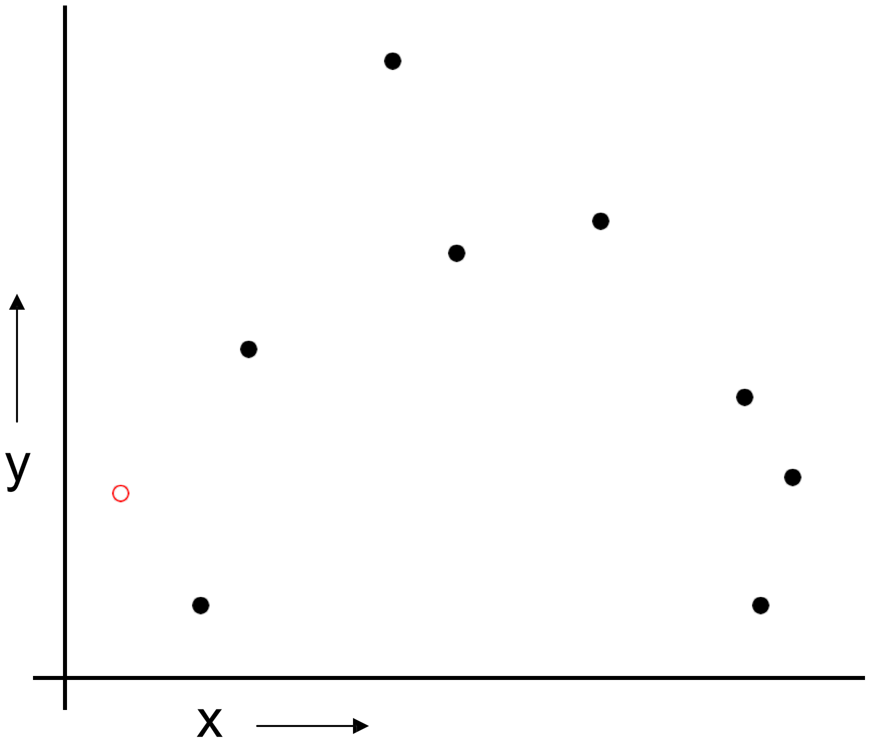
- Very very simple
- Can then simply choose the method with the best test-set score

Bad news:

- Wastes data: we get an estimate of the best method to apply to 30% less data
- If we don't have much data, our test-set might just be lucky or unlucky

We say the "test-set estimator of performance has high variance"

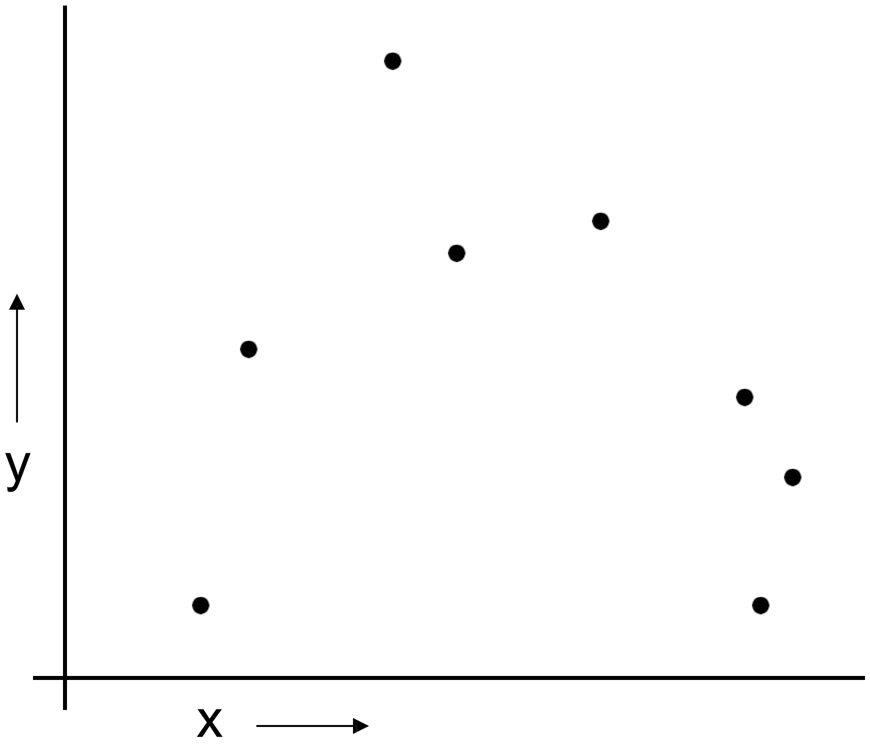
LOOCV (Leave-one-out Cross Validation)



- For k=1 to R

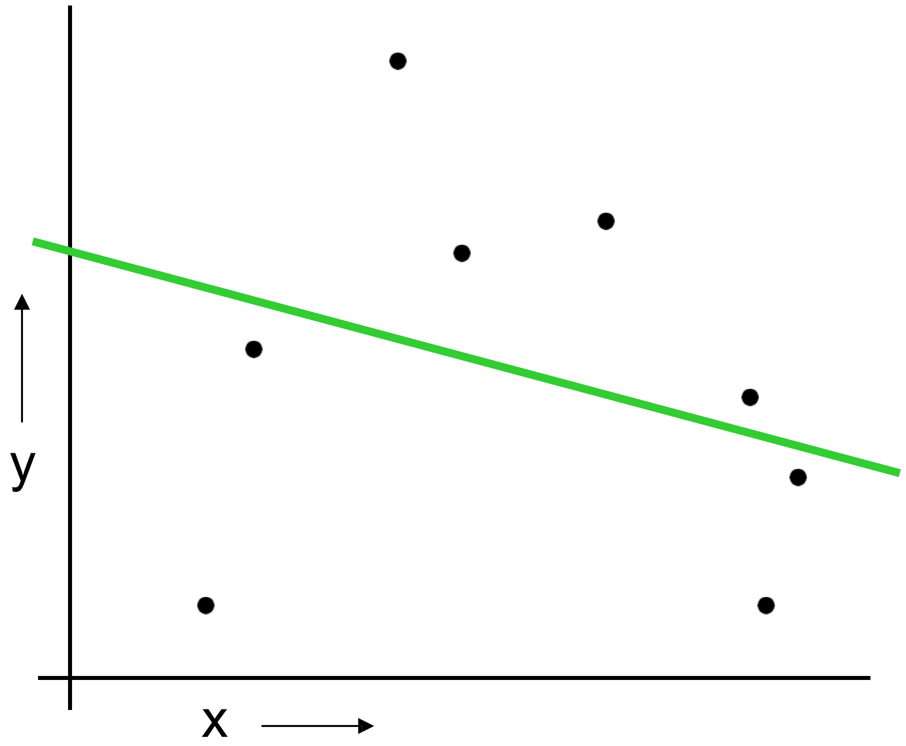
  1. Let (x_k, y_k) be the k^{th} record

LOOCV (Leave-one-out Cross Validation)



- For k=1 to R
  1. Let (x_k, y_k) be the k^{th} record
  2. Temporarily remove (x_k, y_k) from the dataset

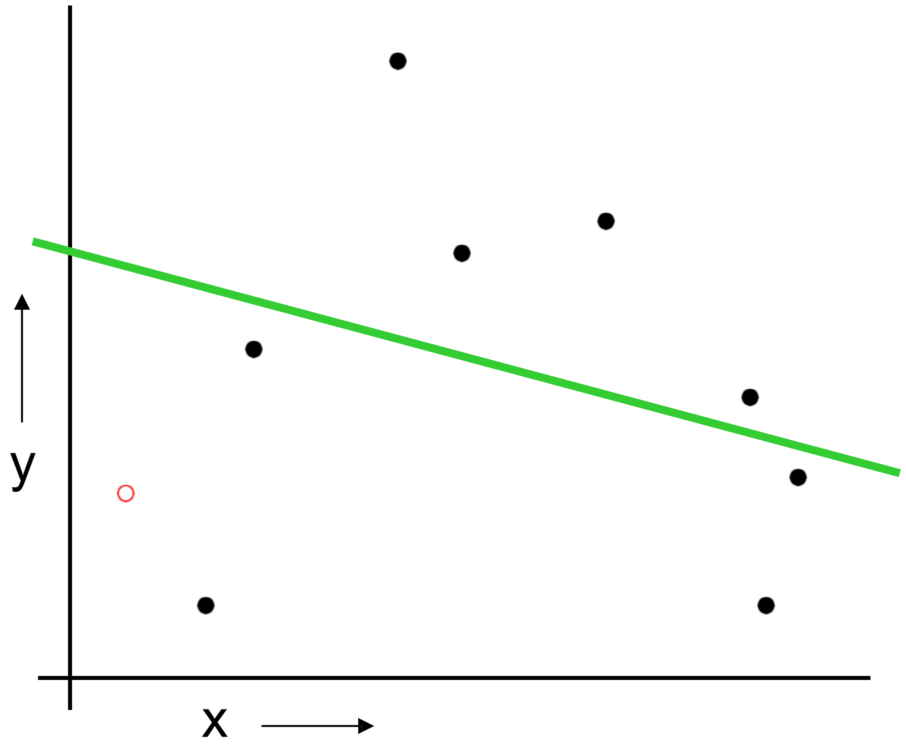
LOOCV (Leave-one-out Cross Validation)



- For k=1 to R

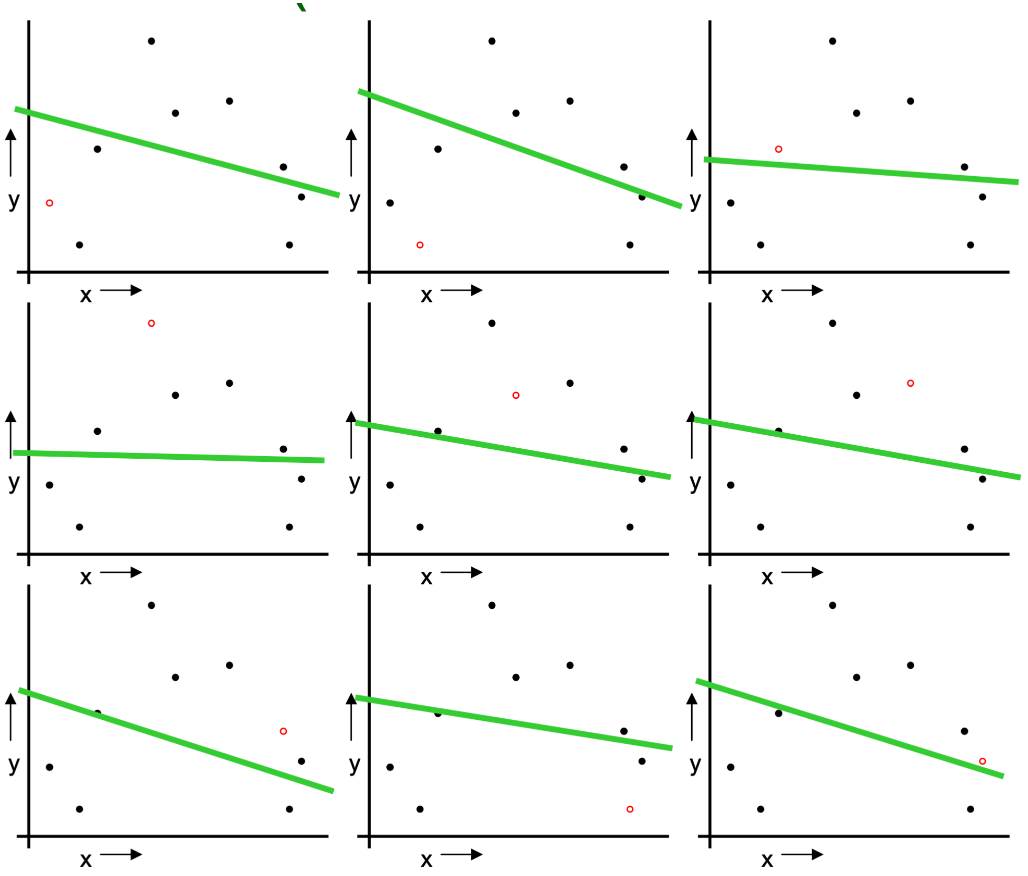
1. Let (x_k, y_k) be the k^{th} record
2. Temporarily remove (x_k, y_k) from the dataset
3. Train on the remaining R-1 datapoints

LOOCV (Leave-one-out Cross Validation)



- For k=1 to R
  1. Let (x_k, y_k) be the k^{th} record
  2. Temporarily remove (x_k, y_k) from the dataset
  3. Train on the remaining R-1 datapoints
  4. Note your error (x_k, y_k)
- When you've done all points, report the mean error.

LOOCV (Leave-one-out Cross Validation)



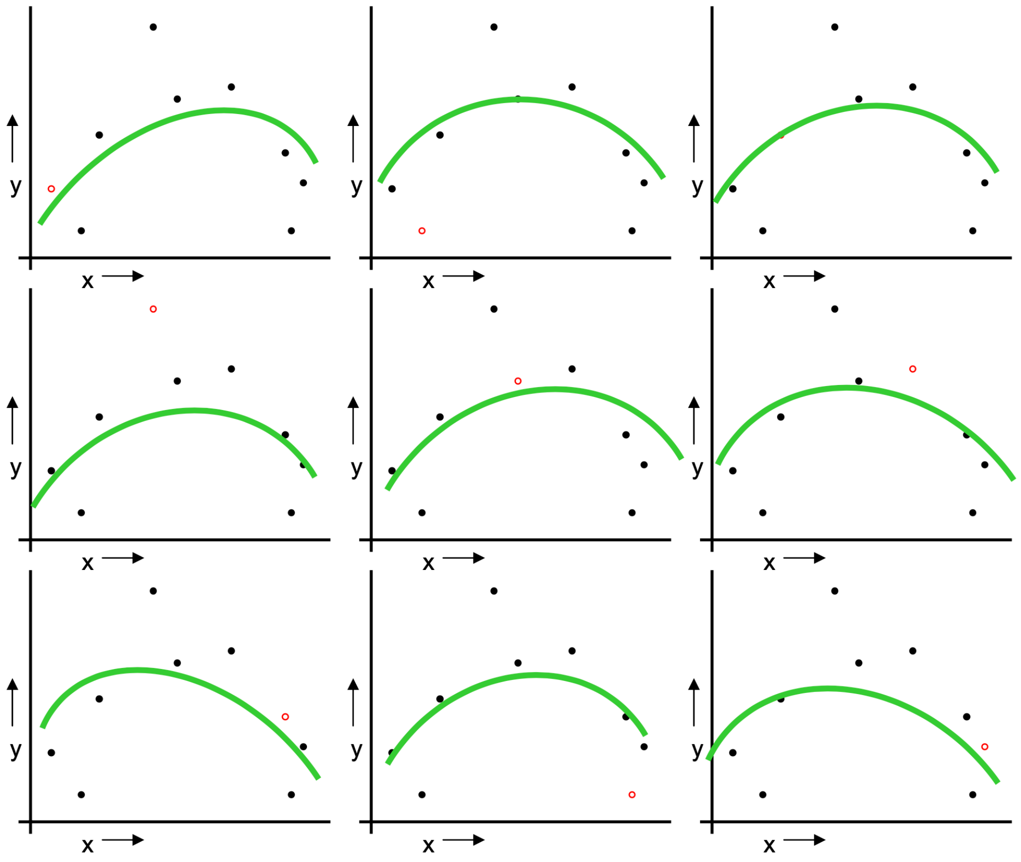
For k=1 to R

1. Let (x_k,y_k) be the k^{th} record
2. Temporarily remove (x_k,y_k) from the dataset
3. Train on the remaining R-1 datapoints
4. Note your error (x_k,y_k)

When you've done all points, report the mean error.

MSE_{LOOCV} = 2.12

LOOCV (Leave-one-out Cross Validation)



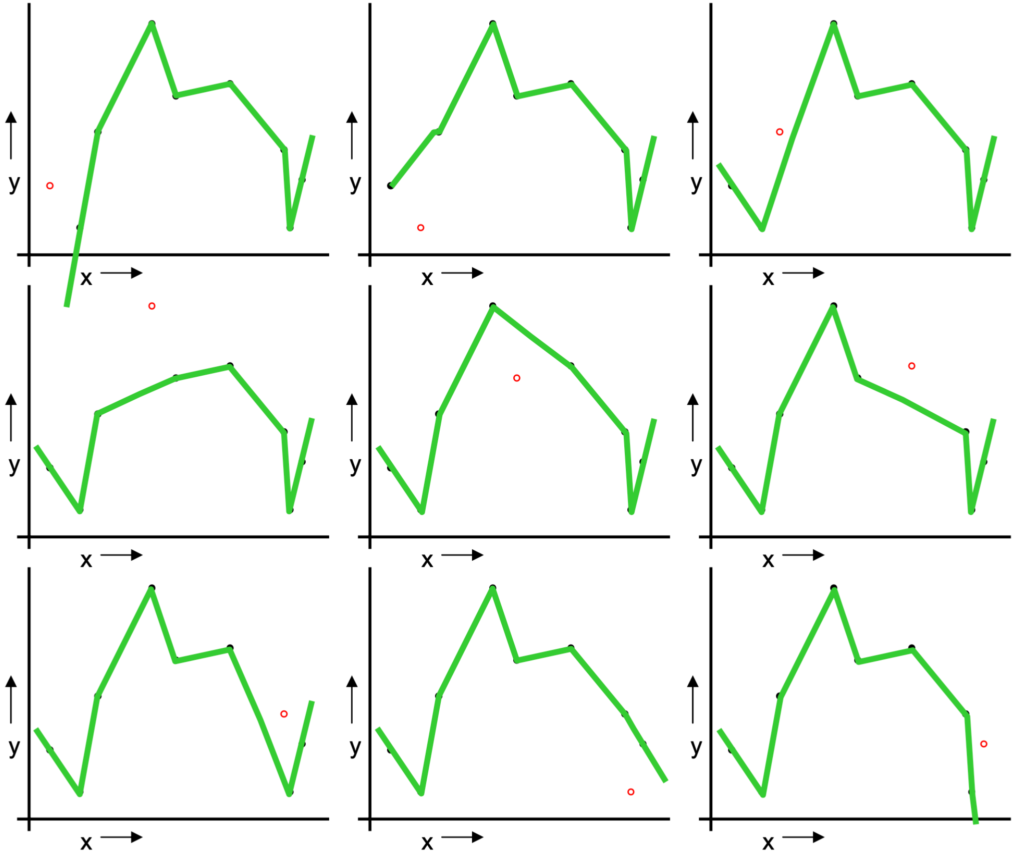
For k=1 to R

1. Let (x_k, y_k) be the k^{th} record
2. Temporarily remove (x_k, y_k) from the dataset
3. Train on the remaining R-1 datapoints
4. Note your error (x_k, y_k)

When you've done all points, report the mean error.

$$MSE_{LOOCV} = 0.962$$

LOOCV (Leave-one-out Cross Validation)



For k=1 to R

1. Let (x_k,y_k) be the k^{th} record
2. Temporarily remove (x_k,y_k) from the dataset
3. Train on the remaining R-1 datapoints
4. Note your error (x_k,y_k)

When you've done all points, report the mean error.

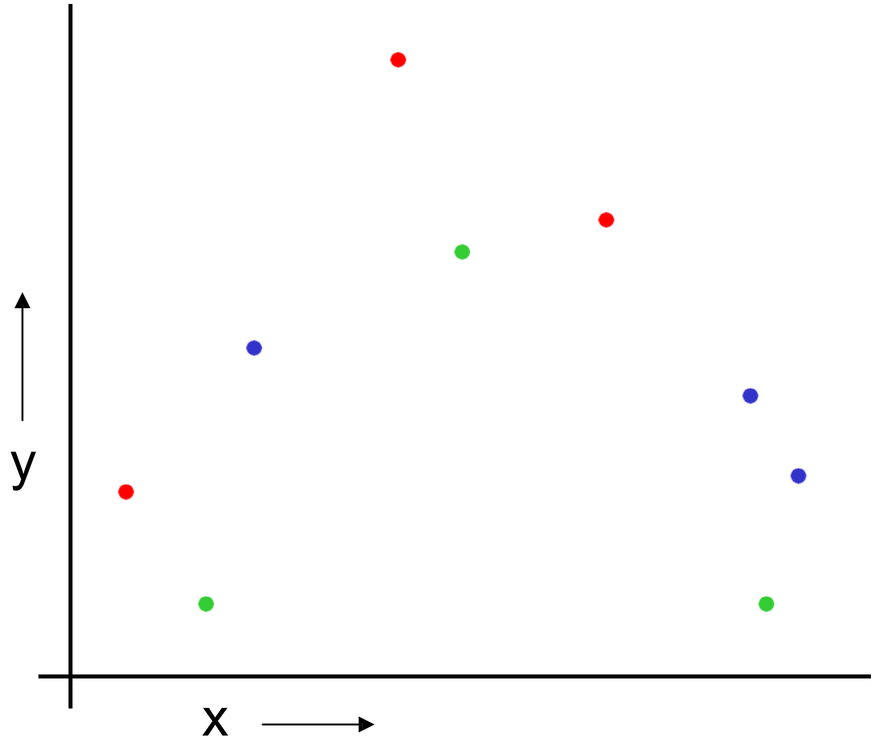
$$MSE_{LOOCV} = 3.33$$

What kind of cross validation?

	Downside	Upside
Test-set	Variance: unreliable estimate of future performance	Cheap
Leave-one-out	Expensive. Has some weird behavior	Doesn't waste data

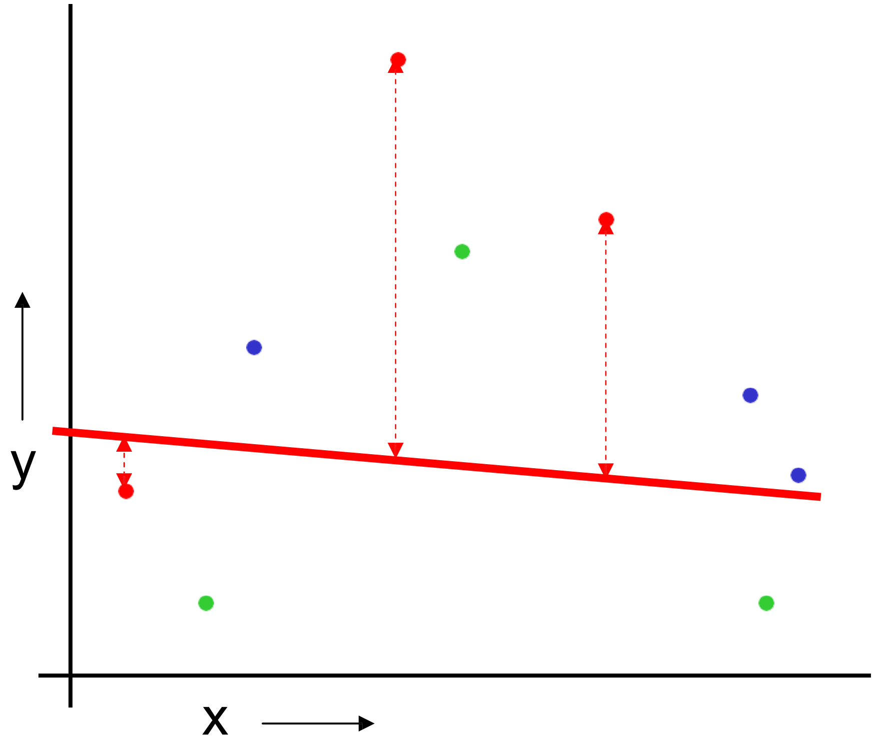
..can we get the best of both worlds?

K-fold Cross Validation



Randomly break the dataset into k partitions (in our example we'll have k=3 partitions colored Red Green and Blue)

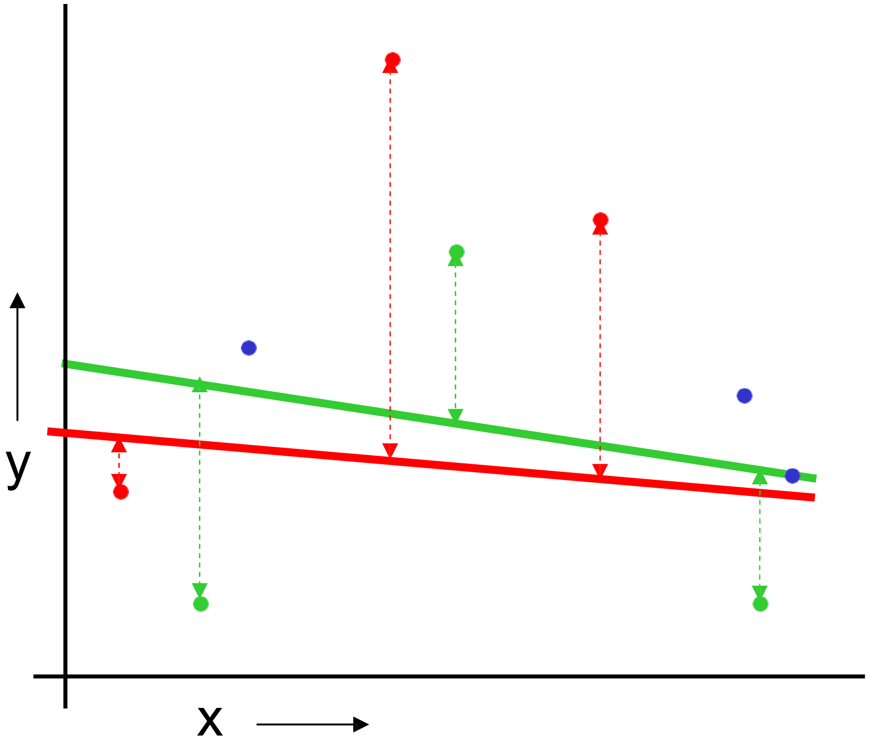
K-fold Cross Validation



Randomly break the dataset into k partitions (in our example we'll have k=3 partitions colored Red Green and Blue)

For the red partition: Train on all the points not in the red partition. Find the test-set sum of errors on the red points.

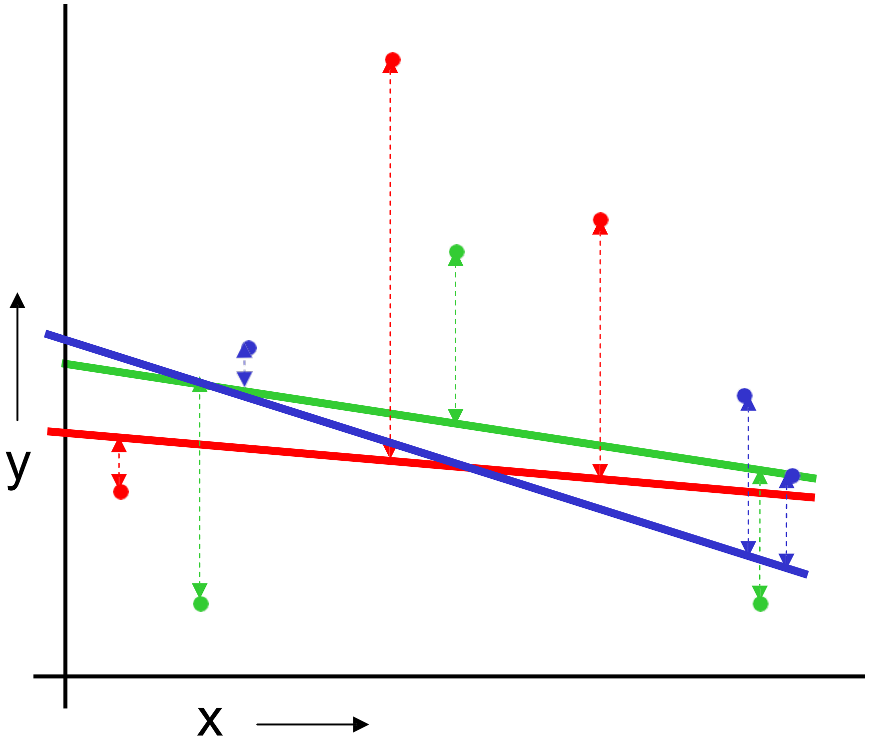
K-fold Cross Validation



Randomly break the dataset into k partitions (in our example we'll have k=3 partitions colored Red Green and Blue)

- For the red partition: Train on all the points not in the red partition. Find the test-set sum of errors on the red points.
- For the green partition: Train on all the points not in the green partition.
  Find the test-set sum of errors on the green points.

K-fold Cross Validation



Linear Regression
MSE_{3FOLD}=2.05

Randomly break the dataset into k partitions (in our example we'll have k=3 partitions colored Red Green and Blue)

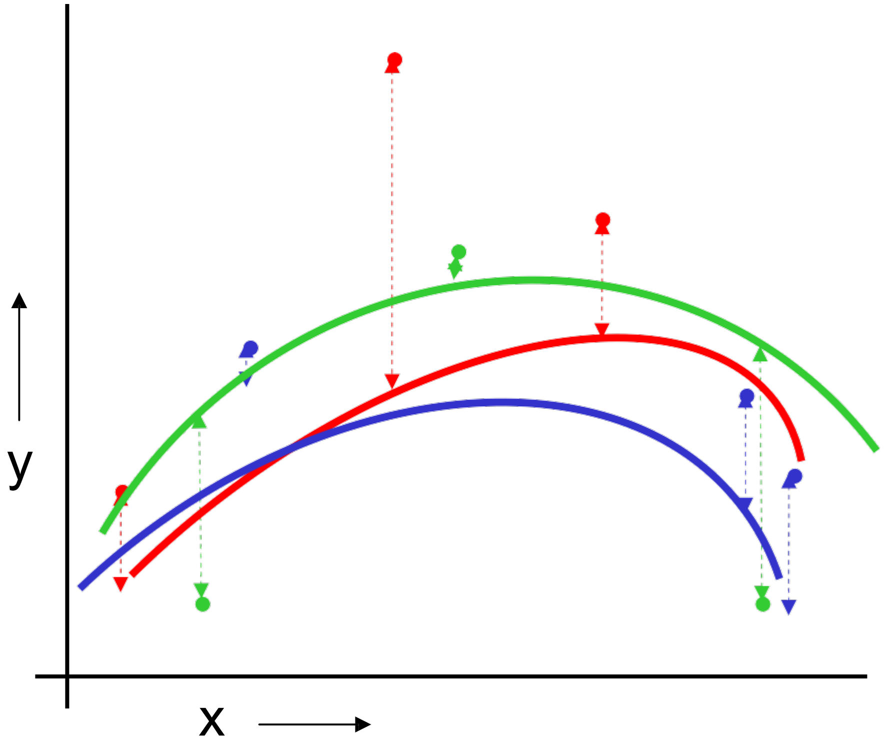
For the red partition: Train on all the points not in the red partition. Find the test-set sum of errors on the red points.

For the green partition: Train on all the points not in the green partition. Find the test-set sum of errors on the green points.

For the blue partition: Train on all the points not in the blue partition. Find the test-set sum of errors on the blue points.

Then report the mean error

K-fold Cross Validation



Quadratic Regression

$MSE_{3FOLD}=1.11$

Randomly break the dataset into k partitions (in our example we'll have k=3 partitions colored Red Green and Blue)

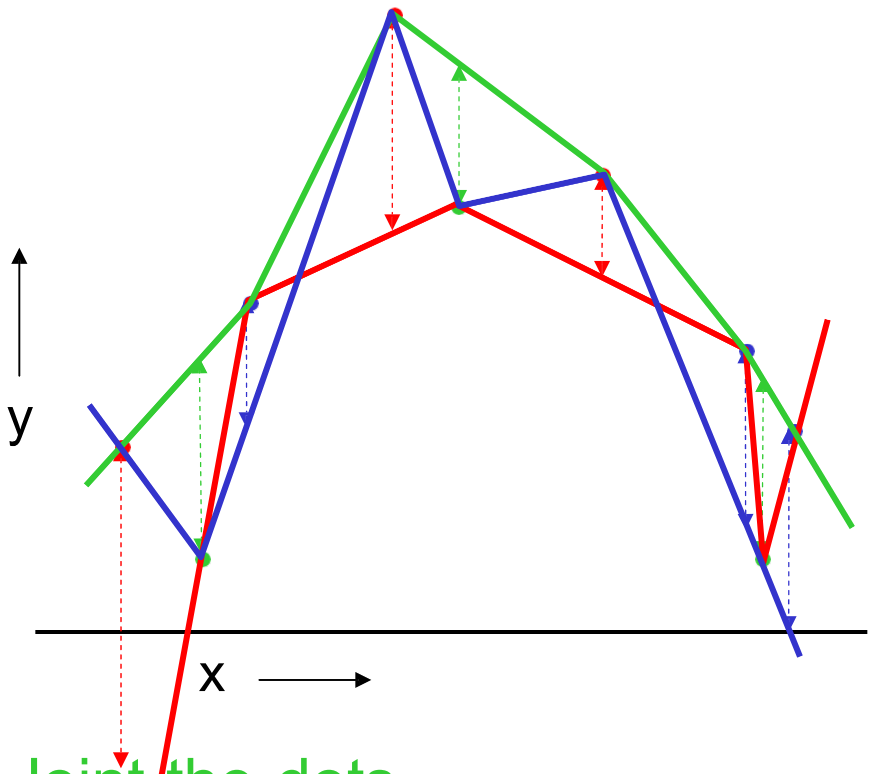
For the red partition: Train on all the points not in the red partition. Find the test-set sum of errors on the red points.

For the green partition: Train on all the points not in the green partition. Find the test-set sum of errors on the green points.

For the blue partition: Train on all the points not in the blue partition. Find the test-set sum of errors on the blue points.

Then report the mean error

K-fold Cross Validation



Joint-the-dots

$MSE_{3FOLD}=2.93$

Randomly break the dataset into k partitions (in our example we'll have k=3 partitions colored Red Green and Blue)

For the red partition: Train on all the points not in the red partition. Find the test-set sum of errors on the red points.

For the green partition: Train on all the points not in the green partition. Find the test-set sum of errors on the green points.

For the blue partition: Train on all the points not in the blue partition. Find the test-set sum of errors on the blue points.

Then report the mean error

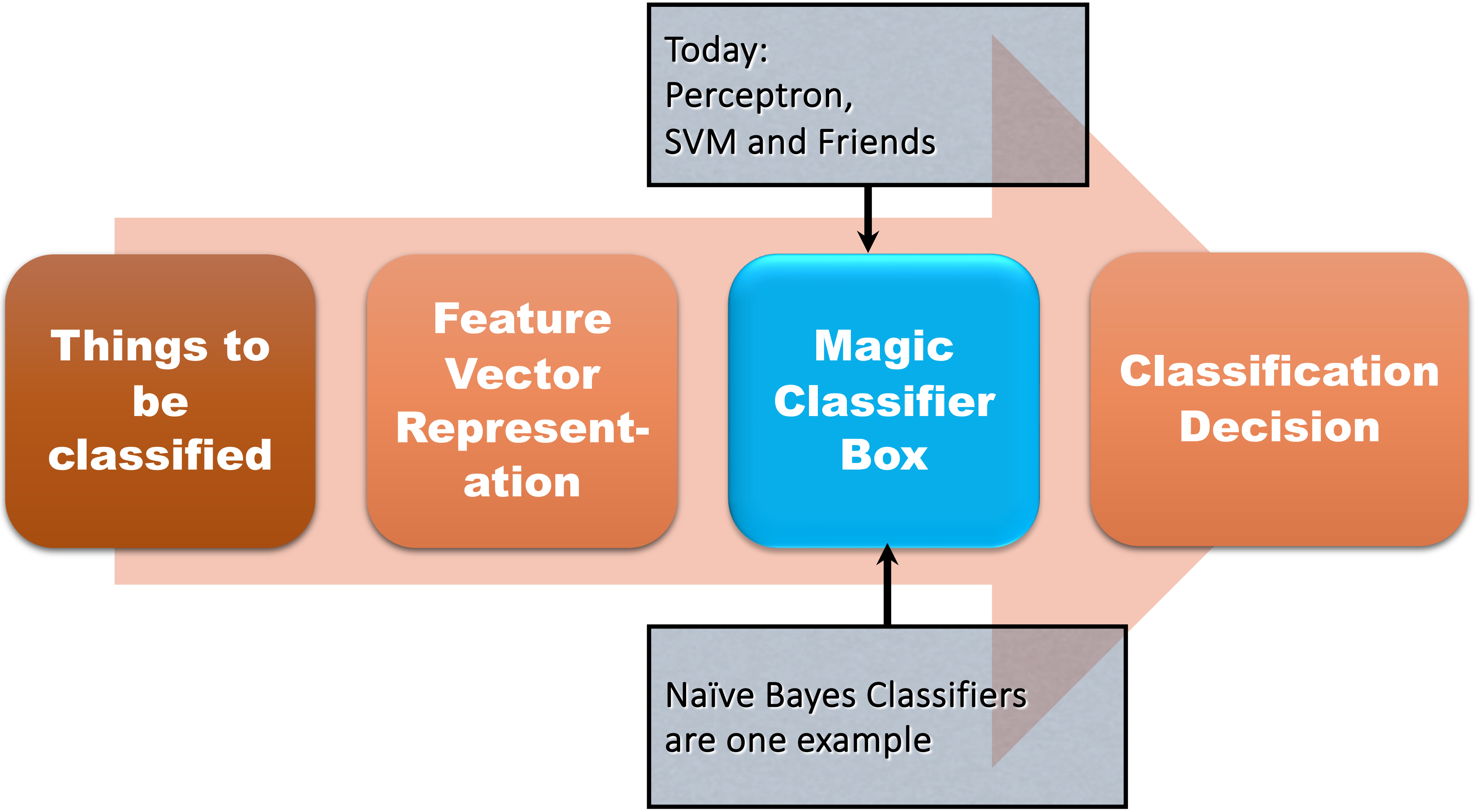
What kind of Cross Validation?

	Downside	Upside
Test-set	Variance: unreliable estimate of future performance	Cheap
Leave-one-out	Expensive. Has some weird behavior	Doesn't waste data
10-fold	Wastes 10% of the data. 10 times more expensive than test set	Only wastes 10%. Only 10 times more expensive instead of R times.
3-fold	Wastier than 10-fold. More Expensive than test set	Slightly better than test-set
R-fold	Identical to Leave-one-out	

Is that a word?

Perceptrons and Neural Nets

Universal Machine Learning Diagram



Generative and Discriminative Models

- *Generative question*:
  - “How can we model the joint distribution of the classes and the features?”
  - Naïve Bayes, Markov Models, HMMs all generative

- *Discriminative question*:
  - “What features *distinguish* the classes from one another?”

Recap

Naïve Bayes: generative classifier

- Need to specify features ahead of time
- Parameters / weights directly estimated from corpus

- Logistic Regression: discriminative classifier
  - Need to specify features ahead of time
  - Parameters / weights learned iteratively
  - Specified particular function (sigmoid) to convert z values into probabilities, handle non-linear input

- Neural Networks:
  - Glue together many classifiers
  - Allow many different non-linear function transformations
  - **Learn both the features and weights iteratively**

Logistic Regression

Logistic regression solves this task by learning, from a training set, a vector of **weights** and a **bias term**.

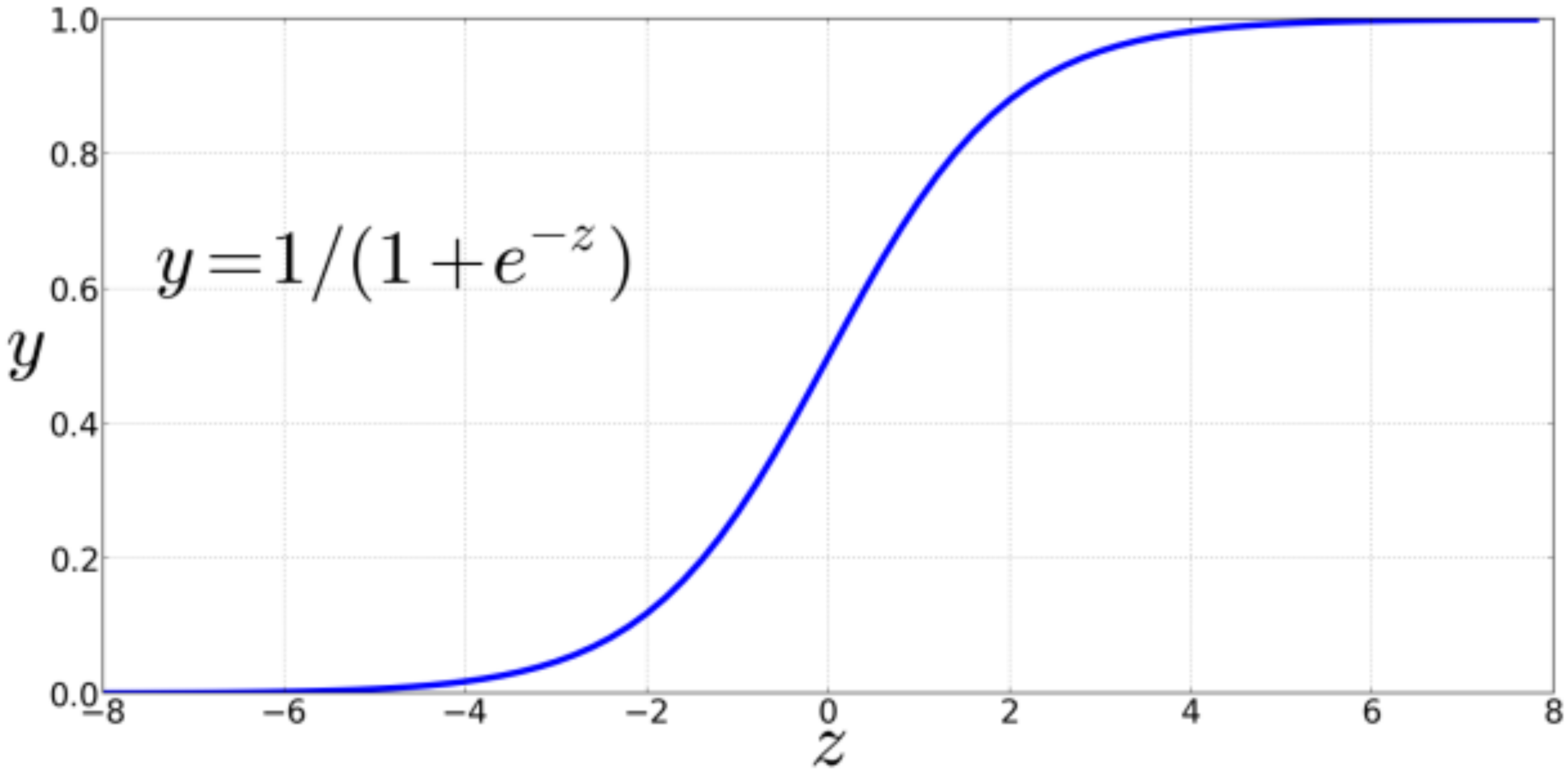
$$z = \left(\sum_{i=1}^{n} w_i x_i\right) + b$$

We can also write this as a dot product:

$$\boxed{z} = w \cdot x + b$$

This is a real number, not a probability!

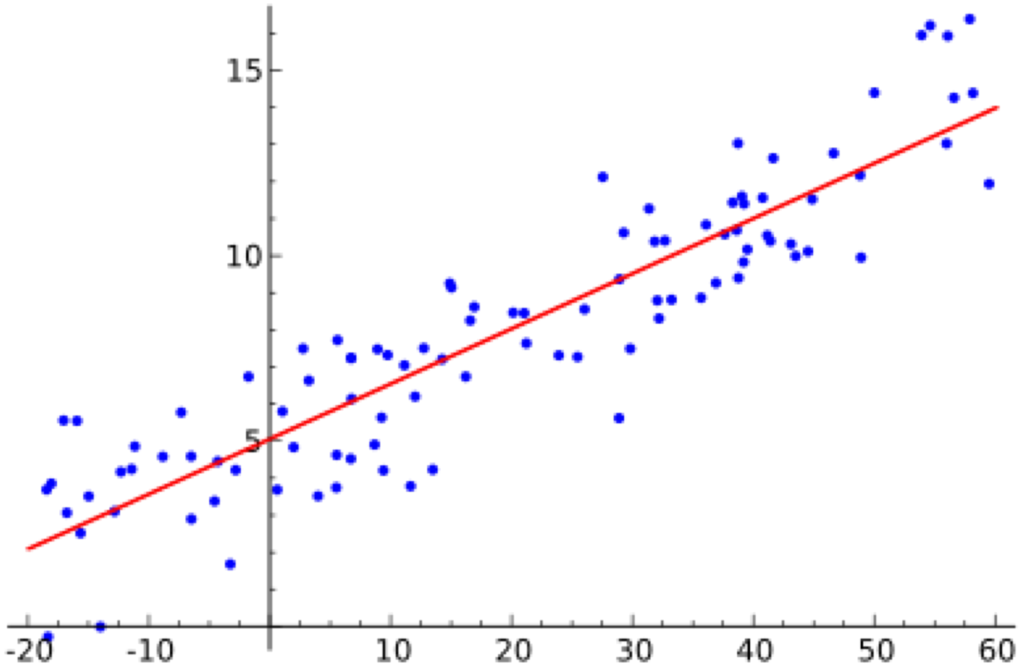
Sigmoid



But without the sigmoid it's just a linear function

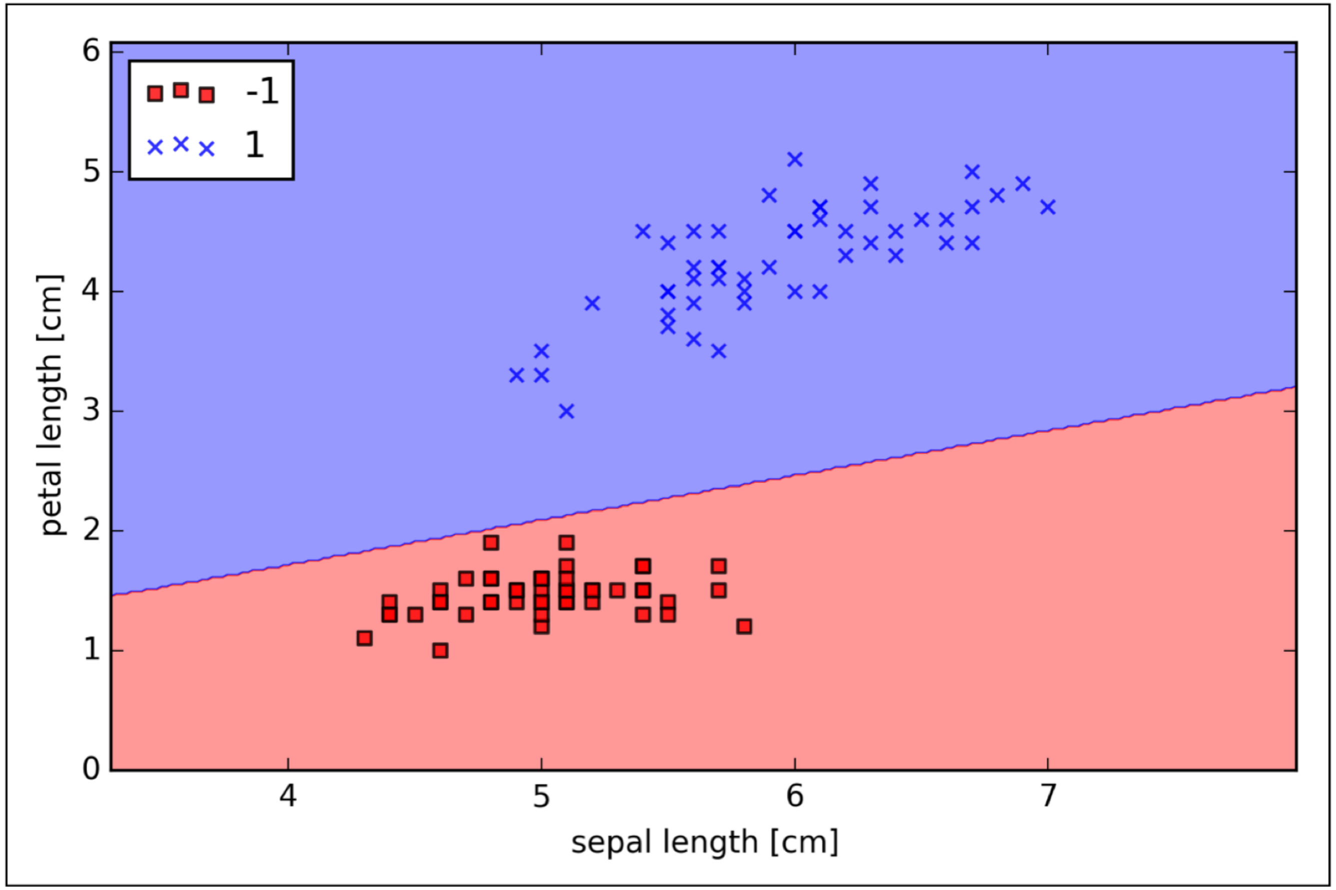
- Regressing a line

$$z = \left(\sum_{i=1}^{n} w_i x_i\right) + b$$

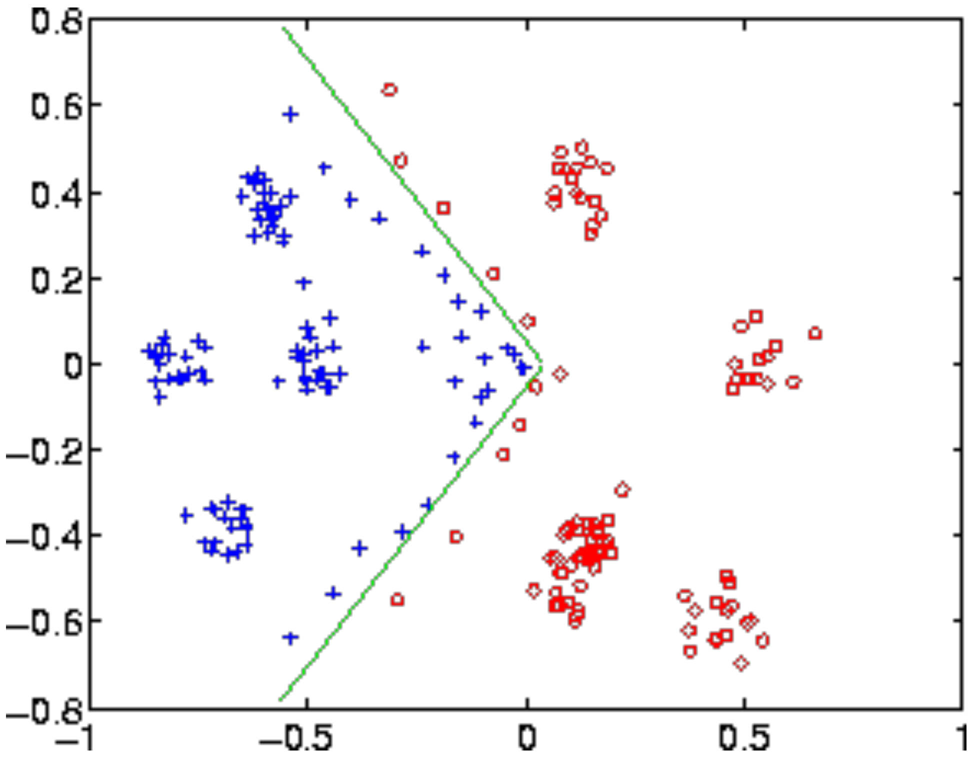


But without the sigmoid it's just a linear function

- Classification:



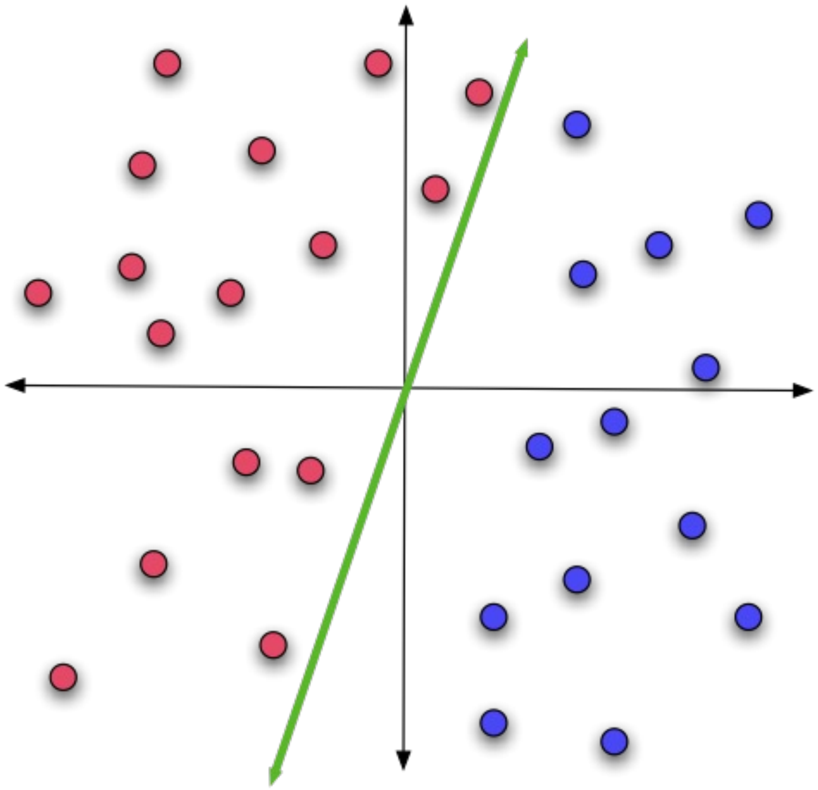
Generative vs. Discriminative: Visual Example



Modeling what sort of bizarre distribution produced these training points is hard, but distinguishing the classes is a piece of cake! *chart from MIT tech report #507, Tony Jebara*

Linear Classification: Informal…

Find a (line, plane, hyperplane) that divides the red points from the blue points….



Hyperplane

- Just a subspace whose dimension is one less than that of its containing space.

- If the containing space is 3-dimensional, then its hyperplanes are 2-d
- If the containing space is 2-dimensional, then its hyperplanes are 1-d (lines)

Linear Classification: Slightly more formal

Input encoded as feature vector $\vec{x}$

Model encoded as $\vec{w}$

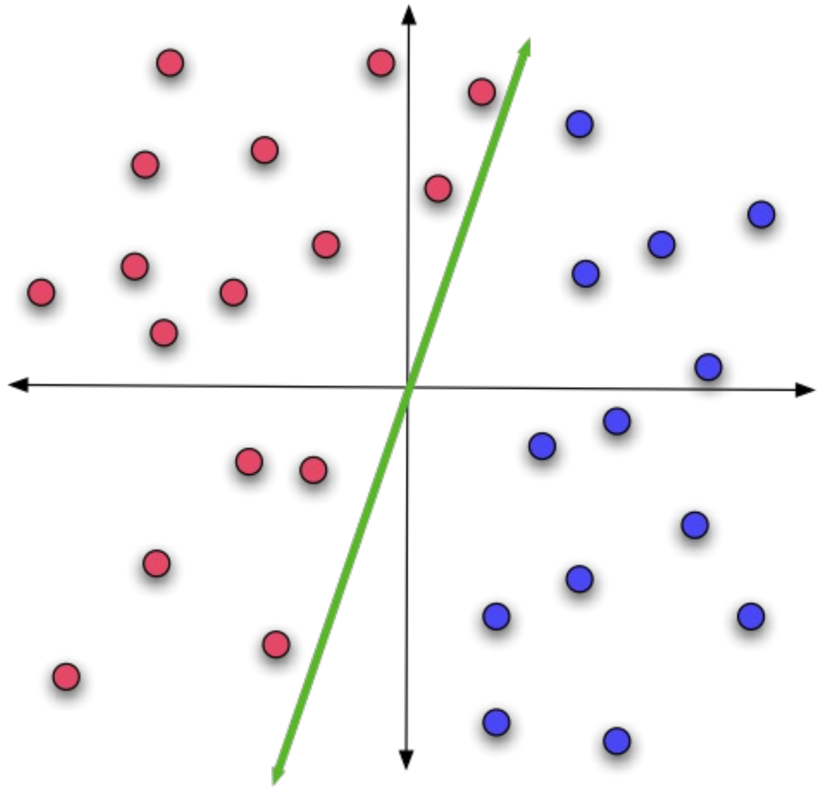
Just return $y = \vec{w} \cdot \vec{x}$!

sign(y) tell us the class:

+ - blue

- - red

(Vectors normalized to length 1, and we assume that the hyperplane passes through 0,0)



Linear Classification: Slightly more formal

Input encoded as feature vector $\vec{x}$

Model encoded as $\vec{w}$

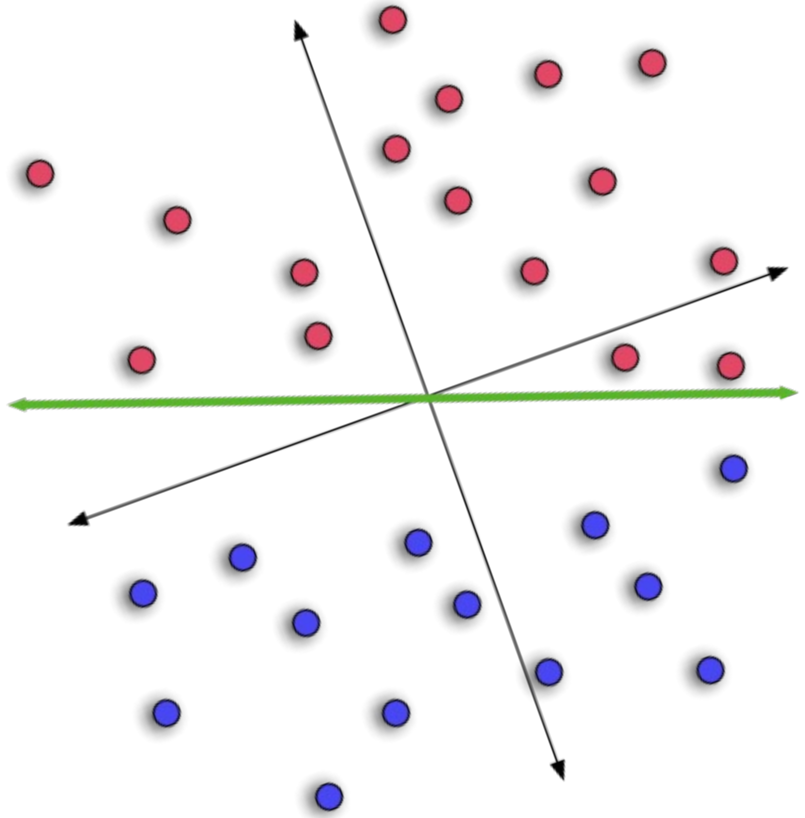
Just return $y = \vec{w} \cdot \vec{x}$!

sign(y) tell us the class:

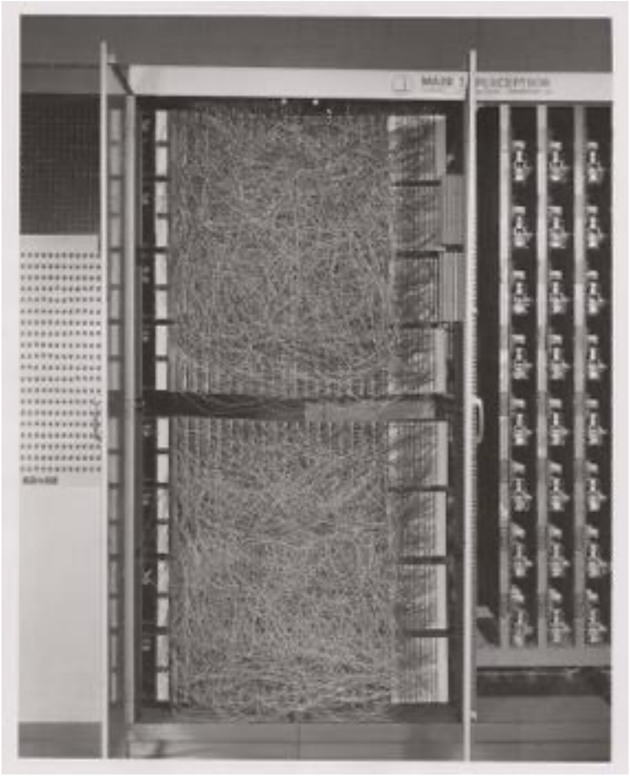
+ - blue

- - red

(Vectors normalized to length 1, and we assume that the hyperplane passes through 0,0)



Perceptron



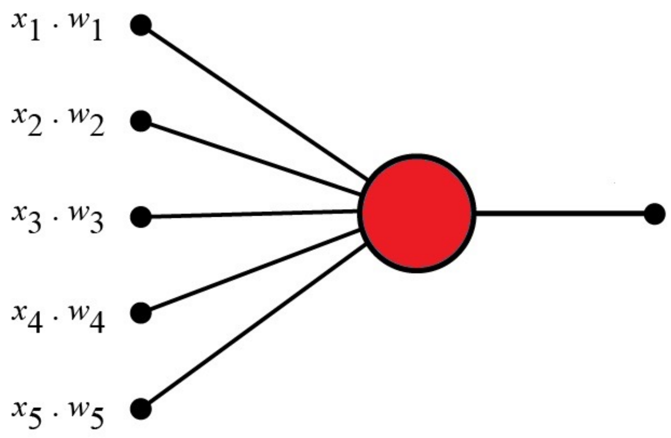
Perceptron is an algorithm for binary classification that uses a linear prediction function:

$$f(\mathbf{x}) = \begin{cases} 1, & \mathbf{w}^{*}\mathbf{x} + b \geq 0 \\ -1, & \mathbf{w}^{*}\mathbf{x} + b < 0 \end{cases}$$

This is a *step function*, which reads:

- the output is 1 if "$\mathbf{w}^{*}\mathbf{x} + b \geq 0$" is true, and the output is -1 if instead "$\mathbf{w}^{*}\mathbf{x} + b < 0$" is true

Perceptron

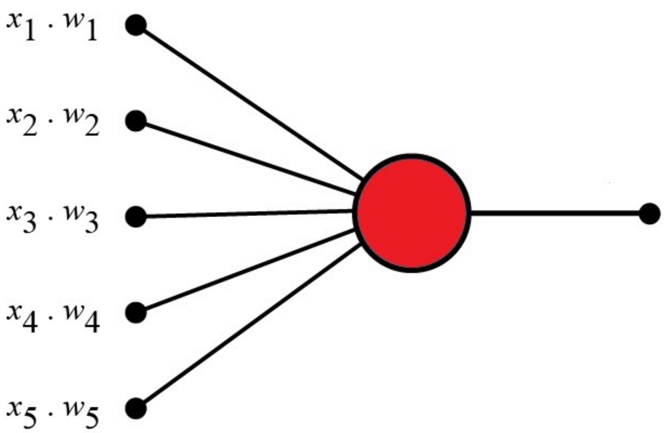


Perceptron is an algorithm for binary classification that uses a linear prediction function:

$$f(\mathbf{x}) = \begin{cases} 1, & \mathbf{w}^{*}\mathbf{x} + b \geq 0 \\ -1, & \mathbf{w}^{*}\mathbf{x} + b < 0 \end{cases}$$

By convention, the two classes are +1 or -1.

Perceptron



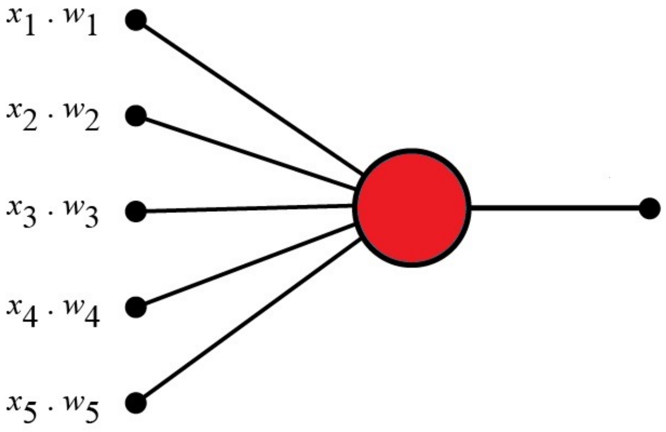
Perceptron is an algorithm for binary classification that uses a linear prediction function:

$$f(\mathbf{x}) = \begin{cases} 1, & \mathbf{w}^{*}\mathbf{x} + b \geq 0 \\ -1, & \mathbf{w}^{*}\mathbf{x} + b < 0 \end{cases}$$

By convention, ties are broken in favor of the positive class.

- If "$\mathbf{w}^{*}\mathbf{x}$ + b" is exactly 0, output +1 instead of -1.

Perceptron

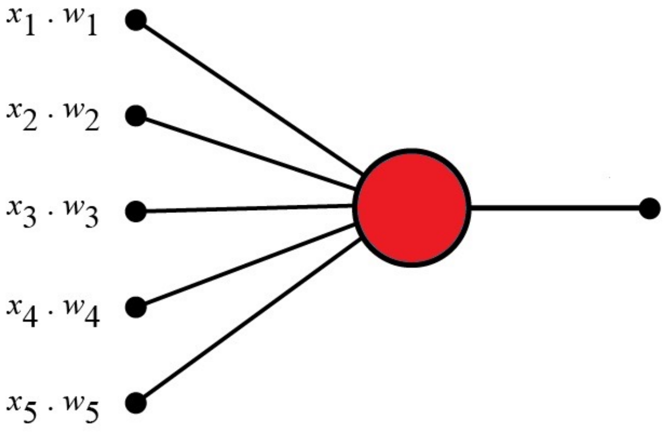


The **w** parameters are unknown. This is what we have to learn.

$$f(\mathbf{x}) = \begin{cases} 1, & \mathbf{w}^{*}\mathbf{x} + b \geq 0 \\ -1, & \mathbf{w}^{*}\mathbf{x} + b < 0 \end{cases}$$

In the same way that linear regression learns the slope parameters to best fit the data points, perceptron learns the parameters to best separate the instances.

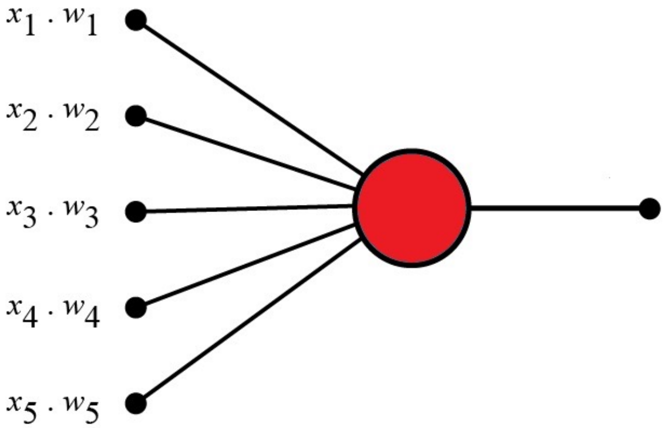
Perceptron: Learning the Weights



The perceptron algorithm learns the weights by:

1. Initialize all weights **w** to 0
2. Iterate through the training data. For each training instance, classify the instance.
   a) If the prediction (the output of the classifier) was correct, don't do anything. (It means the classifier is working, so leave it alone!)
   b) If the prediction was wrong, modify the weights by using the **update rule**.
3. Repeat step 2 some number of times (more on this later).

Perceptron: Learning the Weights



What does an **update rule** do?

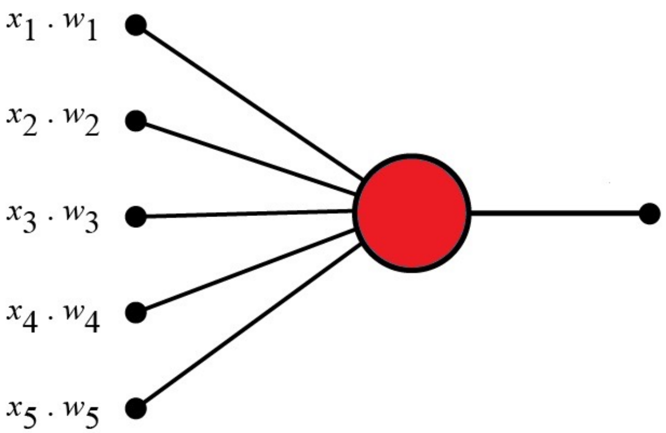
- If the classifier predicted an instance was negative but it should have been positive…

  Currently: $\mathbf{w}^{*}\mathbf{x_i} + b < 0$

  Want: $\mathbf{w}^{*}\mathbf{x_i} + b \geq 0$

  - Adjust the weights **w** so that this function value moves toward positive
- If the classifier predicted positive but it should have been negative, shift the weights so that the value moves toward negative.

Perceptron: Learning the Weights



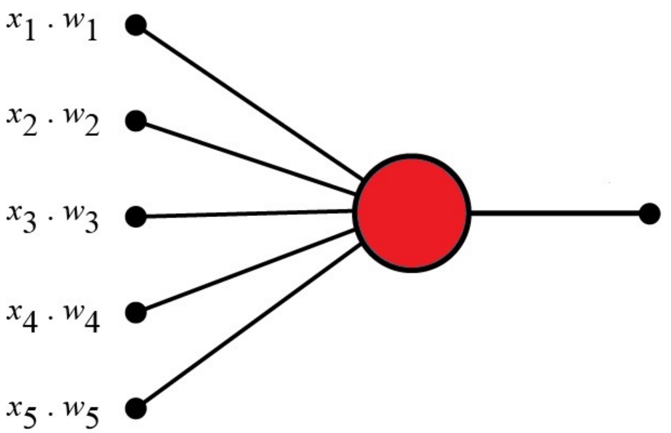
The perceptron update rule:

$w_j \mathrel{+}= (y_i - f(x_i))\ x_{ij}$

w_j	The weight of feature j
y_i	The true label of instance i
x_i	The feature vector of instance i
$f(x_i)$	The class prediction for instance i
x_{ij}	The value of feature j in instance i

Let's assume x_{ij} is 1 in this example for now.

Perceptron: Learning the Weights



The perceptron update rule:

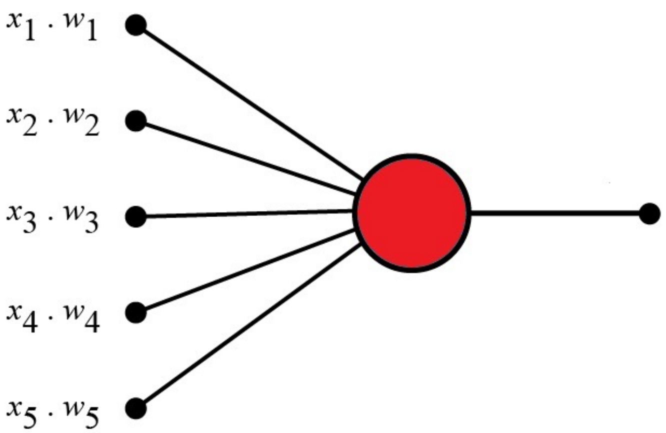
$w_j \mathrel{+}= (y_i - f(x_i))\ x_{ij}$

w_j	The weight of feature j
y_i	The true label of instance i
x_i	The feature vector of instance i
$f(x_i)$	The class prediction for instance i
x_{ij}	The value of feature j in instance i

This term is 0 if the prediction was correct ($y_i = f(x_i)$).

- Then the entire update rule is 0, so no change is made.

Perceptron: Learning the Weights



The perceptron update rule:

$w_j \mathrel{+}= (y_i - f(x_i))\ x_{ij}$

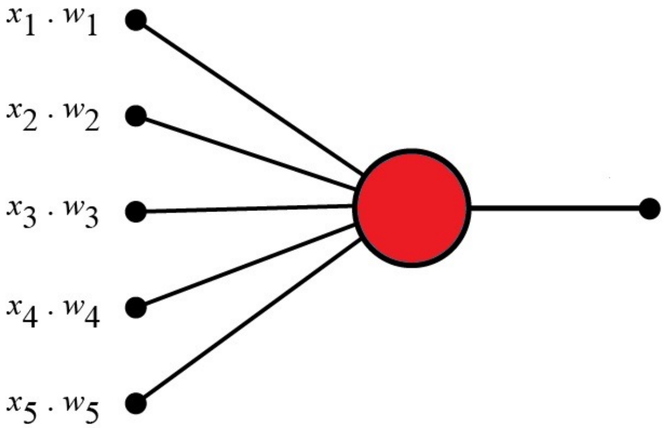
w_j	The weight of feature j
y_i	The true label of instance i
x_i	The feature vector of instance i
$f(x_i)$	The class prediction for instance i
x_{ij}	The value of feature j in instance i

If the prediction is wrong:

- This term is +2 if $y_i = +1$ and $f(x_i) = -1$.
- This term is -2 if $y_i = -1$ and $f(x_i) = +1$.

The *sign* of this term indicates the direction of the mistake.

Perceptron: Learning the Weights



The perceptron update rule:

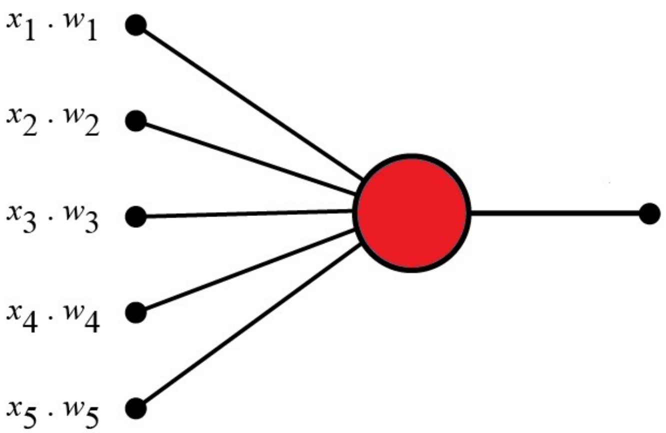
$w_j \mathrel{+}= (y_i - f(x_i))\ x_{ij}$

w_j	The weight of feature j
y_i	The true label of instance i
x_i	The feature vector of instance i
$f(x_i)$	The class prediction for instance i
x_{ij}	The value of feature j in instance i

If the prediction is wrong:

- The $(y_i - f(x_i))$ term is +2 if $y_i = +1$ and $f(x_i) = -1$.
  - This will increase w_j (still assuming x_{ij} is 1)…
  - …which will increase $\mathbf{w}^*\mathbf{x_i} + b$…
  - …which will make it more likely $\mathbf{w}^*\mathbf{x_i} + b \geq 0$ next time (which is what we need for the classifier to be correct)

Perceptron: Learning the Weights



The perceptron update rule:

$w_j \mathrel{+}= (y_i - f(x_i))\ x_{ij}$

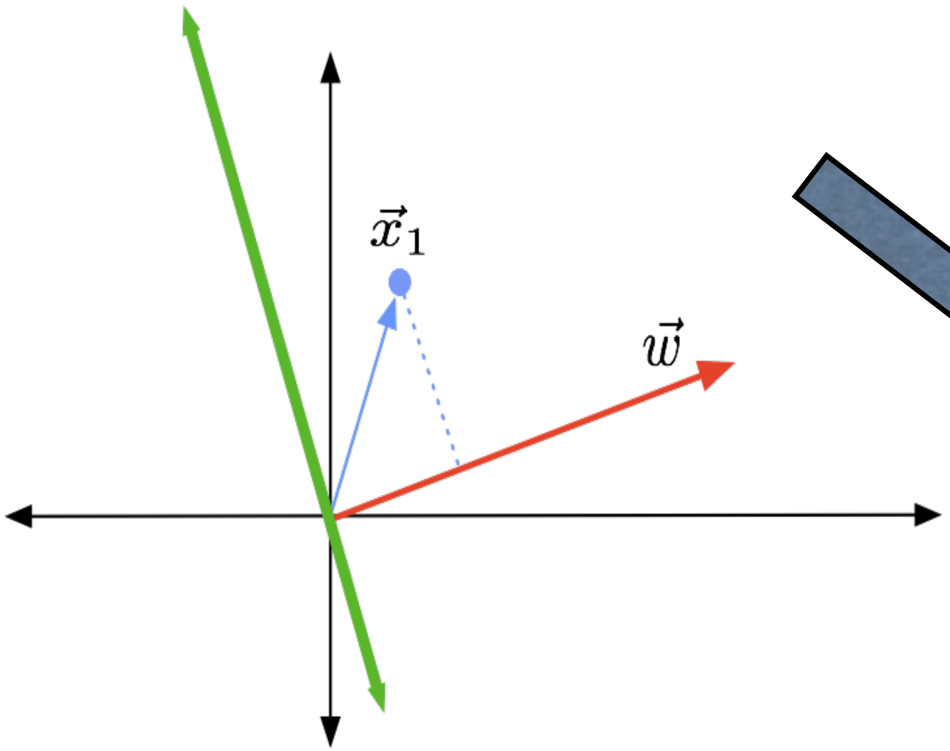
w_j	The weight of feature j
y_i	The true label of instance i
x_i	The feature vector of instance i
$f(x_i)$	The class prediction for instance i
x_{ij}	The value of feature j in instance i

If x_{ij} is 0, there will be no update.

- The feature does not affect the prediction for this instance, so it won't affect the weight updates.

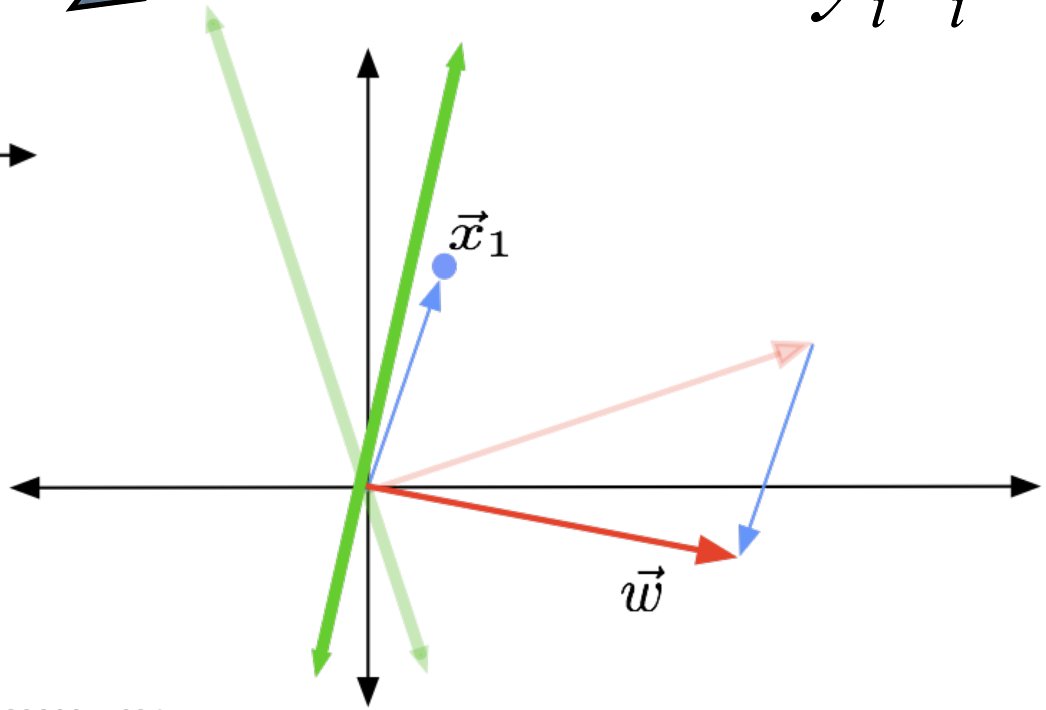
If x_{ij} is negative, the sign of the update flips.

Perceptron Update Example



If $\vec{x}_1$ is supposed to be on the other side....

$$\vec{w} = \vec{w} + y_i \vec{x}_i$$



Perceptron Learning Algorithm

Input: A list T of training examples $\langle \vec{x}_0, y_0 \rangle \ldots \langle \vec{x}_n, y_n \rangle$ where
$\forall i : y_i \in \{+1, -1\}$

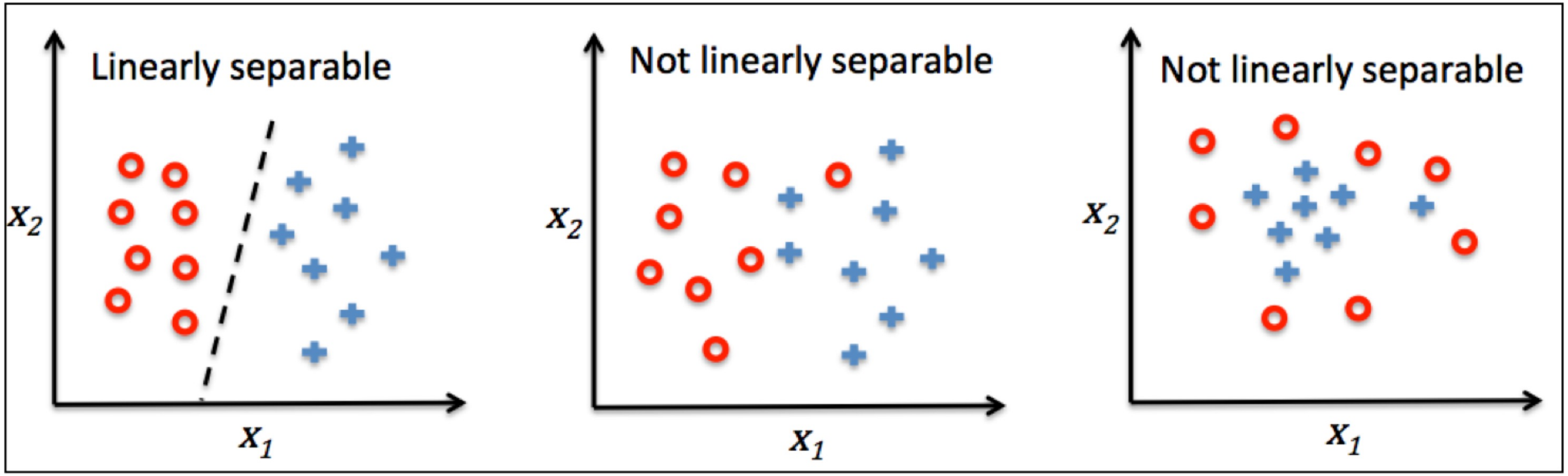
Output: A classifying hyperplane $\vec{w}$

Randomly initialize $\vec{w}$;

while *model* $\vec{w}$ *makes errors on the training data* **do**
  for $\langle \vec{x}_i, y_i \rangle$ *in* T **do**
    Let $\hat{y} = sign(\vec{w} \cdot \vec{x}_i)$;
    if $\hat{y} \neq y_i$ **then**
      $\vec{w} = \vec{w} + y_i \vec{x}_i$;
    end
  end
end

Linear Separability

The training instances are **linearly separable** if there exists a hyperplane that will separate the two classes.



Linear Separability

If the training instances are not linearly separable, the classifier will always get some predictions wrong.

- You need to implement some type of **stopping criteria** for when the algorithm will stop making updates, or it will run forever.
- Usually this is specified by running the algorithm for a maximum number of **iterations** or **epochs**.

Learning Rate

Let's make a modification to the update rule:

$$w_j \mathrel{+}= \eta \, (y_i - f(x_i)) \; x_{ij}$$

where η is called the **learning rate** or **step size**.

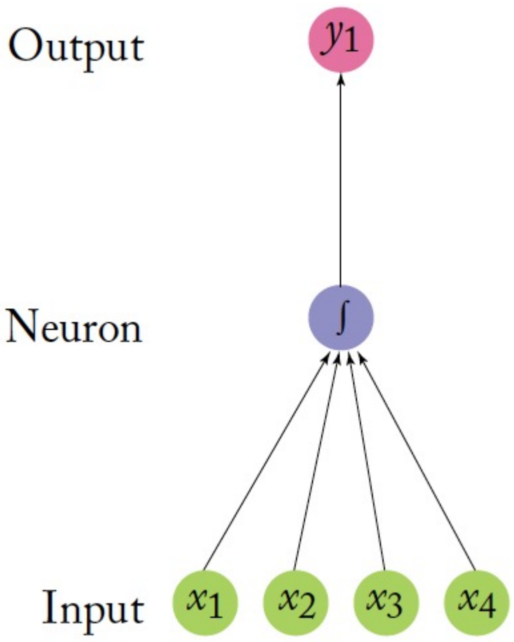
- When you update w_j to be more positive or negative, this controls the size of the change you make (or, how large a "step" you take).
- If η=1 (a common value), then this is the same update rule from the earlier slide.

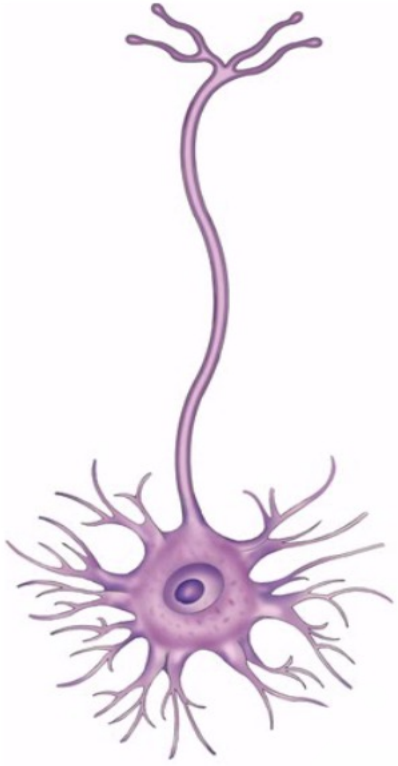
Learning Rate

- How to choose the step size?
- If η is too small, the algorithm will be slow because the updates won't make much progress.
- If η is too large, the algorithm will be slow because the updates will "overshoot" and may cause previously correct classifications to become incorrect.

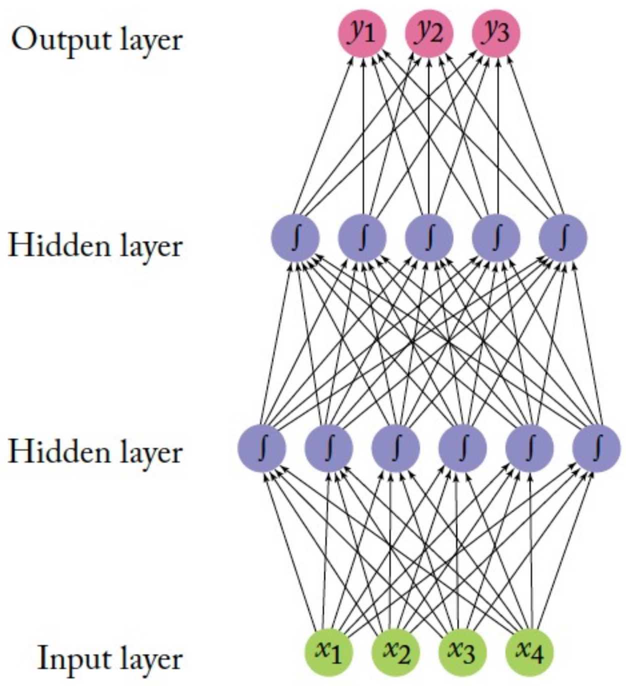
Perceptrons

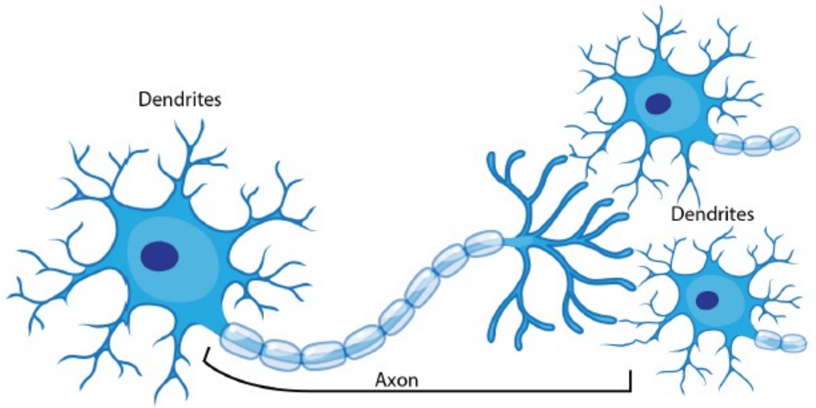
A single neuron



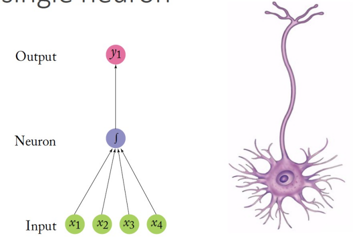


Neural networks





A single neuron

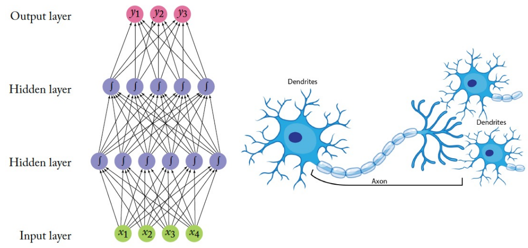


The simplest neural network is called a perceptron. It is simply a linear model:

$$\mathrm{NN}_{\mathrm{Perceptron}}(\boldsymbol{x}) = \boldsymbol{x}\boldsymbol{W} + \boldsymbol{b}$$

$$\boldsymbol{x} \in \mathbb{R}^{d_{in}}, \quad \boldsymbol{W} \in \mathbb{R}^{d_{in} \times d_{out}}, \quad \boldsymbol{b} \in \mathbb{R}^{d_{out}}$$

where W is the weight matrix and b is a bias term.



Neural Networks

To go beyond linear function, we introduce a non-linear hidden layer.
The result is called a Multi-Layer Perceptron with one hidden layer.

$$\mathrm{NN}_{\mathrm{MLP1}}(\boldsymbol{x}) = g(\boldsymbol{x}\boldsymbol{W}^{\mathbf{1}} + \boldsymbol{b}^{\mathbf{1}})\boldsymbol{W}^{\mathbf{2}} + \boldsymbol{b}^{\mathbf{2}}$$

$$\boldsymbol{x} \in \mathbb{R}^{d_{in}}, \quad \boldsymbol{W}^{\mathbf{1}} \in \mathbb{R}^{d_{in} \times d_1}, \quad \boldsymbol{b}^{\mathbf{1}} \in \mathbb{R}^{d_1}, \quad \boldsymbol{W}^{\mathbf{2}} \in \mathbb{R}^{d_1 \times d_2}, \quad \boldsymbol{b}^{\mathbf{2}} \in \mathbb{R}^{d_2}$$

Here $\mathbf{W}^1$ and $\mathbf{b}^1$ are a matrix and a bias for the **first** linear transformation of the input **x**,

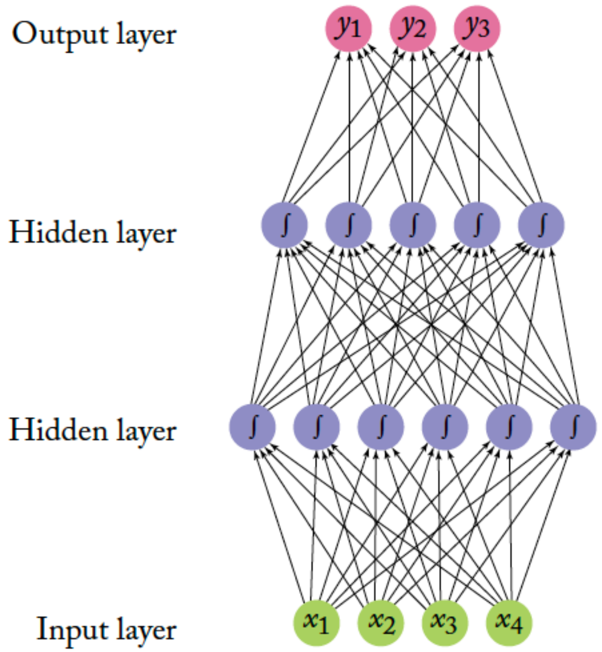
g is a nonlinear function (also an activation function),

$\mathbf{W}^2$ and $\mathbf{b}^2$ are the matrix and bias term for a **second** linear transform.

Neural Networks

We can add additional linear transformations and nonlinearities, resulting with a MLP with two hidden layers:

$$\mathrm{NN}_{\mathrm{MLP2}}(\boldsymbol{x}) = (g^2(g^1(\boldsymbol{x}\boldsymbol{W}^1 + \boldsymbol{b}^1)\boldsymbol{W}^2 + \boldsymbol{b}^2))\boldsymbol{W}^3.$$



Same equation, but written with intermediary variables:

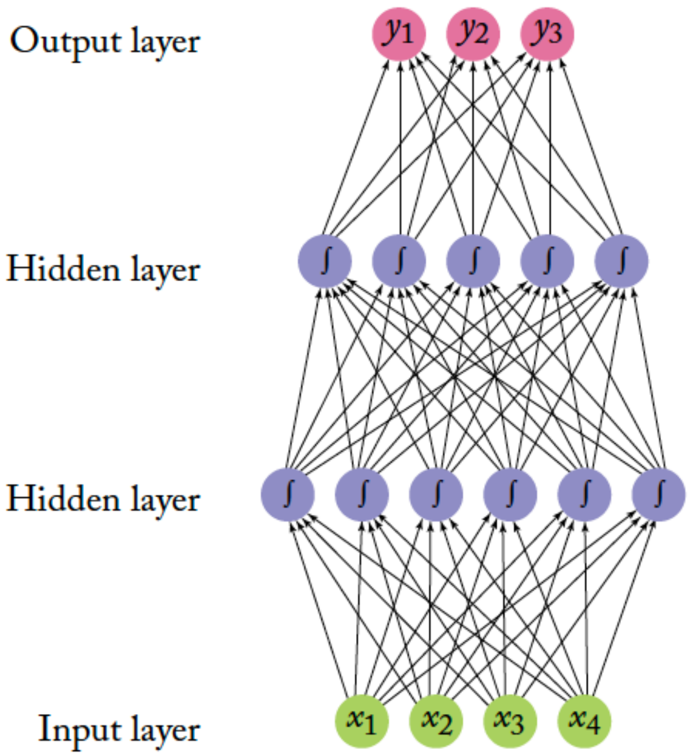
$$\begin{aligned} \mathrm{NN}_{\mathrm{MLP2}}(\boldsymbol{x}) &= \boldsymbol{y} \\ \boldsymbol{h}^1 &= g^1(\boldsymbol{x}\boldsymbol{W}^1 + \boldsymbol{b}^1) \\ \boldsymbol{h}^2 &= g^2(\boldsymbol{h}^1\boldsymbol{W}^2 + \boldsymbol{b}^2) \\ \boldsymbol{y} &= \boldsymbol{h}^2\boldsymbol{W}^3. \end{aligned}$$

Neural Networks

A Multi-Layer Perceptron with one hidden layer is a “universal approximator”.

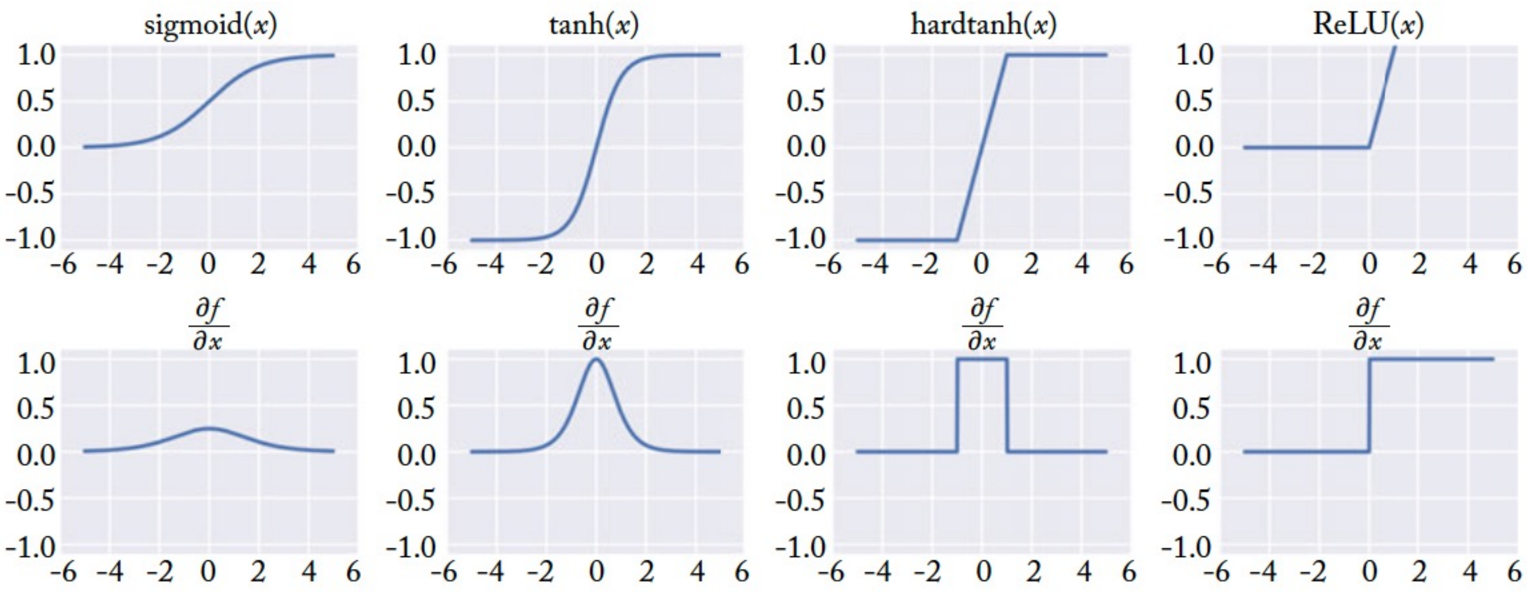
It can approximate a family of functions that includes all continuous functions on a closed and bounded subset of R^n

It can approximate any function mapping from any finite dimensional discrete space to another.



Many complications

Which non-linear function to use?



Stochastic gradient descent is more complicated than the perceptron learning algorithm

What features to use? (Width)

How many hidden layers to use? (Depth)

Recap

Naïve Bayes: generative classifier
- Need to specify features ahead of time
- Parameters / weights directly estimated from corpus

- Logistic Regression: discriminative classifier
  - Need to specify features ahead of time
  - Parameters / weights learned iteratively
  - Specified particular function (sigmoid) to convert z values into probabilities, handle non-linear input

- Neural Networks:
  - Glue together many perceptrons
  - Allow many different non-linear function transformations
  - **Learn both the features and weights iteratively**