

CS 457/557: Machine Learning

Lecture 01-1: Introduction to Machine Learning

Introduction to Machine Learning

What is Machine Learning?

Definition

Machine learning is the science of getting computers to act without being explicitly programmed. (from Coursera Machine Learning course)

Applications:

- Speech recognition
("Okay Google", etc.)
- Image classification
- Self-driving cars
- Watson, DeepMind



Good motivating discussion:

https://d21.ai/chapter_introduction/index.html

Machine Learning in Context

Machine learning is a **subfield** of **artificial intelligence**:

Deep Learning ⊂ Machine Learning ⊂ Artificial Intelligence

Definition

Artificial intelligence is “the study of the conjecture that every aspect of learning or any other feature of intelligence can in principle be so precisely described that a machine can be made to simulate it.”
(Dartmouth Workshop on AI, 1956)

Definition

Deep learning is a set of machine learning methods based on artificial neural networks with multiple network layers.

Very active area of research and development right now!

Defining Intelligence

What is **intelligence**?

- Turing test: Intelligence = Acting Humanly
- Rationality: Doing the “right thing”

Definition

An **intelligent system (agent)** is one that acts so as to achieve the best (expected) outcome.

A notion of **best** implies some **measure of quality**. Examples:

- Number of cat pictures correctly identified
- Hours of real-time driving without accidents
- Number of games of chess won
- ...

Defining Learning

Definition

An intelligent system is **learning** if it improves its performance after making observations about the world.

Roughly speaking, learning is the process of converting experience into expertise or knowledge.

— Shalev-Shwartz and Ben-David, *Understanding Machine Learning*

A **learning problem** involves three basic components:

- Task T
- Performance measure P
- Experience E

We say that a machine learns with respect to a particular task T , performance metric P , and type of experience E , if the system reliably improves its performance P at task T , following experience E .

— Mitchell, *The Discipline of Machine Learning*

Example Learning Problem

Example: recognizing wake-words (e.g., “Hey Siri”, “Okay Google”):

- Task: Take system action based on speech
- Performance: How often correct action is taken during testing
- Experience: ???

The experience is typically described by **data**, represented as a set of **input-output** pairs:

$$\{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \dots, (\mathbf{x}_n, y_n)\} \quad (\text{also denoted } \{(\mathbf{x}_i, y_i)\}_{i=1}^n).$$

- \mathbf{x}_i is a **vector** of input (feature / attribute) values
 - E.g., \mathbf{x}_i could store various properties of a sound wave recording
- y_i represents a scalar output value (response)
 - E.g., $y_i = 1$ if the recording is “Okay Google” and 0 otherwise

(Note: textbook uses N instead of n for number of input-output pairs)

Another Approach: Supervised Learning

- ▶ Collect a large set of sample things a set of test users say to our system
- ▶ For each, map it to a correct outcome action the system should take

“Call my wife”

“Set an alarm for 4:00 AM”

“Play Pod Save America”

...

call(555-123-4567)

alarm_set(04:00)

podcast_play(“Pod
Save America”)

...

- ▶ A large set of such (*speech, action*) pairs can be created
- ▶ This can then form the **experience**, E , the system needs

Inductive Learning

How can we **learn** from the experience captured by $\{(\mathbf{x}_i, y_i)\}_{i=1}^n$?

By **assuming** that there is **some functional relationship** between the inputs and outputs, typically given by

$$y_i = f(\mathbf{x}_i) \quad \text{or} \quad y_i = f(\mathbf{x}_i) + \epsilon_i, \quad \text{where } \epsilon_i \text{ represents noise.}$$

Definition

Induction (inductive reasoning) is the process of extracting a general pattern from a specific set of examples.

In the simplest form, **induction** is the task of learning an **unknown function** on some inputs from **examples** of its outputs.

Types of ML: Supervised Learning

Supervised Learning Problem

Given a *labeled* data set $\{(\mathbf{x}_i, y_i)\}_{i=1}^n$ with $y = f(\mathbf{x})$ for some unknown function f , identify a **hypothesis function** h that approximates f well (i.e., $h(\mathbf{x}) \approx f(\mathbf{x})$ for all \mathbf{x}).

- f captures **how the world actually works**
- h captures **how we think the world works**

Remaining questions:

- What hypothesis functions should be considered?
- What makes one hypothesis function better than another?
- How do we find a **good** hypothesis function from the available options?

CS 457/557: Machine Learning

Lecture 01-2: Introduction to Machine Learning (Online: #02)

Types of ML: Supervised Learning

Supervised Learning Problem

Given a *labeled* data set $\{(\mathbf{x}_i, y_i)\}_{i=1}^n$ with $y = f(\mathbf{x})$ for some unknown function f , identify a **hypothesis function** h that approximates f well (i.e., $h(\mathbf{x}) \approx f(\mathbf{x})$ for all \mathbf{x}).

The **supervised learning process**:

- ① Start with a labeled data set $\{(\mathbf{x}_i, y_i)\}_{i=1}^n$ (set of input-output pairs)
- ② Choose a **hypothesis space (model class)** \mathcal{H} representing the set of all possible hypotheses h to consider
- ③ Use a **learning algorithm** to find a **hypothesis (model)** $h^* \in \mathcal{H}$ such that $h^*(\mathbf{x}_i) \approx y_i$ for all $i \in \{1, 2, \dots, n\}$
- ④ (Optional) Use h^* to **predict** outputs for other inputs!

Some Truth in Advertising

Let's start by telling the truth: machines don't learn. What a typical "learning machine" does, is finding a mathematical formula, which, when applied to a collection of inputs (called "training data"), produces the desired outputs. This mathematical formula also generates the correct outputs for most other inputs (distinct from the training data) on the condition that those inputs come from the same or a similar statistical distribution as the one the training data was drawn from.

Why isn't that learning? Because if you slightly distort the inputs, the output is very likely to become completely wrong. It's not how learning in animals works. If you learned to play a video game by looking straight at the screen, you would still be a good player if someone rotates the screen slightly. A machine learning algorithm, if it was trained by "looking" straight at the screen, unless it was also trained to recognize rotation, will fail to play the game on a rotated screen.

So why the name "machine learning" then? The reason, as is often the case, is marketing: Arthur Samuel, an American pioneer in the field of computer gaming and artificial intelligence, coined the term in 1959 while at IBM. Similarly to how in the 2010s IBM tried to market the term "cognitive computing" to stand out from competition, in the 1960s, IBM used the new cool term "machine learning" to attract both clients and talented employees.

As you can see, just like artificial intelligence is not intelligence, machine learning is not learning. However, machine learning is a universally recognized term that usually refers to the science and engineering of building machines capable of doing various useful things without being explicitly programmed to do so. So, the word "learning" in the term is used by analogy with the learning in animals rather than literally.

— Andriy Burkov, Preface to *The Hundred-Page Machine Learning Book*

Decisions to Make in Supervised Learning

In practice, we often need to make numerous decisions when formulating the data set $\{(\mathbf{x}_i, y_i)\}_{i=1}^n$.

Example: Medical Informatics

- Data set contains information about patients
- The **features** of the data set include things like age, height, weight, gene data, disease status, life span (if known), etc.

What do we want to **learn** from this data?

How we answer this influences:

- which features are used as inputs and which feature is used as output
- our choice of hypothesis space (model class)
- the machine learning algorithms that we can apply
- the measurements of success that we can use

One Approach: Regression

- ▶ We decide that we want to try to learn to predict how long patients will live
- ▶ We base this upon information about the degree to which they express a specific gene
- ▶ A **regression problem**: the function we learn is the “best (linear) fit” to the data we have

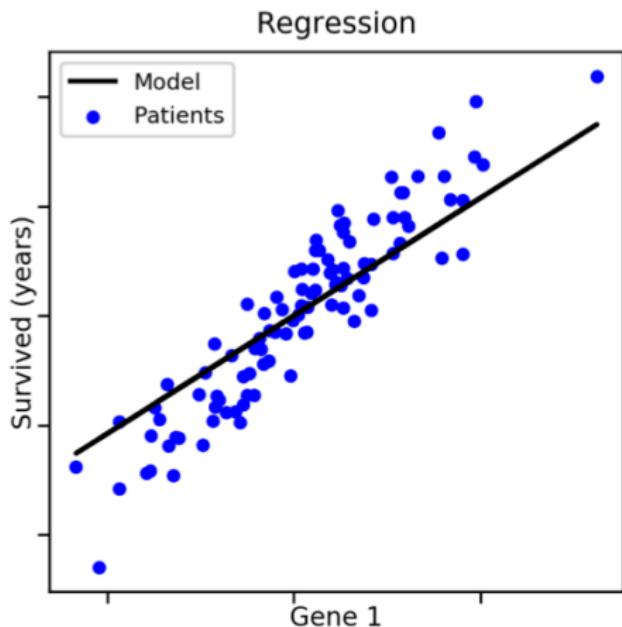


Image source: <https://aldro61.github.io/microbiome-summer-school-2017/sections/basics/>

Another Approach: Classification

- ▶ We decide instead that we simply want to decide whether a patient will get the disease or not
- ▶ We base this upon information about expression of two genes
- ▶ A **classification problem**: learned function separates individuals into 2 groups (binary classes)

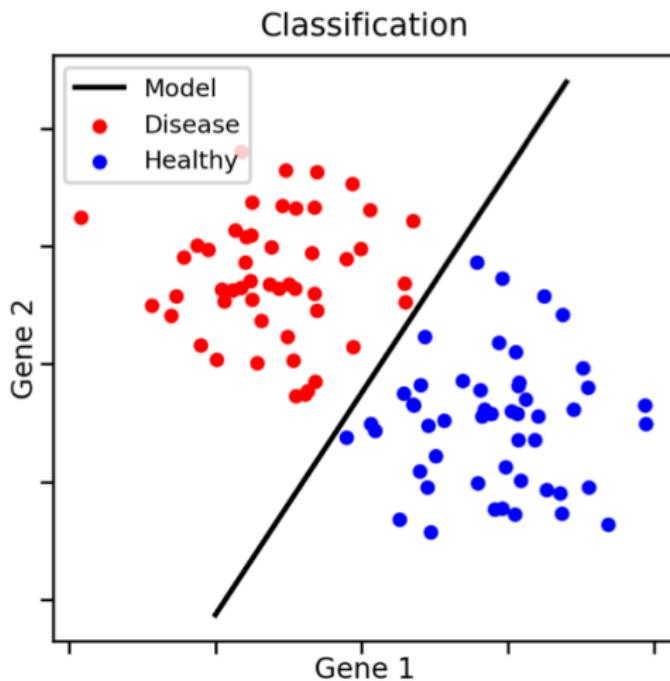
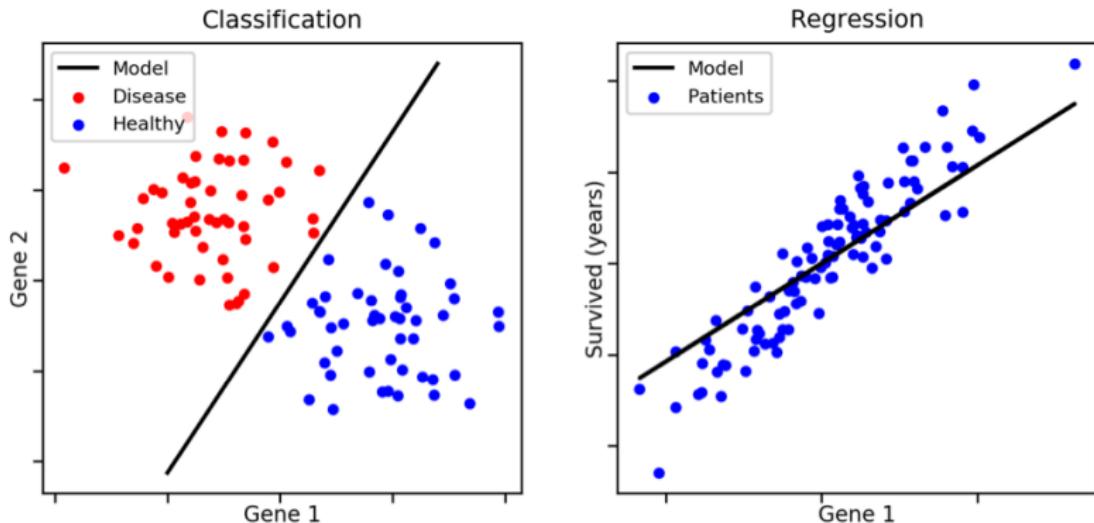


Image source: <https://aldro61.github.io/microbiome-summer-school-2017/sections/basics/>

Which is the Correct Approach?



- ▶ The approach we use depends upon what we want to achieve, and what works best based upon the data we have
- ▶ Much machine learning involves investigating different approaches

Comic of the Day



<http://xkcd.com/1838/>

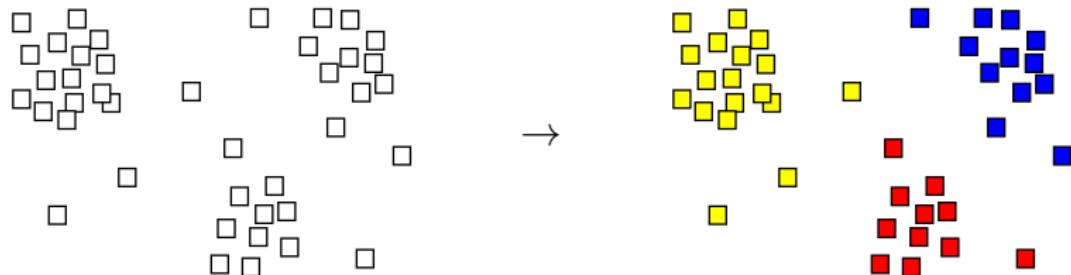
Additional Types of Machine Learning

Types of ML: Unsupervised Learning

Definition

Unsupervised learning involves extracting structure from an *unlabeled* data set $\{\mathbf{x}_i\}_{i=1}^n$. This structure could be in the form of labels (e.g., clustering), new features (e.g., dimensionality reduction), or some model involving \mathbf{x} .

Canonical example is **clustering**, or grouping data points by similarity.



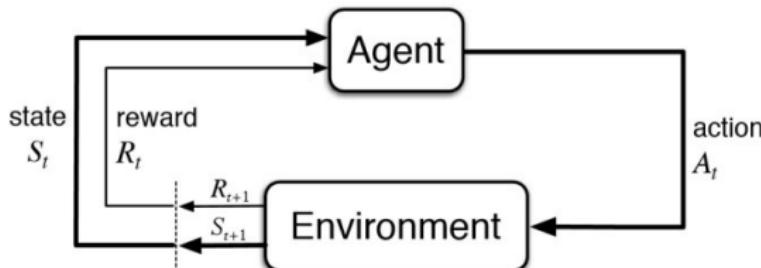
Types of ML: Reinforcement Learning

Definition

Reinforcement learning is learning what to do—how to map situations to actions—so as to maximize a numerical reward signal. The learner is not told which actions to take, but instead must discover which actions yield the most reward by trying them.

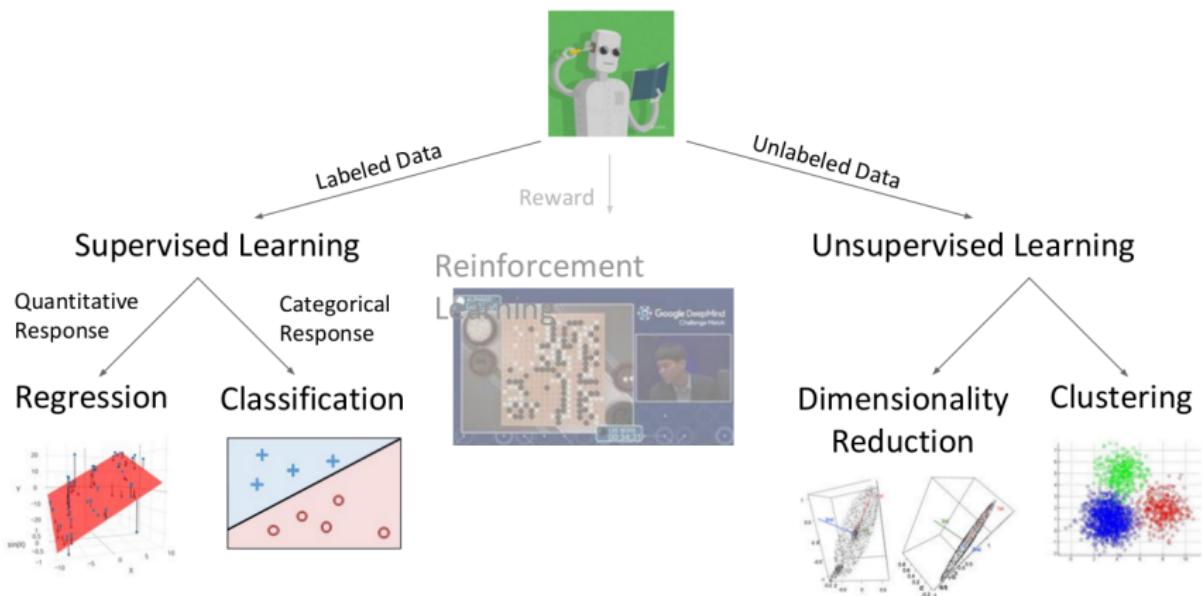
(*Reinforcement Learning* by Sutton and Barto)

Useful for learning how to play games (e.g., Tic-tac-toe, Go)



(Image source: <https://www.kdnuggets.com/2018/03/5-things-reinforcement-learning.html>)

Core Machine Learning Taxonomy



(Image source: DS 100 lecture notes)

Other Types of ML

In addition to supervised, unsupervised, and reinforcement learning, ML includes the following:

- Deep learning: learning involving multi-layer artificial neural networks (can be applied in a variety of contexts, including supervised, unsupervised, and reinforcement learning)
- Semi-supervised learning: some data points have labels, others don't
- Weakly supervised learning: all data points have labels, but labels may be noisy
- Online learning: data becomes available over time, and models need to be updated
- Transfer learning: take knowledge gained in one domain and apply it to help learn in another domain

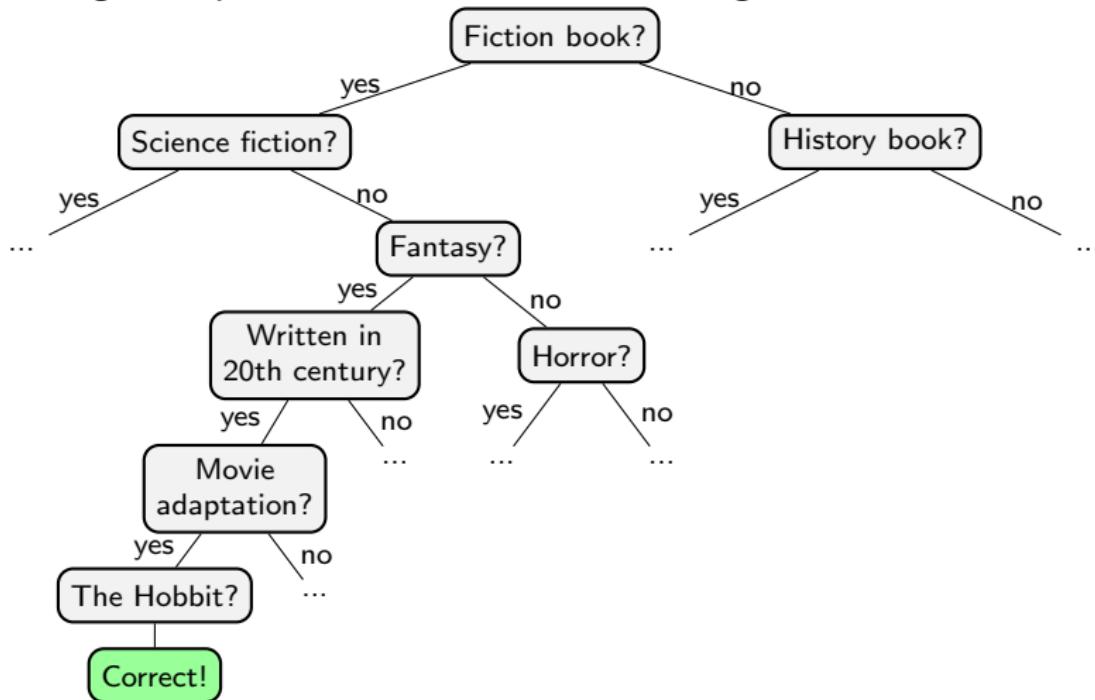
CS 457/557: Machine Learning

Lecture 02-1: Decision Trees

Lecture 02-1a: Decision Trees

Classification by Flow Chart

Motivating example: 20 Questions: I'm thinking of a book...



Decision Trees

- ▶ A decision tree leads us from a set of **attributes** (features of the input) to some output
- ▶ For example, we have a database of customer records for restaurants
- ▶ These customers have made a number of decisions about whether to wait for a table, based on a number of attributes:
 1. *Alternate*: is there an alternative restaurant nearby?
 2. *Bar*: is there a comfortable bar area to wait in?
 3. *Fri/Sat*: is today Friday or Saturday?
 4. *Hungry*: are we hungry?
 5. *Patrons*: number of people in the restaurant (*None, Some, Full*)
 6. *Price*: price range (*\$, \$\$, \$\$\$*)
 7. *Raining*: is it raining outside?
 8. *Reservation*: have we made a reservation?
 9. *Type*: kind of restaurant (*French, Italian, Thai, Burger*)
 10. *WaitEstimate*: estimated wait time in minutes (*0-10, 10-30, 30-60, >60*)
- ▶ The function we want to learn is whether or not a (future) customer will decide to wait, given some particular set of attributes

Decisions Based on Attributes

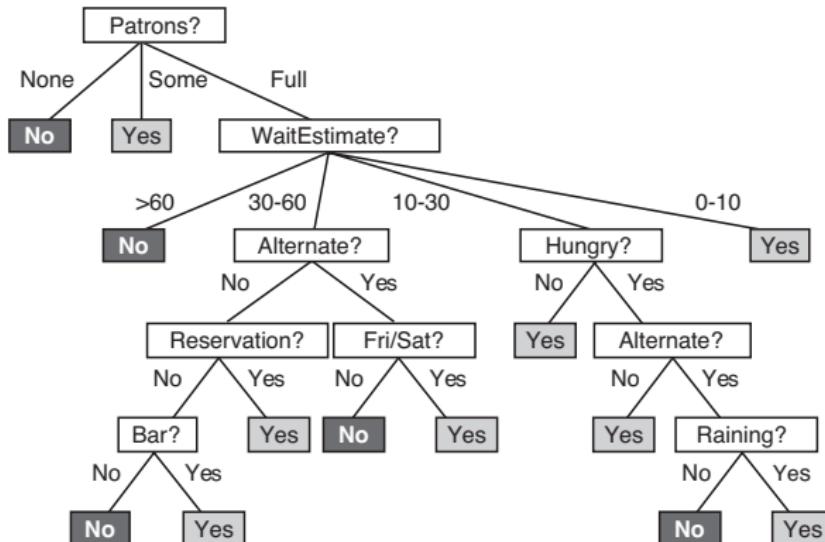
- ▶ **Training set:** cases where patrons have decided to wait or not, along with the associated attributes for each case

Example	Attributes										Target Wait
	Alt	Bar	Fri	Hun	Pat	Price	Rain	Res	Type	Est	
X_1	T	F	F	T	Some	\$\$\$	F	T	French	0–10	T
X_2	T	F	F	T	Full	\$	F	F	Thai	30–60	F
X_3	F	T	F	F	Some	\$	F	F	Burger	0–10	T
X_4	T	F	T	T	Full	\$	F	F	Thai	10–30	T
X_5	T	F	T	F	Full	\$\$\$	F	T	French	>60	F
X_6	F	T	F	T	Some	\$\$	T	T	Italian	0–10	T
X_7	F	T	F	F	None	\$	T	F	Burger	0–10	F
X_8	F	F	F	T	Some	\$\$	T	T	Thai	0–10	T
X_9	F	T	T	F	Full	\$	T	F	Burger	>60	F
X_{10}	T	T	T	T	Full	\$\$\$	F	T	Italian	10–30	F
X_{11}	F	F	F	F	None	\$	F	F	Thai	0–10	F
X_{12}	T	T	T	T	Full	\$	F	F	Burger	30–60	T

- ▶ We now want to learn a tree that agrees with the decisions already made, in hopes that it will allow us to predict future decisions

Decision Tree Functions

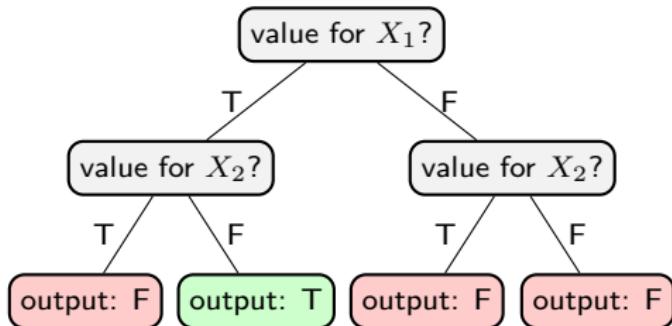
- For the examples given, here is a “true” tree (one that will lead from the inputs to the same outputs)



Expressiveness of Decision Trees

A function of two boolean variables: A decision tree for the function:

X_1	X_2	$Y = X_1 \wedge \neg X_2$
T	T	F
T	F	T
F	T	F
F	F	F



Such trees can express **any deterministic function!**

E.g., with boolean functions:

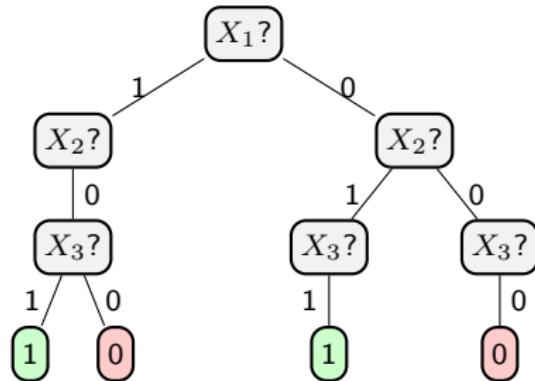
- Each row in the truth table corresponds to a path in the tree
- Leaf nodes encode the output of the function with the path from the root as input

Lecture 02-1b: Building Decision Trees

Building Decision Trees, Take 1

To **build** a tree from a data set $\{(\mathbf{x}_i, y_i)\}_{i=1}^N$, we can create different paths from the root for each input \mathbf{x}_i .

Index	Inputs			Output
	X_1	X_2	X_3	
1	0	0	0	0
2	0	1	1	1
3	1	0	1	1
4	1	0	0	0



To **predict** the output y_{N+1} associated with a new example \mathbf{x}_{N+1} , we follow the path in the tree specified by the values of \mathbf{x}_{N+1} :

- If we reach a leaf, we use the leaf's value as our prediction
- If we reach an internal node E that branches on X_i but has no child node corresponding to attribute value $x_{N+1,i}$, then we predict the most common output (class) from all examples at or below S

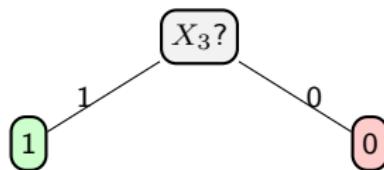
Problems with the Building Process

There are some **problems** with trees built in this way:

- ① What happens with collisions?
- ② They are **overfit** to training data and do not **generalize** well to new examples.
- ③ They are not **compact**.

A more compact tree results from splitting at X_3 only:

Index	Inputs			Output
	X_1	X_2	X_3	
1	0	0	0	0
2	0	1	1	1
3	1	0	1	1
4	1	0	0	0



Searching for Trees?

Instead of **building** trees by “inserting” full paths corresponding to data points, can we **search** for trees that are compact and consistent with the data set?

Unfortunately the answer is no in most cases, as exhaustive search is **too expensive** due to the large number of possible functions (trees) to consider.

- For p binary input attributes, there are 2^p possible input combinations (i.e., rows in the truth table)
- With a binary output, there are 2^{2^p} possible functions
- For $p = 5$, this leads to over 4,294,967,296 trees!

Building Trees Top-Down

Rather than search across the set of **all** trees, we will **build** our trees by:

- ① Choosing an attribute A from the data set
- ② Dividing our data set (examples) according to the values of A
- ③ Placing each subset of examples into a subtree below the node for attribute A

There are many possible implementations of this, but it is perhaps most easily understood recursively!

One important remaining question: **how** do we choose the attribute A that we use to split our examples?

Decision Tree Learning Algorithm

```
function DECISION-TREE-LEARNING(examples, attributes, parent-examples) returns
  tree
    if examples is empty then return PLURALITY-VALUE(parent-examples)
    else if all examples have the same classification then return the classification
    else if attributes is empty then return PLURALITY-VALUE(examples)
    else
       $A \leftarrow \operatorname{argmax}_{a \in \text{attributes}} \text{IMPORTANCE}(a, \text{examples})$ 
      tree  $\leftarrow$  a new decision tree with root test A
      for each value  $v_k$  of A do
         $exs \leftarrow \{e : e \in \text{examples} \text{ and } e.A = v_k\}$ 
        subtree  $\leftarrow$  DECISION-TREE-LEARNING(exs, attributes - A, examples)
        add a branch to tree with label (A =  $v_k$ ) and subtree subtree
    return tree
```

PLURALITY-VALUE(): returns output decision-value for **majority** of examples

IMPORTANCE(): **rates** attributes for their importance in making decisions for the given set of examples (the only actually complex part)

Lecture 02-1c: Choosing Important Attributes

Choosing “Important” Attributes

- ▶ The precise tree we build will depend upon the order in which the algorithm chooses attributes and splits up examples
- ▶ Suppose we have the following training set of 6 examples, defined by the boolean attributes A, B, C, with outputs as shown:

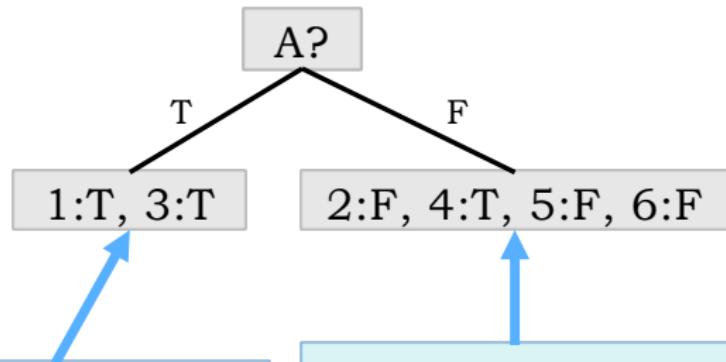
Case	A	B	C	Output
1	T	F	F	T
2	F	T	T	F
3	T	T	F	T
4	F	F	T	T
5	F	F	F	F
6	F	T	F	F

- ▶ We will consider two possible orders for the attributes when we build our tree: {A, B, C} and {C, B, A}

Choosing “Important” Attributes

- Suppose we use the order {A, B, C}: start by dividing up cases based on variable A

Case	A	B	C	Output
1	T	F	F	T
2	F	T	T	F
3	T	T	F	T
4	F	F	T	T
5	F	F	F	F
6	F	T	F	F



Here, all Outputs are the same, so we can replace this with a simple leaf node with that value.

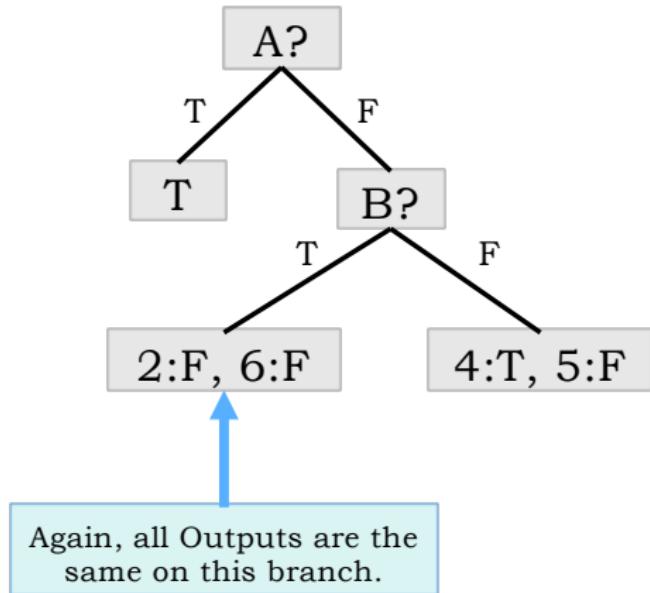
Each of these is a case for which attribute A has the right value, along with the appropriate Output value for that case.

This is an example of the **second** base case stopping condition of the recursive algorithm.

Choosing “Important” Attributes

- ▶ Order {A, B, C}: next, divide un-decided cases based on variable B

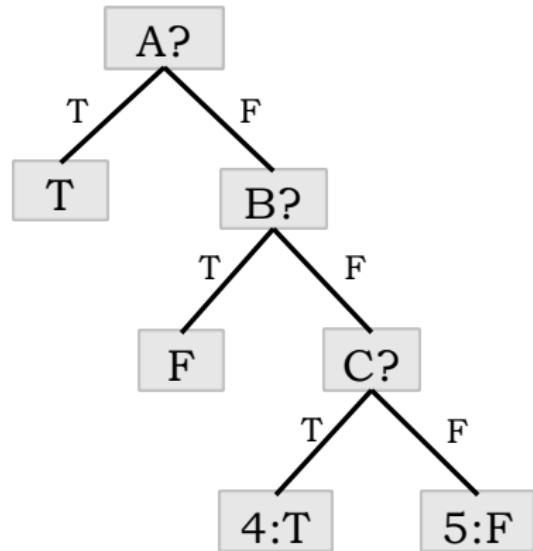
Case	A	B	C	Output
1	T	F	F	T
2	F	T	T	F
3	T	T	F	T
4	F	F	T	T
5	F	F	F	F
6	F	T	F	F



Choosing “Important” Attributes

- ▶ Order {A, B, C}: last, divide un-decided cases based on variable C

Case	A	B	C	Output
1	T	F	F	T
2	F	T	T	F
3	T	T	F	T
4	F	F	T	T
5	F	F	F	F
6	F	T	F	F

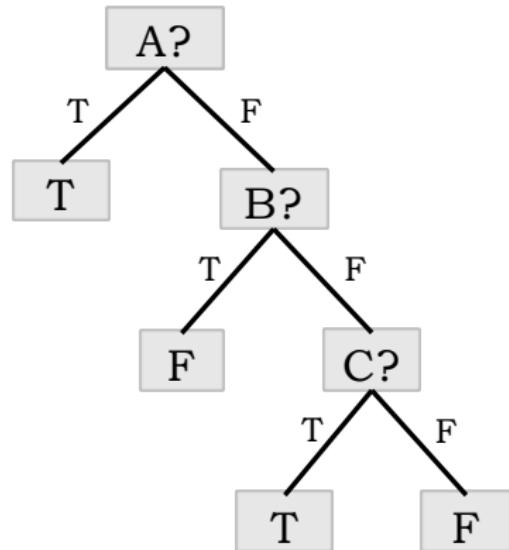


Now, we can replace the last nodes with the relevant decision Output.

Choosing “Important” Attributes

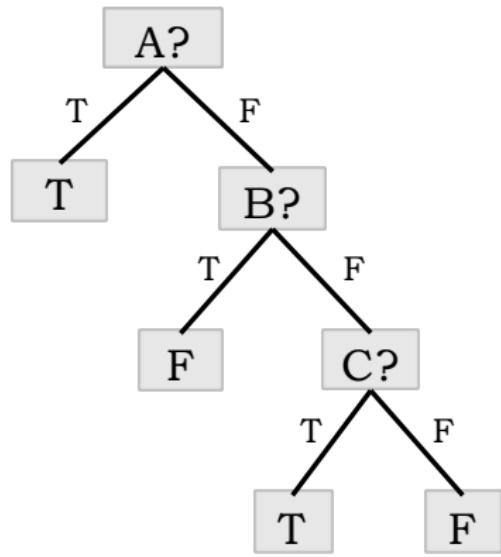
- ▶ Order {A, B, C}: the final decision tree for our data-set

Case	A	B	C	Output
1	T	F	F	T
2	F	T	T	F
3	T	T	F	T
4	F	F	T	T
5	F	F	F	F
6	F	T	F	F

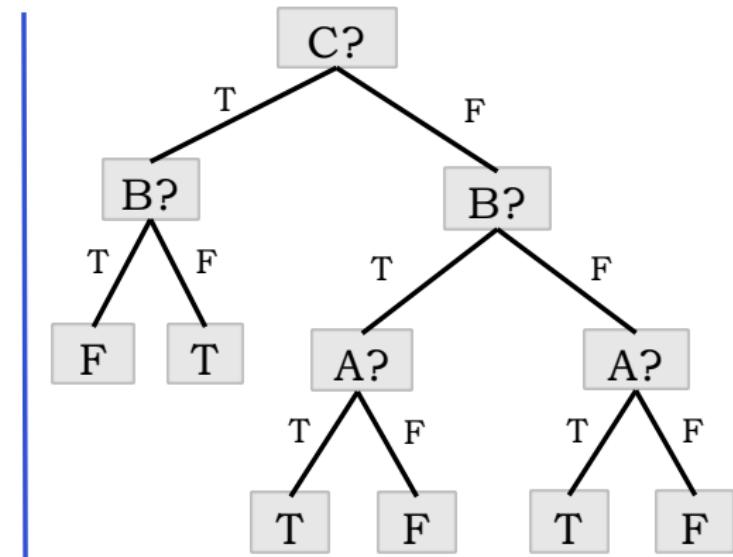


Choosing “Important” Attributes

- If we reverse the order of attributes and do the same process, we get a different, somewhat larger tree (although both will give same decision results on our set)



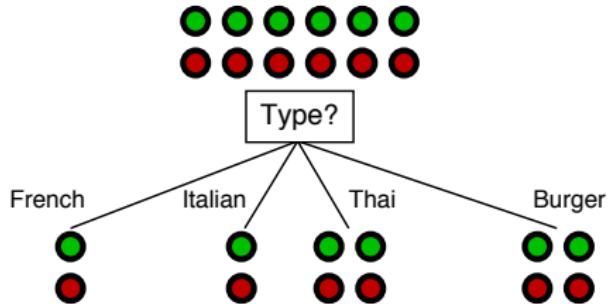
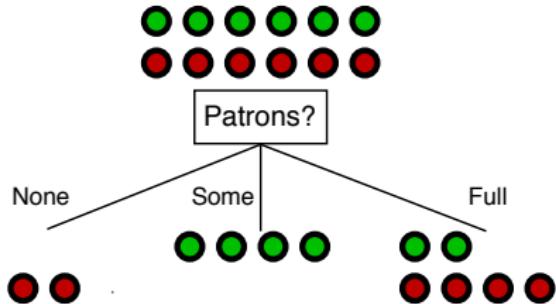
{A, B, C}



{C, B, A}

Choosing Attributes

● = waits
● = doesn't wait



- ▶ Intuitively, a good choice of the attribute to use is one that gives us the **most information** about how output decisions are made
 - ▶ Ideally, it would divide our outputs perfectly, telling us everything we needed to know to make our decision
 - ▶ Often, a single attribute only tells us part of what we need to know, so we prefer those that tell us the most
 - ▶ In the example, **Patrons** gives us more information than **Type**, since some values of the first attribute predict decision **perfectly**, while no values of second do the same

Lecture 02-1d: Entropy and Information Gain

Entropy for Decision Trees

To formalize attribute importance, we need additional notation:

- Any node E in the decision tree contains a subset of (indices of) the data set $\{(\mathbf{x}_i, y_i)\}_{i=1}^N$
 - Root node E_0 has all examples: $E_0 = \{1, 2, \dots, N\}$
- Let $p(E)$ and $n(E)$ be the number of positive and negative examples, respectively:

$$p(E) = |\{i \in E \mid y_i = 1\}| \quad \text{and} \quad n(E) = |\{i \in E \mid y_i = 0\}|.$$

(when the node E is clear from context, we simplify to p and n)

For any node E , the **entropy** of E is defined as

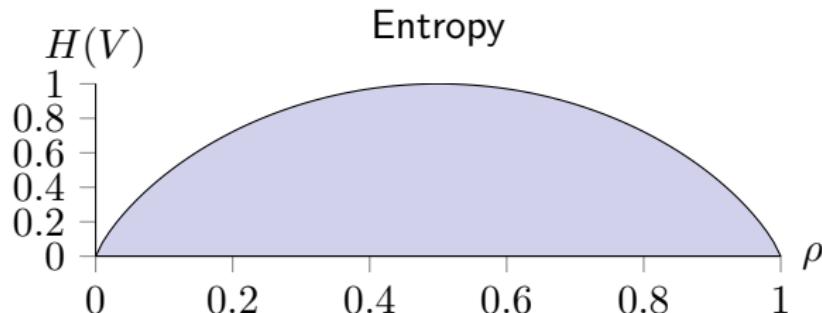
$$H(E) = - \left(\frac{p}{p+n} \right) \log_2 \left(\frac{p}{p+n} \right) - \left(\frac{n}{p+n} \right) \log_2 \left(\frac{n}{p+n} \right).$$

(based on information theory)

Intuition for Entropy

Some intuition: For a Bernoulli random variable V representing the outcome of a coin toss with probability ρ of heads, the entropy of V is

$$\begin{aligned} H(V) &= - \sum_{i \in \text{range}(V)} P(V = i) \log_2 P(V = i) \\ &= -[P(V = 1) \log_2 P(V = 1) + P(V = 0) \log_2 P(V = 0)] \\ &= -[\rho \log_2 \rho + (1 - \rho) \log_2(1 - \rho)] \end{aligned}$$



(Pure nodes have $H(E) = 0$; others have $H(E) > 0$)

Information Gain

Recall: At a node E , if E is not split further, then we use the **most common class** as the predicted value for examples in E .

- Entropy measures the uncertainty about the examples at a node
- Smaller entropy implies better predictions!

Suppose we **split** node E based on some attribute A whose set of possible values is V :

- This creates child nodes $E_v = \{i \in E \mid x_{i,A} = v\}$ for each $v \in V$
- Each child node E_v has entropy $H(E_v)$
- The **(weighted) remaining entropy** after splitting on A is then

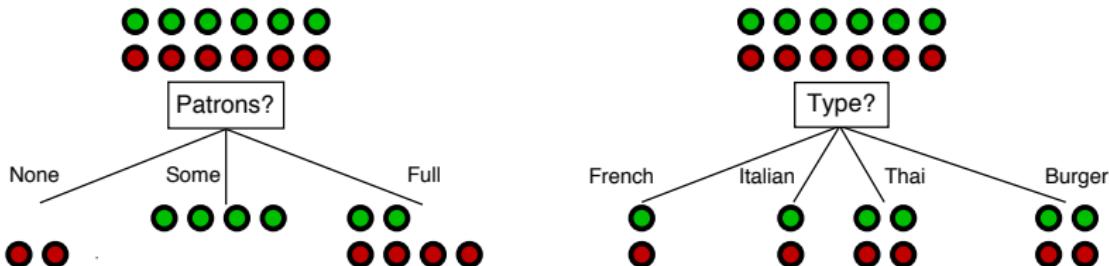
$$\text{Remainder}(A) = \sum_{v \in V} \frac{|E_v|}{|E|} H(E_v).$$

- The **information gain** (entropy reduction) after splitting on A is:

$$\text{Gain}(A) = H(E) - \text{Remainder}(A).$$

Choosing Variables Using the Information Gain

● = waits
● = doesn't wait



- Now we can be precise about how *Patrons* gives us more information than *Type*:

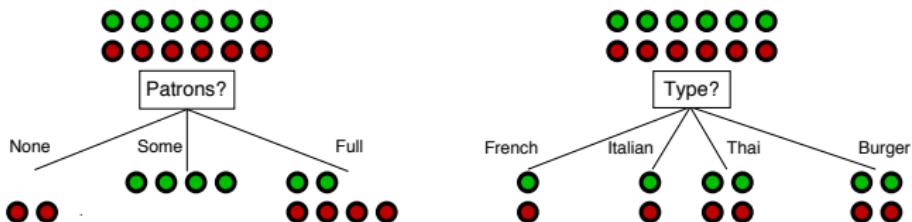
$$H(\text{Examples}) = -\left(\frac{6}{12} \log_2 \frac{6}{12} + \frac{6}{12} \log_2 \frac{6}{12}\right)$$

$$= -\left(\frac{1}{2} \log_2 \frac{1}{2} + \frac{1}{2} \log_2 \frac{1}{2}\right)$$

$$= -\left(-\frac{1}{2} + -\frac{1}{2}\right) = 1.0$$

Choosing Variables Using the Information Gain

● = waits
● = doesn't wait



- Now we can be precise about how *Patrons* gives us more information than *Type*:

$$\begin{aligned} \text{Gain}(\text{Patrons}) &= H(\text{Examples}) - \text{Remainder}(\text{Patrons}) \\ &= 1.0 - \left(\frac{2}{12} H(E_1) + \frac{4}{12} H(E_2) + \frac{6}{12} H(E_3) \right) \end{aligned}$$

Thus, since we have:

$$H(E_1) = -\left(\frac{0}{2} \log_2 \frac{0}{2} + \frac{2}{2} \log_2 \frac{2}{2}\right) = 0$$

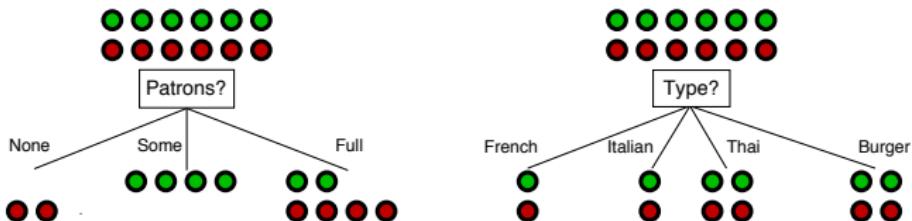
$$H(E_2) = -\left(\frac{4}{4} \log_2 \frac{4}{4} + \frac{0}{4} \log_2 \frac{0}{4}\right) = 0$$

$$H(E_3) = -\left(\frac{2}{6} \log_2 \frac{2}{6} + \frac{4}{6} \log_2 \frac{4}{6}\right) \approx 0.918$$

$$\text{Gain}(\text{Patrons}) = 1.0 - \frac{0.918}{2} = 0.541$$

Choosing Variables Using the Information Gain

● = waits
● = doesn't wait



- Now we can be precise about how *Patrons* gives us more information than *Type*:

$$Gain(Type) = H(Examples) - Remainder(Type)$$

$$= 1.0 - \left(\frac{2}{12} H(E_1) + \frac{2}{12} H(E_2) + \frac{4}{12} H(E_3) + \frac{4}{12} H(E_4) \right)$$

Thus, since we have:

$$H(E_1) = H(E_2) = H(E_3) = H(E_4) = 1.0$$

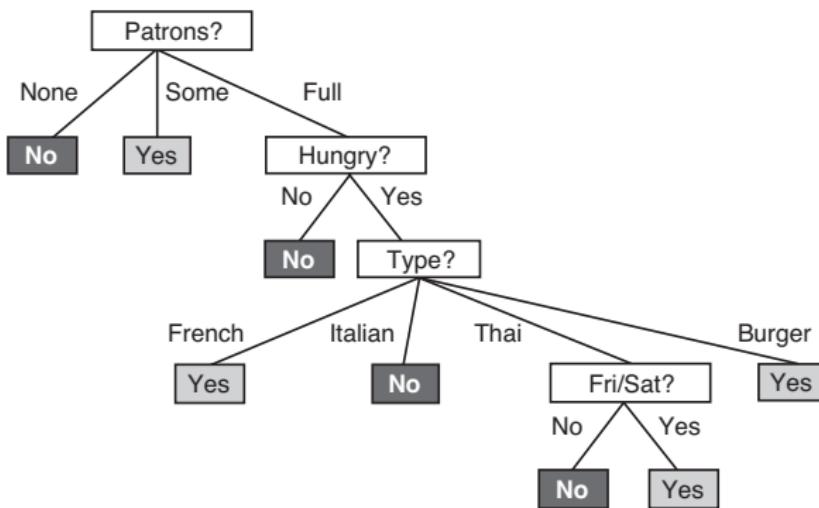
$$Gain(Patrons) = 1.0 - 1.0 = 0$$

And so we would choose to split on *Patrons*, since:

$$Gain(Patrons) = 0.541 > Gain(Type) = 0$$

Learning with Information Gain

- If we use this information gain concept of information to rate the IMPORTANCE of an attribute, and always split based on the one that gives us the greatest gain, we can learn the following, more compact tree for the restaurant example:



CS 457/557: Machine Learning

Lecture 02-2: Decision Trees

Lecture 02-2a: Decision Tree Heuristics

Quick Recap: Decision Tree Construction

Greedy constructive heuristics recursively partition the examples by splitting on one attribute at a time in a top-down fashion.

- Each node E in the tree contains a subset of (indices of) examples
- Root node E_0 initialized with all examples $\{(\mathbf{x}_i, y_i)\}_{i=1}^N$

Building a Decision Tree Recursively

```
procedure RECURSIVESPLIT(Node  $E$ )
    return if  $E$  cannot or should not be split further            $\triangleright E$  is a leaf node
    Let  $A$  be the attribute that maximizes information gain through splitting
    for each possible value  $v$  of attribute  $A$  do
         $E_v \leftarrow \{i \in E \mid x_{i,A} = v\}$ 
        RECURSIVESPLIT( $E_v$ )
```

Information Gain and Other Heuristics

Some remaining issues with attribute selection:

- ① What should be done in the case of ties in information gain?
 - Deterministic or random selection?
 - Use another measure as a tie-breaker (e.g., select the attribute that produces the largest number of pure child nodes)
- ② Do other measures for comparing attributes do better?

There is **no single correct answer** to either of these questions.

Definition

A **heuristic** is a method or approach to a problem that tends to work well in practice, but is not provably optimal.

Information gain is a **heuristic**! In most cases it works well, but there are situations in which other measures would produce better trees. (Still need some notion of “better” for trees, too though!)

Different Heuristics, Different Trees

Index	X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8	X_9	X_{10}	X_{11}	Y
0	0	1	0	0	0	0	0	0	0	0	1	0
1	0	0	1	0	0	0	0	0	0	0	1	1
2	1	0	0	0	0	0	0	0	0	0	1	0
3	0	0	0	0	1	0	0	0	0	0	0	1
4	0	0	0	0	0	0	0	0	0	1	0	0
5	0	0	0	1	0	0	0	0	0	0	0	1
6	0	0	0	0	0	0	0	0	1	0	1	0
7	0	0	0	0	0	1	0	0	0	0	0	1
8	0	0	0	0	0	0	0	1	0	0	1	0
9	0	0	0	0	0	0	1	0	0	0	0	1

{0,2,4,6,8,1,3,5,7,9}

$X_1?$

1

{2}

0

{0,4,6,8,1,3,5,7,9}
(Next split)

{0,2,4,6,8,1,3,5,7,9}

$X_{11}?$

1

{0,2,6,8,1}
(Next split)

0

{4,3,5,7,9}
(Next split)

Using “Decisiveness” as a Heuristic

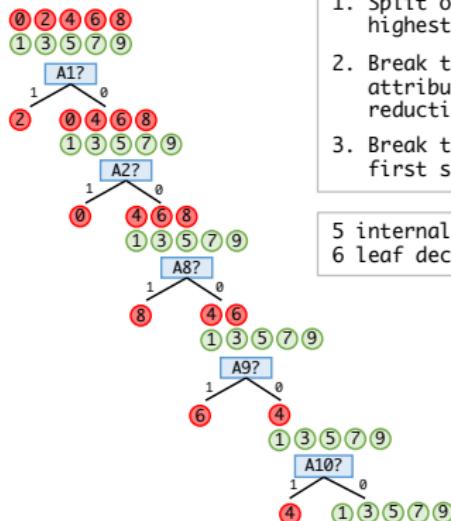
- ▶ Suppose, rather than information gain, we always chose a feature that was most decisive (having the largest number of inputs for which it gives a firm output decision)
- ▶ On this data-set, and breaking ties by always taking the first such feature in the feature-set, we get a tree as shown



1. Split on attribute that gives us highest number of decided inputs.
2. Break ties after step 1: split on first such attribute encountered.

8 internal attribute nodes,
9 leaf decision nodes (17 total).

Combining “Decisiveness” and Information Gain

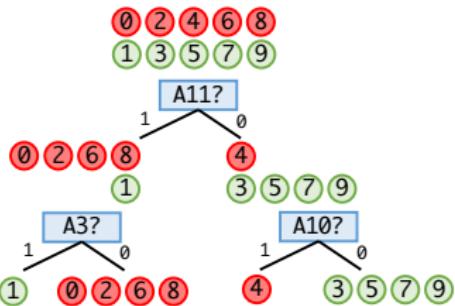


1. Split on attribute that gives us highest number of decided inputs.
2. Break ties after step 1: split on attribute that gives us biggest reduction in overall entropy.
3. Break ties after step 2: split on first such attribute encountered.

5 internal attribute nodes,
6 leaf decision nodes (11 total).

- ▶ We can improve things by combining heuristics
- ▶ If we choose always the attribute that gives us the most decisive split, but break ties using information gain as a **secondary** heuristic, we get a more compact tree (6 total nodes less)

Combining “Decisiveness” and Information Gain



1. Split on attribute that gives us biggest reduction in overall entropy.
2. Break ties after step 2: split on first such attribute encountered.

3 internal attribute nodes,
2 leaf decision nodes (5 total).

- ▶ The best result, however, is to simply use information gain (12 less nodes than the original, largest tree, 6 less than the one using combined heuristics)
 - ▶ Ties don't actually matter here, so no other heuristic needed
- ▶ Is this guaranteed **always** to happen?
 - ▶ No, but the heuristic is considered useful for the reason that there will be more cases in which it works well than ones where it won't..

Another Heuristic: Gini Impurity

Another measure of quality for any node E is the **Gini impurity** or **Gini index**.

$$\begin{aligned}G(E) &= \left(\frac{p}{p+n}\right)\left(1 - \frac{p}{p+n}\right) + \left(\frac{n}{p+n}\right)\left(1 - \frac{n}{p+n}\right) \\&= \frac{2pn}{(p+n)^2}.\end{aligned}$$

(p and n are the number of positive and negative examples in node E)

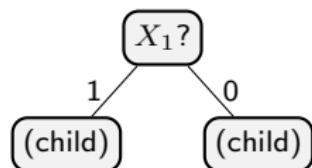
- Interpreted as a measure of total variance across the classes
- Pure nodes have $G(E) = 0$; others have $G(E) > 0$
- Similar to entropy, but computationally simpler (no logarithms)

Lecture 02-2b: Decision Trees with Continuous-Valued Attributes

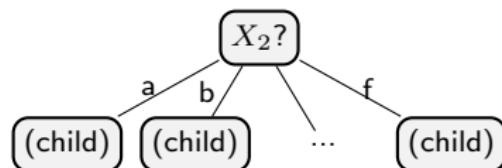
Data for Decision Trees

So far, we have looked at decision trees with binary or discrete inputs:

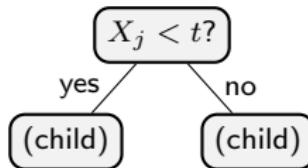
Splitting on binary feature X_1 with values $\{0, 1\}$:



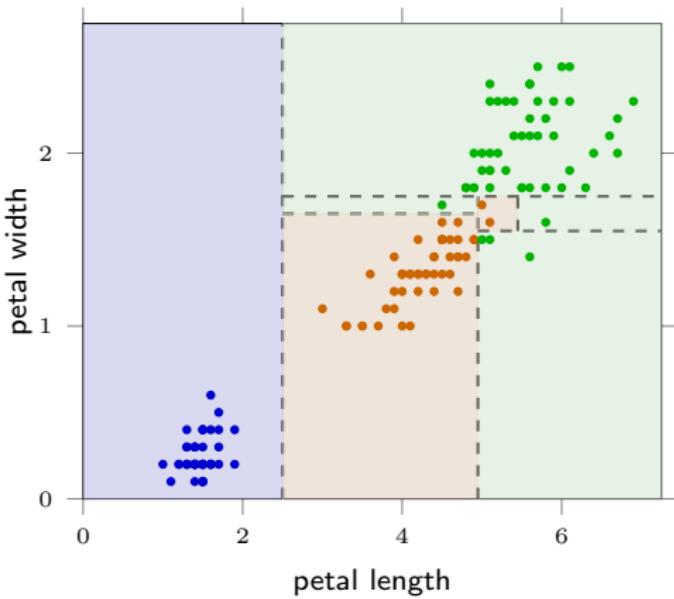
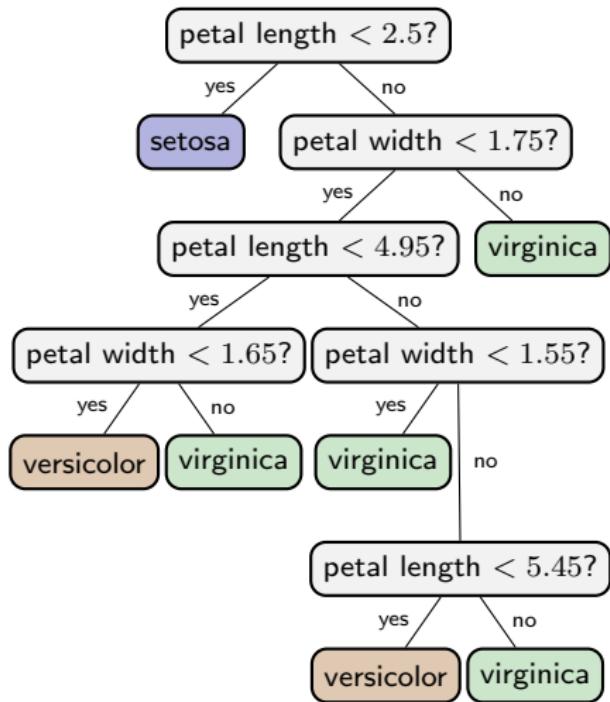
Splitting on discrete feature X_2 with values $\{a, b, c, d, e, f\}$:



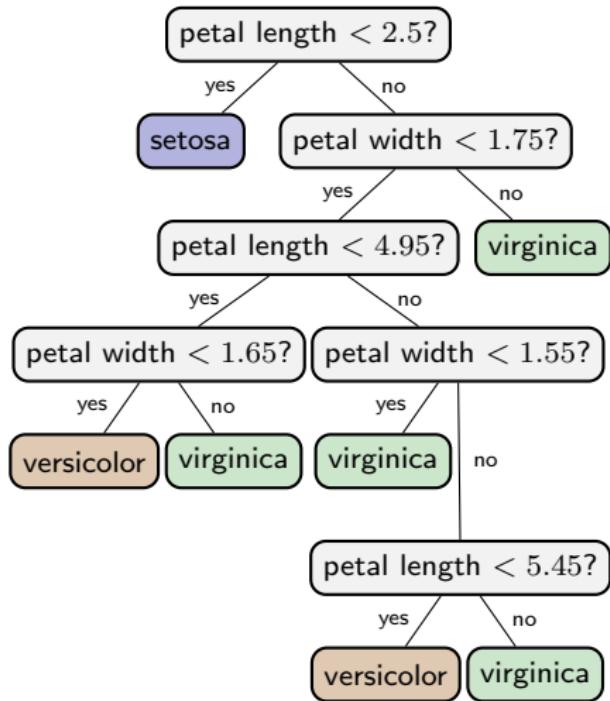
For a **continuous-valued** input attribute X_j , we typically use a binary split of the form " $X_j < t?$ " for some $t \in \mathbb{R}$:



Example: Iris Data



Decision Tree Observations



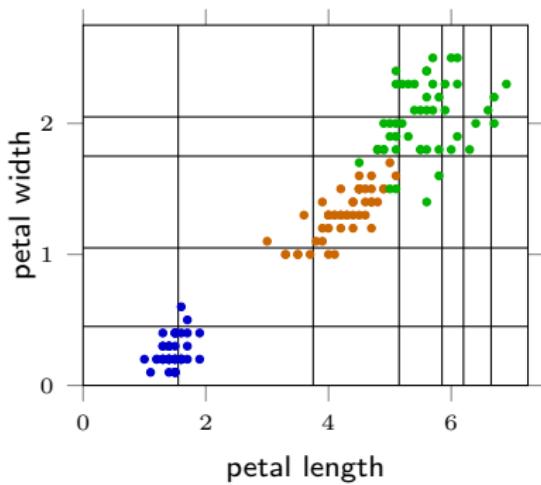
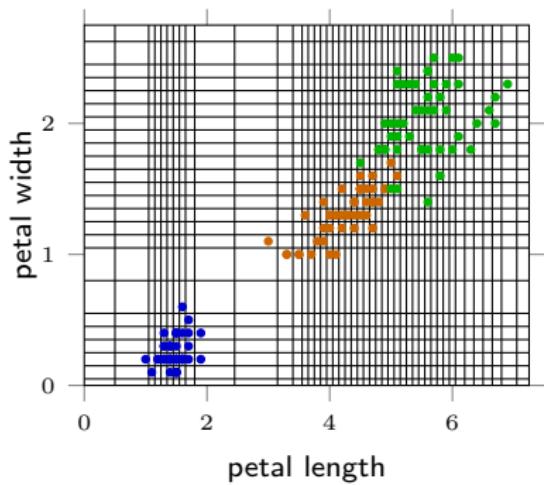
- Some splits result in **pure nodes** corresponding to regions with points in a single class
- Other splits lead to **impure nodes** which are split further
- Picking splits halfway between points makes sense for generalizing
- There are **many different options** for splitting
- Full tree has **perfect accuracy** on the training set (assuming no identical points in opposite classes, which result in **unsplittable nodes**)
⇒ Full tree almost certainly **overfits** (low bias, high variance)

Ways to Select a Split Threshold

With some criterion for measuring the quality of a split (e.g., entropy or Gini impurity), we can choose a split threshold using either:

Best-of-all: Examine all possible attributes and all possible thresholds for a split and take the best one

Best-of-subset: Examine a random subset of attributes and thresholds and choose the best split from there



Regression Trees

What happens if our **output** attribute Y is **not binary**?

- If Y is discrete, then we have **multi-class classification**:
 - Prediction at a node still uses majority vote
 - Entropy measure updated to handle multiple classes
- If Y is continuous, then we have a **regression** problem!

We can adapt decision tree algorithm to create **regression trees**:

- Same types of feature splits are used
- At any node E , the predicted value $\hat{y}(E)$ is the average output value at that node:
$$\hat{y}(E) = \frac{1}{|E|} \sum_{i \in E} y_i.$$
- Node quality is measured as a residual sum of squares:
$$\sum_{i \in E} (y_i - \hat{y}(E))^2$$
 (variance of output values at the node)
- Splits chosen to minimize residual sums of squares across child nodes

Lecture 02-2c: Additional Notes on Decision Trees

Stopping Criteria

When building a decision tree, when should we **stop** splitting a node E ?

Some obvious stopping criteria:

- E is **pure** (contains only points from one class)
- E is **unsplittable**
(e.g., E contains $\mathbf{x}_i = (1, 1, 1), y_i = 0$ and $\mathbf{x}_{i'} = (1, 1, 1), y_{i'} = 1$)

Trees produced with these criteria tend to **overfit**:

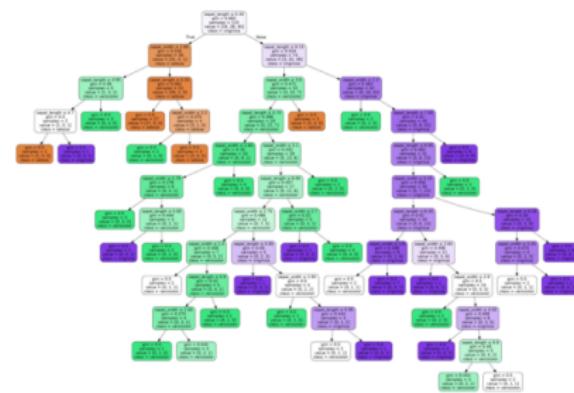
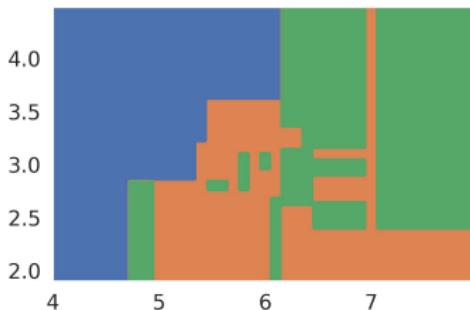


Image source: DS 100 lecture notes

Goals for Constructing Decision Trees

When building a decision tree, we ultimately want:

- ➊ **Good accuracy** on the training set, $\{(\mathbf{x}_i, y_i)\}_{i=1}^N$
- ➋ **Good accuracy** on out-of-sample data

Unfortunately, these two criteria often compete with each other:

- Deep trees can differentiate between most inputs
- Longer paths to reach a decision are less likely to generalize

Instead, we try to **balance** these two goals with heuristics that prevent our trees from getting **too big**.

Dealing with Overfitting

Two general approaches to dealing with **overfitting** during construction:

- ① Add **hard constraints** to prevent full growth; e.g.:

- E is at depth d in tree
- There are too many leaves in the tree already
(every split replaces an existing leaf with two new ones)
- E contains too few data points to split
- Every split at E produces one or more child nodes with an insufficient number of data points
- Information gain by splitting is small
(potentially misses good splits one step ahead)

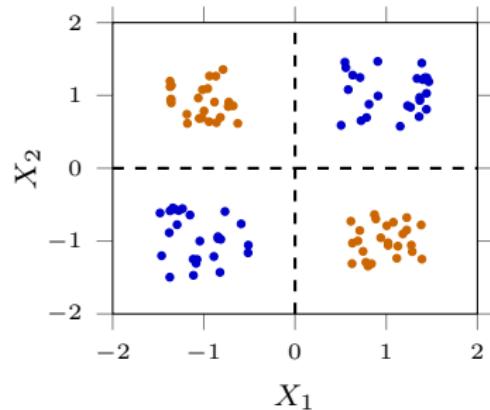
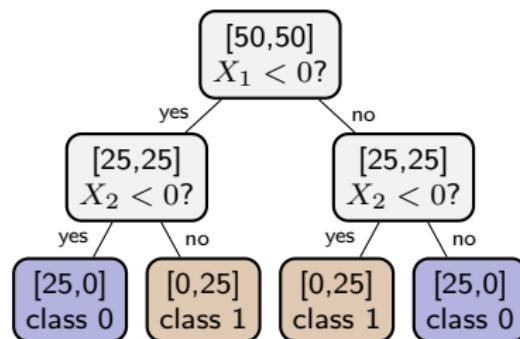
- ② Build the full tree, then **prune back** low-value nodes
(e.g., cost complexity pruning, χ^2 pruning)

Skiena: “Decision tree construction is hacking, not science.”

Another Problem with the Greedy Constructive Approach

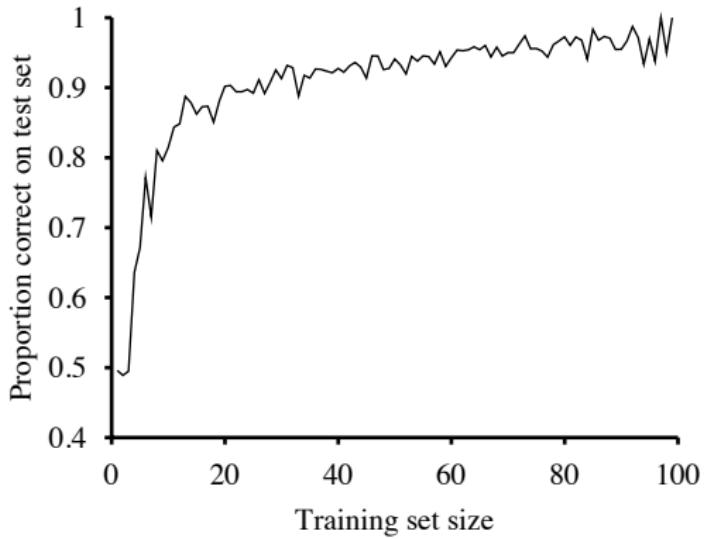
The decision tree construction algorithm is **greedy**: when selecting a split, we only consider the **immediate impact** of the split, not what the split **might allow us to do** further down in the tree.

Example:



- Only 3 splits needed, but first split doesn't yield any information gain.
- Better splits can be found using a look-ahead process (**more work**).

Performance of Learning



- If we start with a set of 100 random examples of the restaurant problem, we can see that the accuracy of the learning increases relative to the size of the training set

Pros and Cons of Decision Trees

Pros

- Easy to interpret
- Non-linear decision boundaries
- Easily supports both categorical and numerical variables
- Insensitive to feature scaling

Cons

- Lack of elegance (math)
- Sensitive to noise in training data (tendency to overfit)
- Decision boundary is axis-aligned
- Finding globally optimal tree is NP-hard
- Does not extrapolate well
- Tend to have lower accuracy compared to other models

CS 457/557: Machine Learning

Lecture 03-1: Simple Linear Regression

Lecture 03-1a: Intro to Linear Regression

Quick Recap: The Supervised Learning Problem

Supervised Learning Problem

Given a *labeled* data set $\{(\mathbf{x}_i, y_i)\}_{i=1}^N$ with $y = f(\mathbf{x})$ for some unknown function f , identify a **hypothesis function** h that approximates f well (i.e., $h(\mathbf{x}) \approx f(\mathbf{x})$ for all \mathbf{x}).

The **supervised learning process**:

- ① Start with a labeled data set $\{(\mathbf{x}_i, y_i)\}_{i=1}^N$ (set of input-output pairs)
- ② Choose a **hypothesis space (model class)** \mathcal{H} representing the set of all possible hypotheses h to consider
- ③ Use a **learning algorithm** to find a **hypothesis (model)** $h^* \in \mathcal{H}$ such that $h^*(\mathbf{x}_i) \approx y_i$ for all $i \in \{1, 2, \dots, N\}$
- ④ (Optional) Use h^* to **predict** outputs for other inputs!

Motivating Simple Linear Regression

Given: $\{(x_i, y_i)\}_{i=1}^N$, where $x_i, y_i \in \mathbb{R}$.

Assume: $y = f(x) + \epsilon$, where ϵ is a noise term.

Find: a function $h : \mathbb{R} \rightarrow \mathbb{R}$ such that $h(x_i) \approx y_i$ for all i .

(Note: Ideally we want $h(x_i) \approx f(x_i)$, but noise complicates things!)

First Question

What is our **hypothesis space**?

First attempt: All possible functions.

Except...how many functions are there that map from \mathbb{R} to \mathbb{R} ? **A lot!**

It often helps to **restrict** our **hypothesis space** in some way.

- Limits the hypothesis functions that we can consider
- Makes it easier to search

Specifying a Hypothesis Space

One way to specify a hypothesis space (**model class**) is as a set of **parameterized functions**.

- Hypothesis space of all linear functions:

$$\mathcal{H}^{(1)} = \{h : \mathbb{R} \rightarrow \mathbb{R} \mid h(x) = w_0 + w_1x\}$$

- Hypothesis space of all linear and quadratic functions:

$$\mathcal{H}^{(2)} = \{h : \mathbb{R} \rightarrow \mathbb{R} \mid h(x) = w_0 + w_1x + w_2x^2\}$$

- Hypothesis space of all polynomial functions of degree at most d :

$$\mathcal{H}^{(3)} = \left\{ h : \mathbb{R} \rightarrow \mathbb{R} \mid h(x) = \sum_{j=0}^d w_j x^j \right\}$$

In each case, the **weights** w_0, w_1, \dots are the **parameters** that fully specify a particular function h .

Linear Functions

For now, we'll stick with the hypothesis space of linear functions:

$$\mathcal{H}^{(1)} = \{h : \mathbb{R} \rightarrow \mathbb{R} \mid h(x) = w_0 + w_1 x\}.$$

Why? Because linear relationships are **easy to understand** and make a good choice for a **default** model.

Occam's Razor

The simplest explanation is the best explanation.

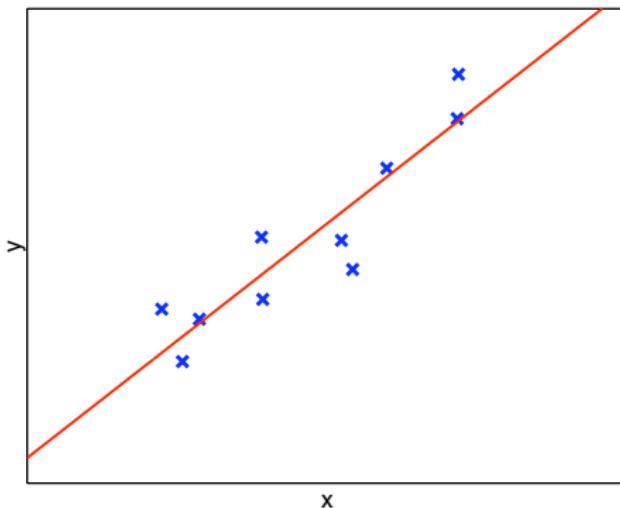
Some additional notation:

- $\mathbf{w} = (w_0, w_1)$ is the **weight vector**
- We typically write a hypothesis function as $h_{\mathbf{w}}$ to indicate its dependence on the weights \mathbf{w}

Examples of particular hypothesis functions:

- $\mathbf{w}^{(1)} = (1, 1) \quad h_{\mathbf{w}^{(1)}}(x) = 1 + x$
- $\mathbf{w}^{(2)} = (7, -3) \quad h_{\mathbf{w}^{(2)}}(x) = 7 - 3x$

An Example



- ▶ For the data given, the best fit for a simple linear function of x is as follows:

$$h(x) \leftarrow y = 1.05 + 1.60x$$

Lecture 03-1b: Building a Learning Algorithm

Learning Algorithm: Loss Functions

Second Question

What is our **learning algorithm**?

I.e., how do we find the **best** hypothesis function from \mathcal{H} ?

To do this, we need some way to **measure** what makes one hypothesis function **better** than another. Look at accuracy of predictions!

- For any $h_{\mathbf{w}}$, predicted output for input x_i is $h_{\mathbf{w}}(x_i) = w_0 + w_1 x_i$.

Definition

A **loss function** (penalty function) measures the difference between a particular input's predicted value and its actual output.

Many loss functions are possible; two common ones for regression are:

- Absolute loss: $\ell_1(y_i, h_{\mathbf{w}}(x_i)) = |y_i - h_{\mathbf{w}}(x_i)|$
- Squared loss: $\ell_2(y_i, h_{\mathbf{w}}(x_i)) = (y_i - h_{\mathbf{w}}(x_i))^2$

Learning Algorithm: Cost Functions

Once we have a loss function for scoring our *individual* predictions, we specify a **cost function (objective function)** that combines losses across all training examples:

$$\begin{aligned} Loss(h_{\mathbf{w}}) &= \sum_{i=1}^N \ell_2(y_i, h_{\mathbf{w}}(x_i)) \\ &= \sum_{i=1}^N (y_i - h_{\mathbf{w}}(x_i))^2 \\ &= \sum_{i=1}^N (y_i - w_0 - w_1 x_i)^2 \end{aligned}$$

This is the **total loss** cost function. Another common cost function is the **average loss**, which just differs by a scaling.

Finding the Best Hypothesis Function

Find $h_{\mathbf{w}} \in \mathcal{H}$ that minimizes $\text{Loss}(h_{\mathbf{w}})$. Equivalently, find

$$\mathbf{w}^* = \arg \min_{\mathbf{w}} \text{Loss}(h_{\mathbf{w}}).$$

This is **really** a problem about finding the **best weights**!

$$\begin{aligned} & \min \sum_{i=1}^N (y_i - w_0 - w_1 x_i)^2 \\ \text{s.t. } & w_0, w_1 \in \mathbb{R} \end{aligned}$$

This is an unconstrained problem with a quadratic objective function given by:

$$J(w_0, w_1) = \sum_{i=1}^N (y_i - w_0 - w_1 x_i)^2.$$

Finding the Best Weights

How do we find the pair of weights (w_0, w_1) that **minimizes** $J(w_0, w_1)$?

Definition

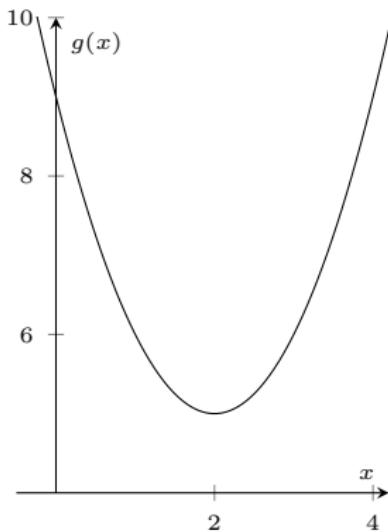
Optimization is the selection of a best element from a set of options.

(Optimization is a subfield of operations research)

First, let's look at minimizing a simpler function involving a single variable:

$g : \mathbb{R} \rightarrow \mathbb{R}$ defined as $g(x) = x^2 - 4x + 9$
for all $x \in \mathbb{R}$.

By inspection, it looks like $x = 2$ minimizes g . How can we formalize this?



Optimization 101: Finding Local Minima and Maxima

Optimization Principles from Calculus

- Fermat's **interior extremum theorem**: For a differentiable function g defined on an open set, every local extremum is a stationary point.
- Stationary points are points where the derivative of g equals 0 (i.e., the tangent line to the function is horizontal).

So to find minima/maxima of a differentiable function, take the derivative of the function, set it equal to zero, and solve for x .

Example: Find minima/maxima of $g(x) = x^2 - 4x + 9$.

$$g(x) = x^2 - 4x + 9$$

$$g'(x) = 2x - 4$$

$$g'(x) = 0 \Leftrightarrow 2x - 4 = 0 \Leftrightarrow 2x = 4 \Leftrightarrow x = 2$$

So $x = 2$ is the only stationary point of g , and is also a local minimum with $g(2) = 3$.

Optimization 102: Finding Local Minima and Maxima

In higher dimensions, we look at the **partial derivatives** of the function to identify stationary points.

Definition

For a two-variable function $g(x, y)$, the **partial derivative** of g with respect to x , denoted $\frac{\partial g}{\partial x}$ or $\frac{\partial g}{\partial x}(x, y)$, is the derivative of g with the variable y treated as constant. The partial derivative with respect to y is defined similarly.

Some examples:

$$\text{If } g(x, y) = ax + by \quad \text{then} \quad \frac{\partial g}{\partial x}(x, y) = a, \quad \frac{\partial g}{\partial y}(x, y) = b$$

$$\text{If } g(x, y) = axy \quad \text{then} \quad \frac{\partial g}{\partial x}(x, y) = ay, \quad \frac{\partial g}{\partial y}(x, y) = ax$$

$$\text{If } g(x, y) = ax^b y^c \quad \text{then} \quad \frac{\partial g}{\partial x}(x, y) = abx^{b-1} y^c, \quad \frac{\partial g}{\partial y}(x, y) = acx^b y^{c-1}$$

Lecture 03-1c: Deriving an Analytical Solution

Derivation of an Analytical Solution

Computing the partial derivative with respect to w_0 yields:

$$\begin{aligned}\frac{\partial}{\partial w_0} J(w_0, w_1) &= \frac{\partial}{\partial w_0} \sum_{i=1}^N (y_i - w_0 - w_1 x_i)^2 && [\text{Definition of } J] \\&= \sum_{i=1}^N \frac{\partial}{\partial w_0} (y_i - w_0 - w_1 x_i)^2 && [\text{Sum of derivatives}] \\&= \sum_{i=1}^N 2(y_i - w_0 - w_1 x_i) \frac{\partial}{\partial w_0} (y_i - w_0 - w_1 x_i) && [\text{Chain rule}] \\&= \sum_{i=1}^N -2(y_i - w_0 - w_1 x_i) && [\text{Derivative of inside}] \\&= 2Nw_0 + 2w_1 \sum_{i=1}^N x_i - 2 \sum_{i=1}^N y_i && [\text{Algebra}].\end{aligned}$$

Derivation (Continued)

Computing the partial derivative with respect to w_1 yields:

$$\begin{aligned}\frac{\partial}{\partial w_1} J(w_0, w_1) &= \frac{\partial}{\partial w_1} \sum_{i=1}^N (y_i - w_0 - w_1 x_i)^2 \\&= \sum_{i=1}^N \frac{\partial}{\partial w_1} (y_i - w_0 - w_1 x_i)^2 \\&= \sum_{i=1}^N 2(y_i - w_0 - w_1 x_i) \frac{\partial}{\partial w_1} (y_i - w_0 - w_1 x_i) \\&= \sum_{i=1}^N -2x_i (y_i - w_0 - w_1 x_i) \\&= 2w_0 \sum_{i=1}^N x_i + 2w_1 \sum_{i=1}^N x_i^2 - 2 \sum_{i=1}^N x_i y_i.\end{aligned}$$

Derivation (Continued)

Setting the partial derivative with respect to w_0 equal to zero and solving for w_0 yields:

$$\begin{aligned}\frac{\partial}{\partial w_0} J(w_0, w_1) = 0 &\Leftrightarrow 2Nw_0 + 2w_1 \sum_{i=1}^N x_i - 2 \sum_{i=1}^N y_i = 0 \\ &\Leftrightarrow 2Nw_0 = 2 \sum_{i=1}^N y_i - 2w_1 \sum_{i=1}^N x_i \\ &\Leftrightarrow w_0 = \frac{1}{N} \sum_{i=1}^N y_i - w_1 \frac{1}{N} \sum_{i=1}^N x_i \quad [\text{Dividing by } 2N] \\ &\Leftrightarrow w_0 = \bar{y} - w_1 \bar{x}\end{aligned}$$

where the last line uses sample means \bar{y} and \bar{x} defined as

$$\bar{x} = \frac{1}{N} \sum_{i=1}^N x_i \quad \text{and} \quad \bar{y} = \frac{1}{N} \sum_{i=1}^N y_i.$$

Derivation (Continued)

Setting the partial derivative with respect to w_1 equal to zero and solving for w_1 yields:

$$\begin{aligned}\frac{\partial}{\partial w_1} J(w_0, w_1) = 0 &\Leftrightarrow 2w_0 \sum_{i=1}^N x_i + 2w_1 \sum_{i=1}^N x_i^2 - 2 \sum_{i=1}^N x_i y_i = 0 \\&\Leftrightarrow w_0 \frac{1}{N} \sum_{i=1}^N x_i + w_1 \frac{1}{N} \sum_{i=1}^N x_i^2 = \frac{1}{N} \sum_{i=1}^N x_i y_i \quad [\text{Dividing by } 2N] \\&\Leftrightarrow (\bar{y} - w_1 \bar{x}) \bar{x} + w_1 \frac{1}{N} \sum_{i=1}^N x_i^2 = \frac{1}{N} \sum_{i=1}^N x_i y_i \quad [\text{Substitution}] \\&\Leftrightarrow \bar{x}\bar{y} - w_1 \bar{x}^2 + w_1 \frac{1}{N} \sum_{i=1}^N x_i^2 = \frac{1}{N} \sum_{i=1}^N x_i y_i \\&\Leftrightarrow w_1 \left(\frac{1}{N} \sum_{i=1}^N x_i^2 - \bar{x}^2 \right) = \frac{1}{N} \sum_{i=1}^N x_i y_i - \bar{x}\bar{y} \quad [\text{Rearranging}] \\&\Leftrightarrow w_1 = \frac{\frac{1}{N} \sum_{i=1}^N x_i y_i - \bar{x}\bar{y}}{\frac{1}{N} \sum_{i=1}^N x_i^2 - \bar{x}^2}.\end{aligned}$$

Least Squares Solution

After a little more algebra, we get the least squares solution as:

$$w_1^* = \frac{\sum_{i=1}^N (x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^N (x_i - \bar{x})^2} \quad \text{and} \quad w_0^* = \bar{y} - w_1^* \bar{x}.$$

With additional notation, we can rewrite the formula for w_1^* as

$$w_1^* = \frac{s_{xy}}{s_x} = r_{xy} \frac{s_y}{s_x}$$

where s_{xy} is the **sample covariance**, s_x and s_y are the respective **sample standard deviations**, and $r_{xy} = \frac{s_{xy}}{s_x s_y}$ is the **sample correlation**.

- If sample correlation is 0, then $w_1^* = 0$ (line is horizontal) and $w_0^* = \bar{y}$.
- If sample correlation is 1, then slope depends on magnitude of x and y as given by standard deviations.

Components of a Machine Learning System

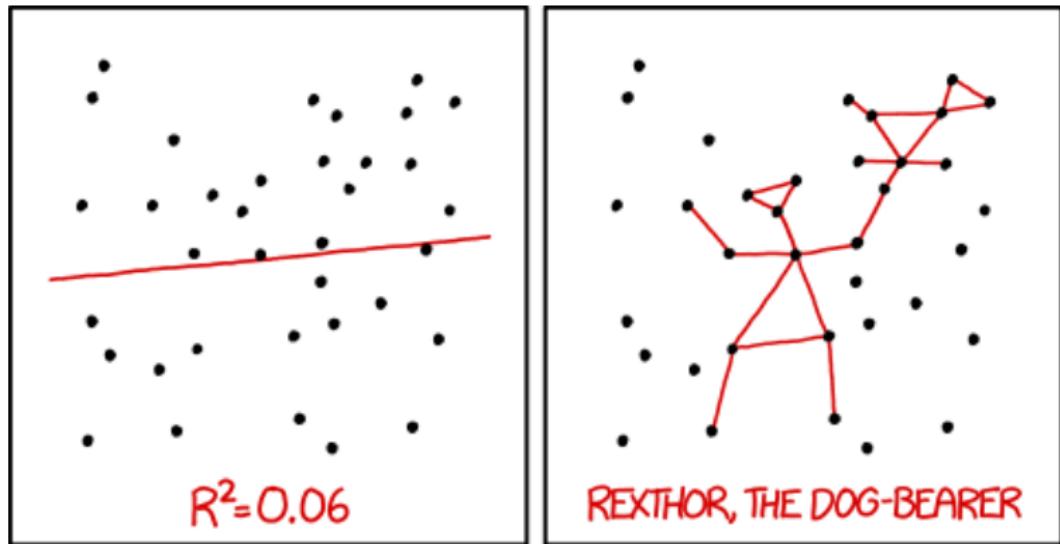
A (supervised) **machine learning system** consists of:

- ① **Data**: Numeric values describing aspects of real-world entities
- ② **Hypothesis space (model class)**: A set of hypotheses to consider
- ③ **Learning Algorithm**:
 - Loss function: penalty for misprediction on single example
 - Cost function: objective to be minimized (e.g., average loss)
 - Fitting method: method for finding a hypothesis function

Example of an ML system: Ordinary least squares regression

- ① Data: $\{(x_i, y_i)\}_{i=1}^N$
- ② Hypothesis space: $\mathcal{H} = \{h : \mathbb{R} \rightarrow \mathbb{R} \mid h(x) = w_0 + w_1x\}$
- ③ Learning algorithm:
 - Loss function: Squared error $\ell_2(y_i, h(x_i)) = (y_i - w_0 - w_1x_i)^2$
 - Cost function: total loss $J(w_0, w_1) = \sum_{i=1}^N (y_i - w_0 - w_1x_i)^2$
 - Fitting method: analytical solution

Comic of the Day



I DON'T TRUST LINEAR REGRESSIONS WHEN IT'S HARDER
TO GUESS THE DIRECTION OF THE CORRELATION FROM THE
SCATTER PLOT THAN TO FIND NEW CONSTELLATIONS ON IT.

<http://xkcd.com/1725/>

CS 457/557: Machine Learning

Lecture 03-2: Multiple Linear Regression

Lecture 03-2a: Multiple Linear Regression

Multiple Input Attributes

With a **single** input attribute, the data set consists of ordered pairs of scalar values, $\{(x_i, y_i)\}_{i=1}^N$.

With **multiple** input attributes, the data set consists of ordered pairs with a **vector input** and a scalar output $\{(\mathbf{x}_i, y_i)\}_{i=1}^N$.

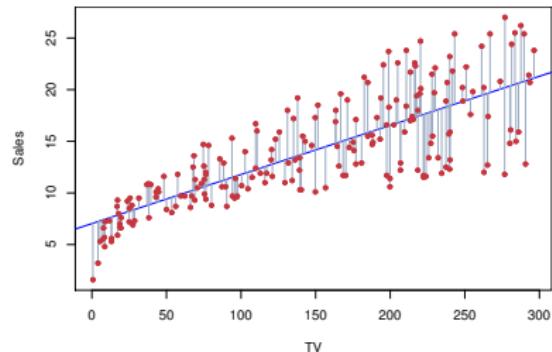
- There are p input attributes, so $\mathbf{x}_i \in \mathbb{R}^p$
- For each \mathbf{x}_i , we have $\mathbf{x}_i = (x_{i1}, x_{i2}, \dots, x_{ip})$ where x_{ij} is the value of attribute j for example i
(Note: Textbook swaps i and j here, so be careful with comparison)
- For generic vector \mathbf{x} , we have $\mathbf{x} = (x_1, x_2, \dots, x_p)$ where x_j is the value of attribute j for \mathbf{x}

We need to **expand the hypothesis space** to accommodate a larger number of input attributes!

From Lines to Planes to Hyperplanes

Simple linear regression considers the hypothesis space

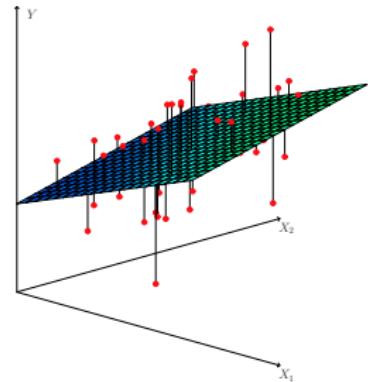
$$\mathcal{H} = \{h : \mathbb{R} \rightarrow \mathbb{R} \mid h(x) = w_0 + w_1 x\}.$$



Multiple linear regression considers the hypothesis space

$$\mathcal{H} = \{h : \mathbb{R}^p \rightarrow \mathbb{R} \mid h(\mathbf{x}) = w_0 + w_1 x_1 + \dots + w_p x_p\}$$

Taken from "An Introduction to Statistical Learning, with applications in R" (Springer, 2013) with permission from the authors: G. James, D. Witten, T. Hastie and R. Tibshirani.



Finding the Best Fitting Hyperplane

Multiple linear regression problem as an ML system:

- Data: $\{(\mathbf{x}_i, y_i)\}_{i=1}^N$, with p input attributes
- Hypothesis space:

$$\mathcal{H} = \left\{ h : \mathbb{R}^p \rightarrow \mathbb{R} \mid h(\mathbf{x}) = w_0 + \sum_{j=1}^p w_j x_j \right\}$$

- Learning algorithm:
 - Loss function: squared loss $\ell_2(y_i, h_{\mathbf{w}}(\mathbf{x}_i))$
 - Cost function: average loss

$$J(\mathbf{w}) = \frac{1}{N} \sum_{i=1}^N \ell_2(y_i, h_{\mathbf{w}}(\mathbf{x}_i)) = \frac{1}{N} \sum_{i=1}^N \left(y_i - w_0 - \sum_{j=1}^p w_j x_{ij} \right)^2$$

- Fitting method: ???

Towards an Analytical Solution

First, to make our notation more compact, we **augment** our data points with a zeroth attribute by setting $x_{i0} = 1$ for all i . Then:

$$h_{\mathbf{w}}(\mathbf{x}_i) = w_0 + \sum_{j=1}^p w_j x_{ij} = w_0 x_{i0} + \sum_{j=1}^p w_j x_{ij} = \sum_{j=0}^p w_j x_{ij}.$$

To find the optimal weight vector \mathbf{w}^* , we need to solve:

$$\min J(w_0, w_1, \dots, w_p) := \frac{1}{N} \sum_{i=1}^N \left(y_i - \sum_{j=0}^p w_j x_{ij} \right)^2$$

$$\text{s.t. } w_0, w_1, \dots, w_p \in \mathbb{R}$$

We'll derive an analytical solution for this!

Lecture 03-2a: Deriving an Analytical Solution

Partial Derivatives of the Objective Function

Definition

For a multi-variable function $g(v_1, v_2, \dots, v_k) = g(\mathbf{v})$, the **partial derivative** of g with respect to v_i , denoted $\frac{\partial g}{\partial v_i}$ or $\frac{\partial}{\partial v_i} g(\mathbf{v})$, is the derivative of g with the variables v_j ($j \neq i$) treated as constants.

Some calculus and algebra yields:

$$\begin{aligned}\frac{\partial}{\partial w_k} J(\mathbf{w}) &= \frac{\partial}{\partial w_k} \left(\frac{1}{N} \sum_{i=1}^N \left(y_i - \sum_{j=0}^p w_j x_{ij} \right)^2 \right) \\ &= \vdots \\ &= \frac{1}{N} \sum_{i=1}^N -2x_{ik} \left(y_i - \sum_{j=0}^p w_j x_{ij} \right)\end{aligned}$$

Solving the System of Equations

To find \mathbf{w}^* , we set each partial derivative of $J(\mathbf{w})$ equal to 0 and solve the resulting system of $p + 1$ equations:

$$\frac{1}{N} \sum_{i=1}^N -2x_{i0} \left(y_i - \sum_{j=0}^p w_j x_{ij} \right) = 0$$

$$\frac{1}{N} \sum_{i=1}^N -2x_{i1} \left(y_i - \sum_{j=0}^p w_j x_{ij} \right) = 0$$

$$\vdots = 0$$

$$\frac{1}{N} \sum_{i=1}^N -2x_{ip} \left(y_i - \sum_{j=0}^p w_j x_{ij} \right) = 0$$

This becomes **much simpler** to do with some vector and matrix notation.

The Normal Equations

In linear algebra terms, we have:

$$\mathbf{w} = \begin{pmatrix} w_0 \\ w_1 \\ \vdots \\ w_p \end{pmatrix}, \quad \mathbf{y} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{pmatrix}, \text{ and } \mathbf{X} = \begin{bmatrix} x_{1,0} & x_{1,1} & x_{1,2} & \dots & x_{1,p} \\ x_{2,0} & x_{2,1} & x_{2,2} & \dots & x_{2,p} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_{N,0} & x_{N,1} & x_{N,2} & \dots & x_{N,p} \end{bmatrix}.$$

(Treat data points as rows and stack to form a matrix).

The solution $\mathbf{w}^* = \arg \min_{\mathbf{w}} J(\mathbf{w})$ is obtained by solving the **normal equations**:

$$\mathbf{X}^T \mathbf{X} \mathbf{w}^* = \mathbf{X}^T \mathbf{y}.$$

Provided that $\mathbf{X}^T \mathbf{X}$ is invertible, we have:

$$\mathbf{w}^* = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y}.$$

Computational Costs

Matrix and vector operations have associated computational costs that the notation makes easy to hide. For n -dimensional vectors \mathbf{u} , \mathbf{v} and $n \times n$ dimensional matrices \mathbf{A} , \mathbf{B} :

- Computing dot product $\mathbf{u} \cdot \mathbf{v}$ is $O(n)$
- Computing matrix-vector product \mathbf{Av} is $O(n^2)$
- Computing matrix-matrix product \mathbf{AB} is $O(n^3)$
- Computing the inverse matrix \mathbf{A}^{-1} is $O(n^3)$
- Solving a system of equations $\mathbf{Au} = \mathbf{v}$ is $O(n^3)$
(but generally preferred over computing \mathbf{u} as $\mathbf{A}^{-1}\mathbf{v}$ for numerical stability)

Because of the expense of matrix inversion, other ways to find the solution are **desirable** – we'll see one in the next lecture.

Lecture 03-2c: Polynomial Regression

Back to Single Input Variable

Let's return to the single input attribute setting with $\{(x_i, y_i)\}_{i=1}^N$.

Simple linear regression considers a hypothesis space of linear functions:

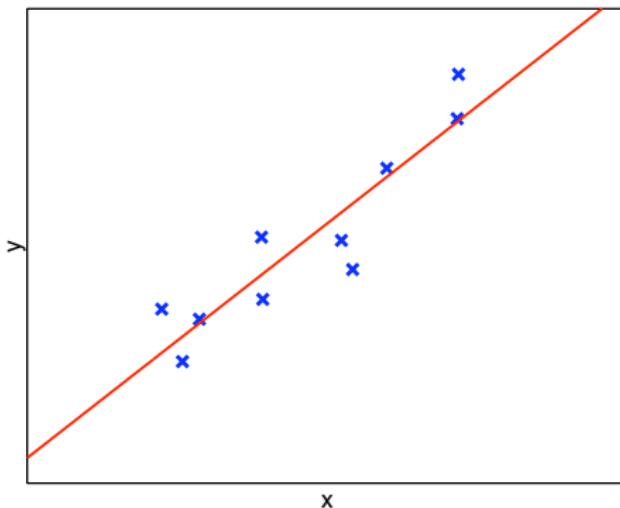
$$\mathcal{H} = \{h : \mathbb{R} \rightarrow \mathbb{R} \mid h(x) = w_0 + w_1 x\}.$$

However, often the true function f is nonlinear, so being able to consider **higher-order** hypothesis functions is **desirable**. For example, consider the hypothesis space of all polynomial functions of degree at most d :

$$\mathcal{H} = \left\{ h : \mathbb{R} \rightarrow \mathbb{R} \mid h(x) = w_0 + w_1 x + w_2 x^2 + \dots + w_d x^d \right\}$$

The value d is called a **model hyperparameter**: unlike the model parameters w_j which are found during learning, d must be specified by the user beforehand.

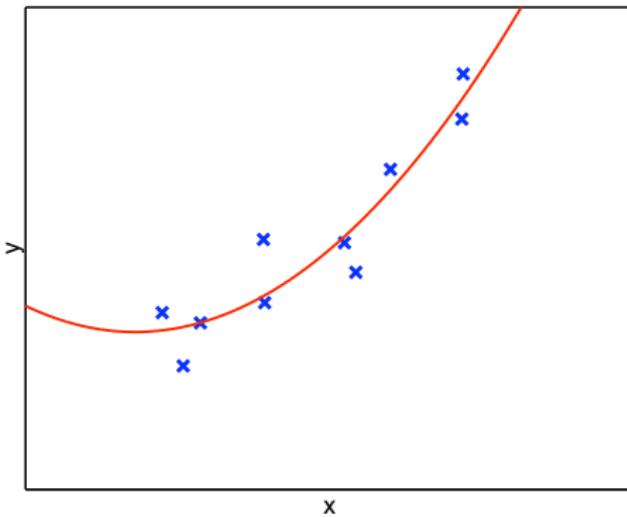
An Example



- ▶ For the data given, the best fit for a simple linear function of x is as follows:

$$h(x) \leftarrow y = 1.05 + 1.60x$$

An Order-2 Solution

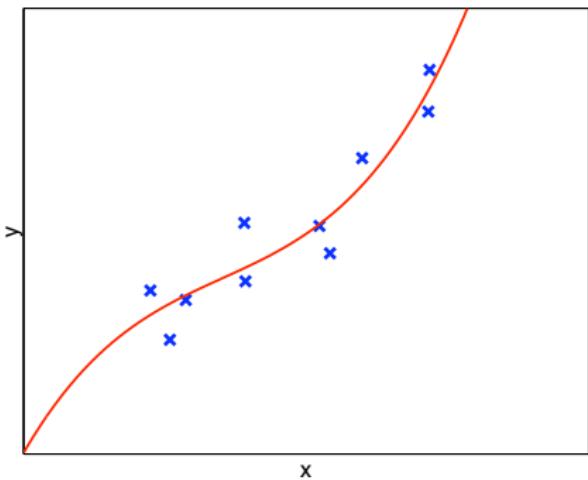


- With an order-2 function, we can fit our data somewhat better than with the original version

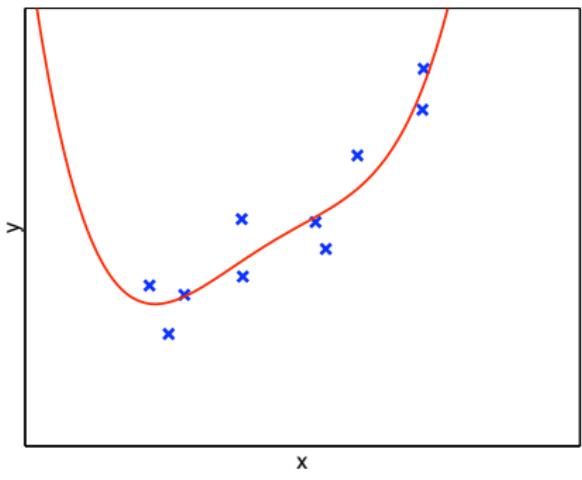
$$h(x) \leftarrow y = 0.73 + 1.74x + 0.68x^2$$

Higher-Order Fitting

Order-3 Solution

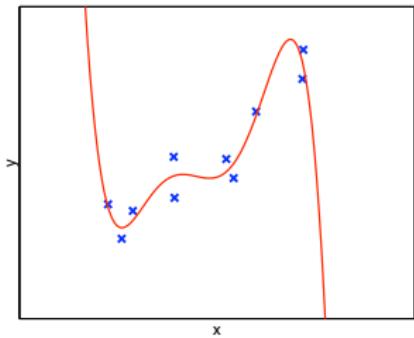


Order-4 Solution

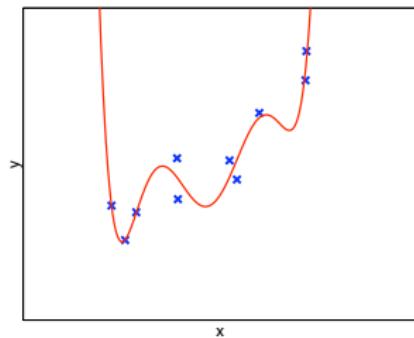


Even Higher-Order Fitting

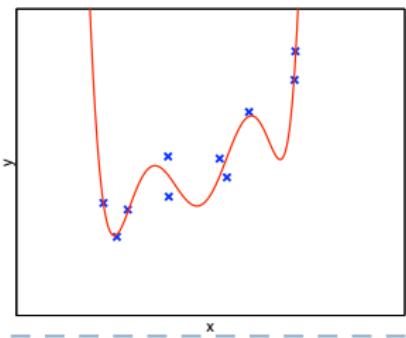
Order-5 Solution



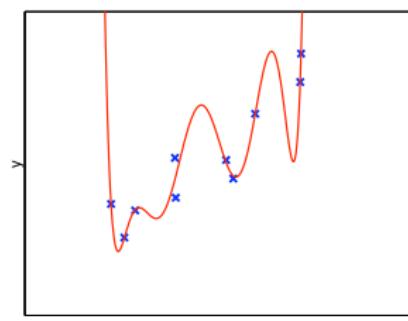
Order-6 Solution



Order-7 Solution



Order-8 Solution



Polynomial Regression

Polynomial regression with **model hyperparameter** $d \in \mathbb{Z}^+$:

- Data: $\{(x_i, y_i)\}_{i=1}^N$
- Hypothesis space:

$$\mathcal{H} = \left\{ h : \mathbb{R} \rightarrow \mathbb{R} \mid h(x) = w_0 + w_1x + w_2x^2 + \dots w_dx^d \right\}$$

- Learning algorithm:
 - Loss function: squared loss $\ell_2(y_i, h_{\mathbf{w}}(x_i))$
 - Cost function: average loss

$$J(\mathbf{w}) = \frac{1}{N} \sum_{i=1}^N \left(y_i - \sum_{j=0}^d w_j x_i^j \right)^2$$

- Fitting method: ???

Lecture 03-2d: Fitting a Polynomial Regression Model

Comparing Multiple Linear Regression and Polynomial Regression

Multiple linear regression:

- Data: $\{(\mathbf{x}_i, y_i)\}_{i=1}^N$, p input attributes
- Hypothesis space:

$$\mathcal{H} = \left\{ h : \mathbb{R}^p \rightarrow \mathbb{R} \mid h(\mathbf{x}) = \sum_{j=0}^p w_j x_j \right\}$$

- Learning algorithm:

- Loss function: squared loss
- Cost function: average loss

$$J(\mathbf{w}) = \frac{1}{N} \sum_{i=1}^N \left(y_i - \sum_{j=0}^p w_j x_{ij} \right)^2$$

- Fitting method: Analytical solution

Polynomial regression:

- Data: $\{(\mathbf{x}_i, y_i)\}_{i=1}^N$
- Hypothesis space:

$$\mathcal{H} = \left\{ h : \mathbb{R} \rightarrow \mathbb{R} \mid h(x) = \sum_{j=0}^d w_j x^j \right\}$$

- Learning algorithm:

- Loss function: squared loss
- Cost function: average loss

$$J(\mathbf{w}) = \frac{1}{N} \sum_{i=1}^N \left(y_i - \sum_{j=0}^d w_j x_i^j \right)^2$$

- Fitting method: ???

We can **reformulate** polynomial regression as multiple linear regression!

Transforming Polynomial Regression into Multiple Linear Regression

In polynomial regression with $\{(x_i, y_i)\}_{i=1}^N$, we have:

$$h(x_i) = w_0 + w_1 x_i + w_2 x_i^2 + \dots + w_d x_i^d = w_0 + \sum_{j=1}^d w_j x_i^j.$$

This is **polynomial in the features** but **linear in the parameters!**

We can **convert** a polynomial regression problem into a multiple linear regression problem by **creating a new data set**:

Let $\mathbf{x}_i = (x_i, x_i^2, \dots, x_i^d)$ for all i , so $x_{ij} = x_i^j$ for $j \in \{1, 2, \dots, d\}$.

Then in multiple linear regression with $\{(\mathbf{x}_i, y_i)\}_{i=1}^N$, we have:

$$h(\mathbf{x}_i) = w_0 + w_1 x_{i1} + w_2 x_{i2} + \dots + w_d x_{id} = w_0 + \sum_{j=1}^d w_j x_{ij}.$$

The optimal weight vector \mathbf{w}^* for the multiple linear regression problem is also optimal for the polynomial regression problem!

Lecture 03-2e: Extensions to Linear Regression

Including Interaction Terms

We can extend the ability to add higher-order terms involving a single attribute to higher-order terms involving several features.

For example, suppose we have $\{(\mathbf{x}_i, y_i)\}_{i=1}^N$ with two input attributes, and we want to use the hypothesis space

$$\mathcal{H} = \{h : \mathbb{R}^2 \rightarrow \mathbb{R} \mid h(\mathbf{x}) = w_0 + w_1x_1 + w_2x_2 + w_3x_1^2 + w_4x_2^2 + w_5x_1x_2\}.$$

We can use this hypothesis space by creating a new data set:

- Original data: $\mathbf{x}_i = (x_{i1}, x_{i2})$
- Modified data: $\mathbf{x}'_i = (x_{i1}, x_{i2}, x_{i1}^2, x_{i2}^2, x_{i1}x_{i2})$.

I.e., we augment each example with **new derived attributes** whose values are computed from the values of the original attributes.

Then we treat the derived attributes as regular attributes in a MLR problem. **Done!**

Issues of Scale

Polynomial regression with interaction terms can be scaled up to **any number and combination** of attributes, but can quickly become **unmanageable** if we try to add all possible terms!

With p raw attributes, there are:

- p first-order terms (e.g., x_j^1)
- $p + \binom{p}{2}$ second-order terms (e.g., $x_j^2, x_j x_{j'}$)
- $p + 2\binom{p}{2} + \binom{p}{3}$ third-order terms (e.g., $x_j^3, x_j^2 x_{j'}, x_j x_{j'} x_{j''}$)
- ...

Using too many features can also lead to **overfitting**.

- With N data points, a degree $N - 1$ polynomial can fit the data exactly with **no loss**!
- For polynomial regression, typically we use $d \leq 4$

Other Types of Terms

We can also add other types of terms to our hypothesis space, e.g.:

$$\mathcal{H} = \left\{ h : \mathbb{R} \rightarrow \mathbb{R} \mid h(x) = w_0 + w_1\sqrt{x} + w_2\frac{1}{x} + w_3e^{2x} + \dots \right\}.$$

These hypothesis functions are **still linear** in the parameters w , so we can handle this using multiple linear regression with some additional derived attributes!

What **can't** linear regression do? The following hypothesis functions won't work, as they are **not linear** in **all** the parameters w_k :

$$h(x) = w_0 + w_1e^{w_2x}$$

$$h(x) = w_0 + w_1 \sin(w_2x)$$

$$h(x) = w_0 + w_2x + w_3x^{w_4}$$

CS 457/557: Machine Learning

Lecture 03-2 Supplemental: Primer on Linear Algebra

Primer on Scalars, Vectors, and Matrices

- A **scalar** is a single real number (e.g., 2, -0.25 , $\frac{3}{7}$, π).
- A **vector** is a one-dimensional array of scalars.
 - A **row vector** has dimension $1 \times n$ and is written horizontally.

$$(1, 2) \quad (3, \frac{1}{3}, -2)$$

- A **column vector** has dimension $n \times 1$ and is written vertically.

$$\begin{pmatrix} 1 \\ 2 \end{pmatrix} \quad \begin{pmatrix} 3 \\ \frac{1}{3} \\ -2 \end{pmatrix}$$

- A **matrix** is a two-dimensional array of scalars. An $m \times n$ -dimensional matrix has m rows and n columns.

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \quad \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$$

Conventions for Notation

- Scalar variables are lower-case letters in italic font

$$x = 17 \quad y = \pi / \exp \quad \sigma = 10$$

- Vector variables are lower-case letters in bold-faced font

$$\mathbf{x} = (1, 2.4, -3.01) \quad \boldsymbol{\sigma} = (\sqrt{1}, \sqrt{2}, \dots, \sqrt{10})$$

- Matrix variables are upper-case letters in bold-faced font

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \quad \boldsymbol{\Sigma} = \begin{bmatrix} \pi & 0 \\ 0 & -\pi \end{bmatrix}$$

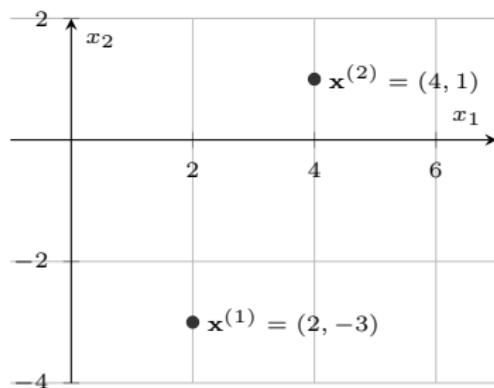
- Components of vector and matrix variables are indicated with corresponding lower-case letters in italic font and subscripts:

$$x_3 = -3.01 \quad x_2 = 2.4 \quad a_{1,2} = 2 \quad a_{2,3} = 6$$

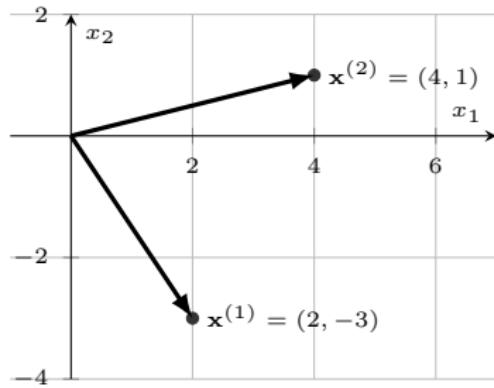
Visualizing Vectors

A vector can be viewed as either a point in p -dimensional space or a direction in p -dimensional space.

Vectors as points:



Vectors as directions:



Each component is a **coordinate**.

Each component is a **displacement**.

Vector Operations

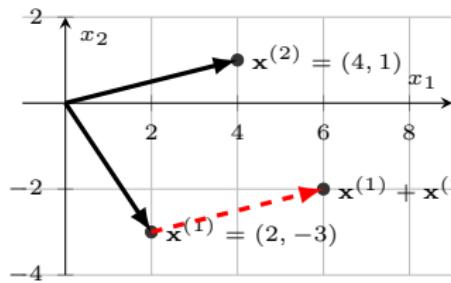
- Scalar multiples of a vector scale each component appropriately:

$$0.3\mathbf{x}^{(1)} = \begin{pmatrix} 0.3x_1^{(1)} \\ 0.3x_2^{(1)} \end{pmatrix} = \begin{pmatrix} 0.3 \cdot 2 \\ 0.3 \cdot (-3) \end{pmatrix} = \begin{pmatrix} 0.6 \\ -0.9 \end{pmatrix}$$

- Two vectors with the same dimensions can be added and subtracted component by component:

$$\mathbf{x}^{(1)} + \mathbf{x}^{(2)} = \begin{pmatrix} x_1^{(1)} + x_1^{(2)} \\ x_2^{(1)} + x_2^{(2)} \end{pmatrix} = \begin{pmatrix} 2 + 4 \\ -3 + 1 \end{pmatrix} = \begin{pmatrix} 6 \\ -2 \end{pmatrix}$$

Addition can be interpreted as a concatenation of the movements



Vector Operations

- The **dot product** of two p -vectors is:

$$\mathbf{x} \cdot \mathbf{y} = \sum_{j=1}^p x_j y_j \quad (\text{We also have } \mathbf{x} \cdot \mathbf{y} = \mathbf{y} \cdot \mathbf{x})$$

(This is almost like vector multiplication, but the result is a **scalar**, **not** a vector whose components are the products of the input vectors' components.)

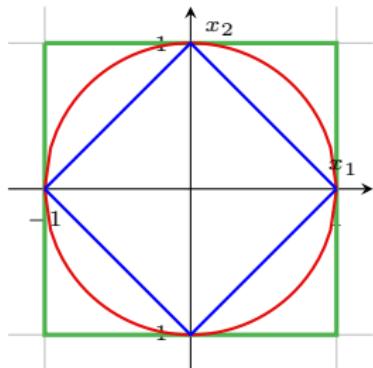
- The transpose of a column vector with dimension $p \times 1$ is a row vector with dimension $1 \times p$ (similar for transpose of a row vector):

$$\mathbf{x} = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} \quad \mathbf{x}^T = (1, 2, 3)$$

Vector Norms

For a vector $\mathbf{x} \in \mathbb{R}^p$:

- The **L2 norm** is $\|\mathbf{x}\| = \|\mathbf{x}\|_2 = \sqrt{\sum_{j=1}^p (x_j)^2} = \sqrt{\mathbf{x} \cdot \mathbf{x}}$
- The **L1 norm** is $\|\mathbf{x}\|_1 = \sum_{j=1}^p |x_j|$ (Manhattan distance)
- The **\mathbb{L}^∞ norm** is $\|\mathbf{x}\|_\infty = \max_{j=1}^p |x_j|$



- All points satisfying $\|\mathbf{x}\|_1 = 1$
- All points satisfying $\|\mathbf{x}\|_2 = 1$
- All points satisfying $\|\mathbf{x}\|_\infty = 1$

Matrix Basics

A **matrix** is a two-dimensional array of scalars. An $m \times n$ -dimensional matrix has m rows and n columns.

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \quad \mathbf{Q} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$$

- Matrices of like dimensions can be added together:

$$\begin{bmatrix} 1 & 2 & 3 \\ 0 & -1 & 2 \end{bmatrix} + \begin{bmatrix} 0 & 4 & -1 \\ 5 & 2 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 6 & 2 \\ 5 & 1 & 2 \end{bmatrix}$$

- Scalar multiplication of a matrix updates the components:

$$2 \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} = \begin{bmatrix} 2 & 4 \\ 6 & 8 \end{bmatrix}$$

Transpose of a Matrix

The **transpose** of a matrix \mathbf{A} , denoted \mathbf{A}^T , is the matrix that results from interchanging the rows and columns of \mathbf{A} :

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix}, \quad \mathbf{A}^T = \begin{bmatrix} a_{11} & a_{21} & \dots & a_{m1} \\ a_{12} & a_{22} & \dots & a_{m2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1n} & a_{2n} & \dots & a_{nm} \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}^T = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}, \quad \begin{bmatrix} 1 & 2 & 3 \\ 2 & 0 & 5 \\ 3 & 5 & 8 \end{bmatrix}^T = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 0 & 5 \\ 3 & 5 & 8 \end{bmatrix}$$

A matrix \mathbf{A} is **symmetric** if $\mathbf{A} = \mathbf{A}^T$.

Matrix Multiplication

If \mathbf{A} is $m \times n$, and \mathbf{B} is $n \times p$, then the matrix product $\mathbf{C} = \mathbf{AB}$ is a matrix with dimension $m \times p$. The entry c_{ij} is equal to the dot product of row i from \mathbf{A} with column j from \mathbf{B} :

$$c_{ij} = \sum_{k=1}^n a_{ik} \cdot b_{kj}$$

Example:

$$\begin{bmatrix} 1 & 3 \\ -1 & 2 \end{bmatrix} \begin{bmatrix} 5 & -1 & 0 \\ 2 & 9 & 4 \end{bmatrix} = \begin{bmatrix} 11 & 26 & 12 \\ -1 & 19 & 8 \end{bmatrix}$$

In order to do the multiplication, the number of columns in the left matrix **needs to equal** the number of rows in the right matrix. (So $\mathbf{AB} \neq \mathbf{BA}$ in general, and both products might not be well-defined.) With transpose, we have $(\mathbf{AB})^T = \mathbf{B}^T \mathbf{A}^T$ (assuming \mathbf{A} and \mathbf{B} have appropriate dimensions)

Matrix-Vector Multiplication

If \mathbf{A} is an $m \times n$ matrix, and \mathbf{x} is an $n \times 1$ column vector, then the product \mathbf{Ax} is an $m \times 1$ column vector:

$$\begin{aligned}\mathbf{Ax} &= \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n \\ a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n \\ \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \dots + a_{mn}x_n \end{pmatrix} \\ &= \begin{pmatrix} a_{11} \\ a_{21} \\ \vdots \\ a_{m1} \end{pmatrix} x_1 + \begin{pmatrix} a_{12} \\ a_{22} \\ \vdots \\ a_{m2} \end{pmatrix} x_2 + \dots + \begin{pmatrix} a_{1n} \\ a_{2n} \\ \vdots \\ a_{mn} \end{pmatrix} x_m\end{aligned}$$

So matrix-vector multiplication produces a **linear combination** of the columns of \mathbf{A} .

Vector-Matrix Multiplication

If \mathbf{y} is a $1 \times m$ row vector and \mathbf{A} is an $m \times n$ matrix, then the product $\mathbf{y}\mathbf{A}$ is a $1 \times n$ row vector:

$$\begin{aligned}\mathbf{y}\mathbf{A} &= (y_1 \quad y_2 \quad \dots \quad y_m) \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix} \\ &= ((y_1 a_{11} + \dots + y_m a_{m1}) \quad \dots \quad (y_1 a_{1n} + \dots + y_m a_{mn})) \\ &= \begin{pmatrix} y_1 a_{11} + y_2 a_{21} + \dots + y_m a_{m1} \\ y_1 a_{12} + y_2 a_{22} + \dots + y_m a_{m2} \\ \vdots \\ y_1 a_{1n} + y_2 a_{2n} + \dots + y_m a_{mn} \end{pmatrix}^T\end{aligned}$$

Vector-Vector Multiplication

If \mathbf{x} is a $1 \times n$ row vector and \mathbf{y} is an $n \times 1$ column vector, then the product \mathbf{xy} is a 1×1 matrix:

$$\mathbf{xy} = (x_1 \quad x_2 \quad \dots \quad x_n) \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} = [\sum_{i=1}^n x_i y_i] = [\mathbf{x} \cdot \mathbf{y}]$$

With a slight abuse of notation, we typically interpret the result as a scalar. We can also abuse notation slightly to rewrite the dot product as a multiplication, e.g.:

- If \mathbf{c} and \mathbf{x} are $n \times 1$ column vectors, then:

$$\mathbf{c} \cdot \mathbf{x} = \mathbf{c}^T \mathbf{x}.$$

The Inverse Matrix

With scalars, division is equivalent to multiplication by the inverse, e.g.:

$$a \div b = a \cdot \frac{1}{b} = a \cdot b^{-1} \quad (\text{provided that } b \neq 0).$$

We can do something similar with matrices, **provided** that they have suitable structure (analogous to not equaling 0). We need a few more things from linear algebra first:

- For any $n \in \mathbb{Z}^+$, the **identity matrix** \mathbf{I}_n or \mathbf{I} is the $n \times n$ matrix with 1s on the main diagonal and 0s elsewhere.
- For any $n \times 1$ vector \mathbf{b} , $\mathbf{I}\mathbf{b} = \mathbf{b}$.
- For any $n \times n$ matrix \mathbf{B} , $\mathbf{IB} = \mathbf{BI} = \mathbf{B}$.
- If \mathbf{A} is $n \times n$ and has appropriate structure, then there exists a unique inverse matrix \mathbf{A}^{-1} that satisfies $\mathbf{A}^{-1}\mathbf{A} = \mathbf{AA}^{-1} = \mathbf{I}$.

CS 457/557: Machine Learning

Lecture 03-3: Gradient Descent

Lecture 03-3a: Numerical Optimization

Numerical Optimization

In multiple linear regression, we have data set $\{(\mathbf{x}_i, y_i)\}_{i=1}^N$ with p attributes and augmented zeroth attribute with $x_{i0} = 1$ for all i .

Our cost function is given by:

$$J(w_0, w_1, \dots, w_p) = \frac{1}{N} \sum_{i=1}^N \left(y_i - \sum_{j=0}^p w_j x_{ij} \right)^2.$$

The **normal equations** give us an **analytical solution** to

$$\begin{aligned} & \min J(\mathbf{w}) \\ \text{s.t. } & \mathbf{w} \in \mathbb{R}^{p+1} \end{aligned}$$

This is nice in theory but doesn't work out as well in practice:

- The equations are slow to solve for large systems
- Alternate cost functions don't always have such analytical solutions

Numerical optimization can be used instead!

Numerical Optimization Terminology

The standard form for an **unconstrained optimization problem** is:

$$\begin{aligned} & \min g(\mathbf{v}) \\ \text{s.t. } & \mathbf{v} \in \mathbb{R}^n \end{aligned}$$

- \mathbf{v} is an n -dimensional **solution vector**
- $g : \mathbb{R}^n \rightarrow \mathbb{R}$ is the **objective function**

Analytical solutions are not always available (e.g., if g has complicated structure), so **numerical algorithms** attempt to **search** for the best solution in a systematic manner.

Constrained optimization places additional requirements on \mathbf{v} to determine a **feasible** solution, which generally makes the problem **harder!**

Improving Search Algorithms

Definition

An **improving search algorithm** (local search algorithm) is an algorithm that takes an initial solution, looks for a better solution nearby, and moves to it if found. The process repeats until no better solutions are found near the current solution.

Improving Search

```
 $v^{(0)} \leftarrow$  initial solution;  $t \leftarrow 0$ 
while a better solution  $v'$  exists near  $v^{(t)}$  do
     $v^{(t+1)} \leftarrow v'$ 
     $t \leftarrow t + 1$ 
```

- Solution in iteration t is $v^{(t)}$
- Notion of “near” is problem-dependent

Direction-Step Paradigm

For numeric optimization with real-valued variables, most improving search algorithms use the **direction-step** movement paradigm.

To move from a current solution $\mathbf{v}^{(t)}$ to a new solution $\mathbf{v}^{(t+1)}$:

- ① Identify a **move (search) direction** \mathbf{d}
- ② Identify a **step size** (learning rate) multiplier $\alpha > 0$
- ③ Set $\mathbf{v}^{(t+1)} = \mathbf{v}^{(t)} + \alpha\mathbf{d}$

The process **repeats** until no further improvement can be made.

Missing details:

- How do we identify a move direction?
- How do we determine how far we can go (step size)?

Improving Directions

Definition

A vector \mathbf{d} is an **improving direction** at solution \mathbf{v} if there exists an $\alpha_{\max} > 0$ such that $g(\mathbf{v} + \alpha\mathbf{d})$ is better than $g(\mathbf{v})$ for all $\alpha \in [0, \alpha_{\max}]$.

(Recall that $[a, b] = \{r \in \mathbb{R} \mid a \leq r \leq b\}$.)

How do we find this? Let's look in one dimension first to simplify.

Definition

For a differentiable function $g : \mathbb{R} \rightarrow \mathbb{R}$, the **derivative** of g , denoted g' , is the function that measures the rate of change in the output of g as its input increases:

$$g'(v) = \lim_{\epsilon \rightarrow 0} \frac{g(v + \epsilon) - g(v)}{\epsilon}$$

We can use the derivative to tell us which way to move!

Finding an Improving Direction in One Dimension

Suppose we are at a point $v^{(t)}$, with objective value $g(v^{(t)})$.

We want to minimize g .

If $g'(v^{(t)}) > 0$:

Increasing $v^{(t)}$ leads to an increase in $g(v^{(t)})$

Decrease $v^{(t)}$ to decrease g .

If $g'(v^{(t)}) < 0$:

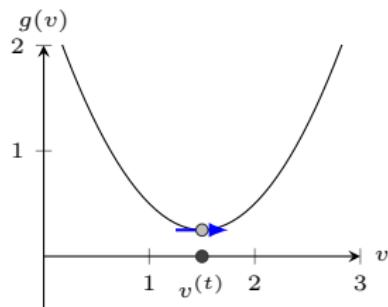
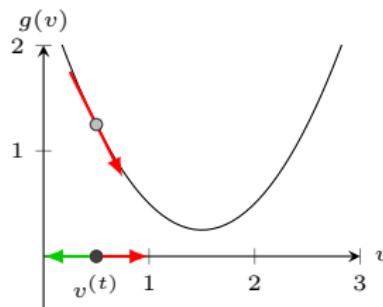
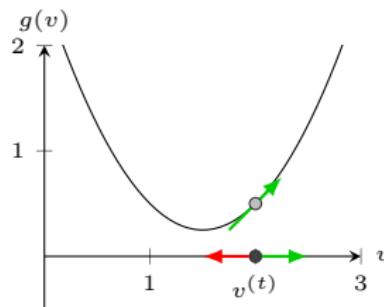
Increasing $v^{(t)}$ leads to a decrease in $g(v^{(t)})$.

Increase $v^{(t)}$ to decrease g .

If $g'(v^{(t)}) = 0$:

$v^{(t)}$ is a stationary point.

Don't change $v^{(t)}$ at all!



Moving in the opposite direction of the derivative decreases the function value.

Lecture 03-3b: Minimizing a Univariate Function

Minimizing a Univariate Function

Minimizing a Function of a Single Variable

procedure MINIMIZE($g : \mathbb{R} \rightarrow \mathbb{R}$, $v \in \mathbb{R}$, $\alpha \in \mathbb{R}^+$)

$v^{(0)} \leftarrow v$; $t \leftarrow 0$

while $g'(v^{(t)}) \neq 0$ **do** ▷ Stop at stationary point

$v^{(t+1)} \leftarrow v^{(t)} - \alpha g'(v^{(t)})$

$t \leftarrow t + 1$

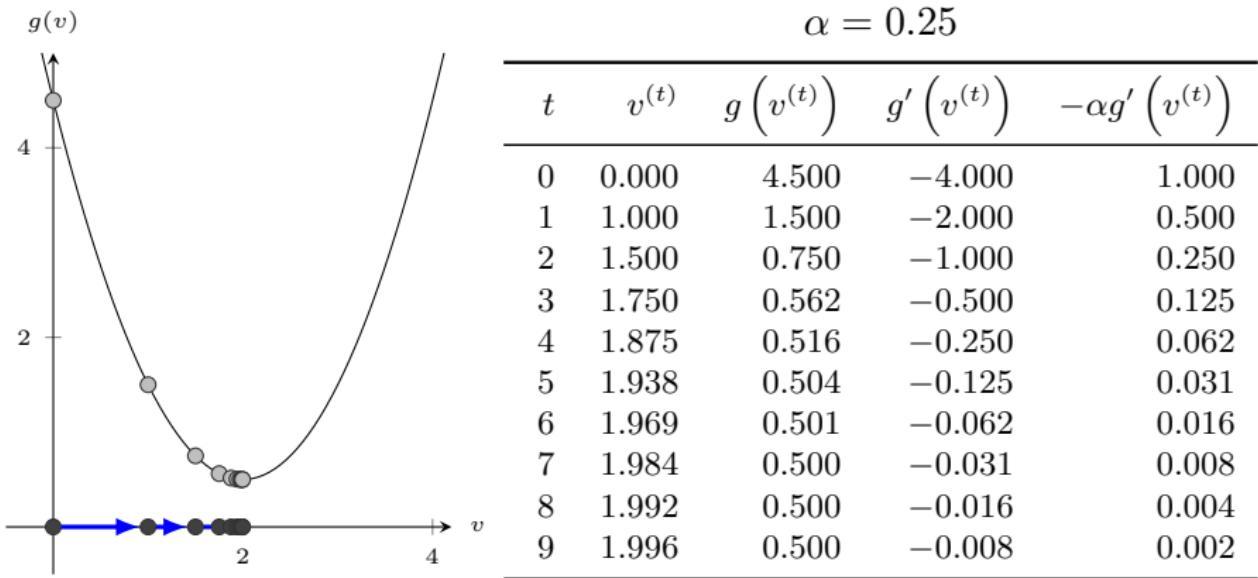
The value α is an **algorithm hyperparameter** set by the user.

Due to numerical issues, we typically include the following stopping conditions:

- A maximum iteration limit (e.g., 10,000 steps)
- Minimal change in solution and/or function value between iterations (e.g., $|g(v^{(t+1)}) - g(v^{(t)})| \leq 0.00001$)

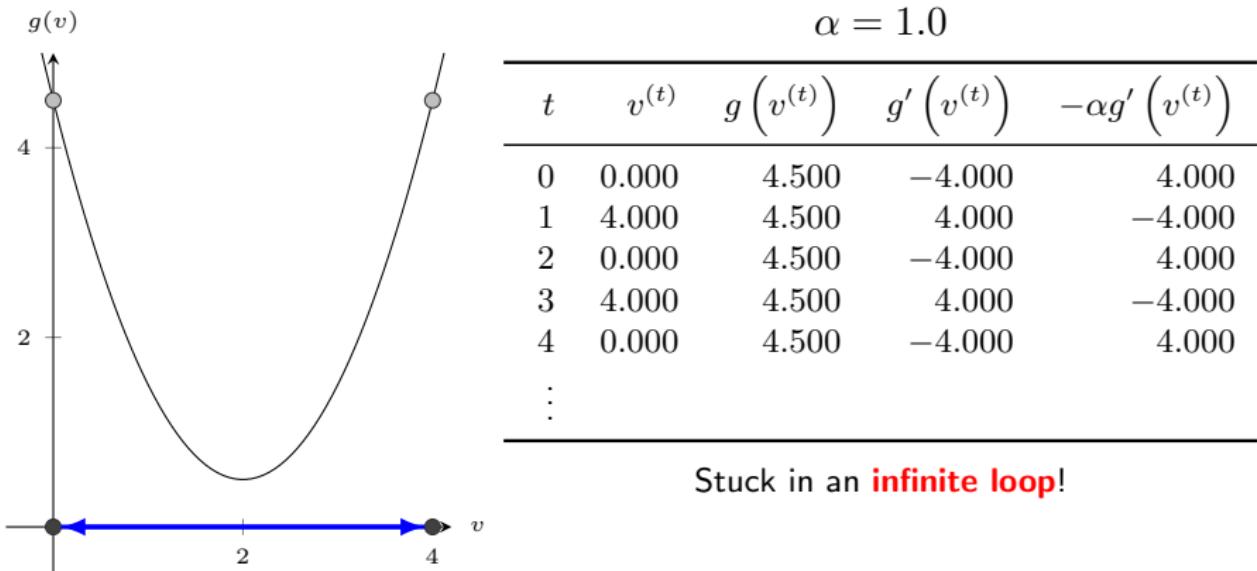
Example

Applying this algorithm to minimize $g(v) = v^2 - 4v + 4.5$ with $g'(v) = 2v - 4$ using initial solution $v^{(0)} = 0$:



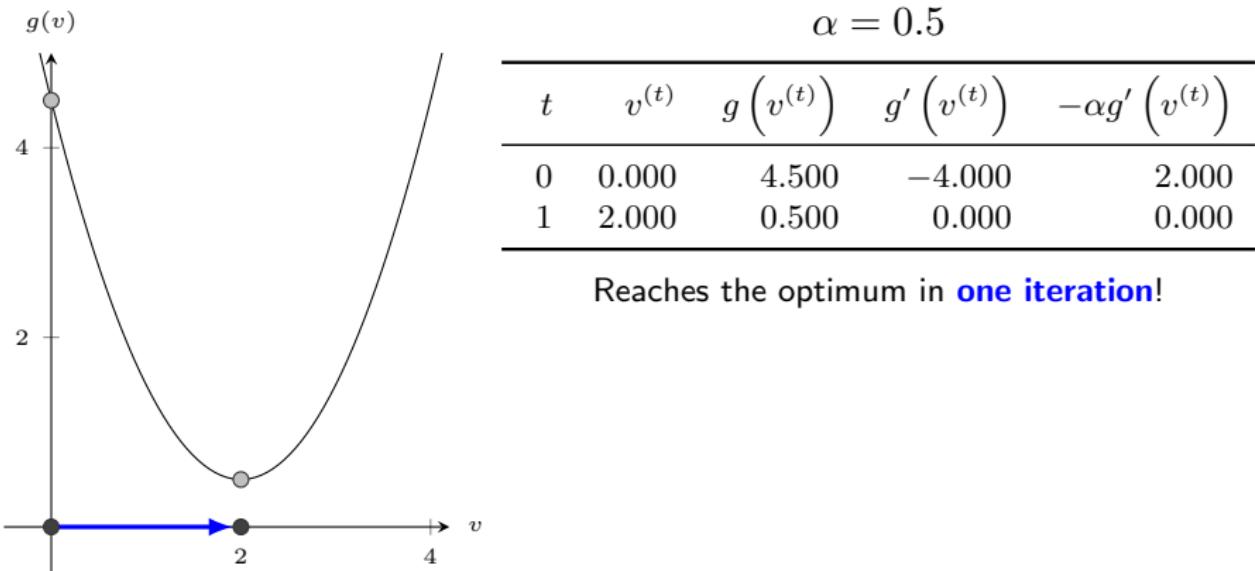
Example with $\alpha = 1.0$

Using the same function and initial solution but a different value of α changes the behavior:



Example with $\alpha = 0.5$

Using the same function and initial solution but a different value of α changes the behavior:



Algorithm Hyperparameters and Rates of Convergence

The **algorithm hyperparameter** α affects algorithm behavior:

- Different values for α affect rate of convergence
 - If α is too small, then convergence will be slow
 - If α is too large, then convergence may be lost
 - In our example, the behavior with $\alpha = 0.5$ is **not** the norm
- α is typically chosen to be **reasonably** small as first derivatives provide **local information** only
 - Common options are 0.001, 0.003, 0.01, 0.03, 0.1, 0.3
- **Adaptive algorithms** update α during the search:
 - If progress is too slow, increase α by multiplicative factor
 - If objective value increases, then step size is too large so decrease α by multiplicative factor

Lecture 03-3c: Moving to Higher Dimensions

Finding an Improving Direction in Higher Dimensions

In higher dimensions, the function $g : \mathbb{R}^n \rightarrow \mathbb{R}$ doesn't have a **single derivative**, so instead we look at its **partial derivatives**.

If $\frac{\partial}{\partial v_k} g(\mathbf{v}^{(t)}) > 0$:

Increasing $\mathbf{v}^{(t)}$ in dimension k leads to an **increase** in $g(\mathbf{v}^{(t)})$.

So **decrease** the k th component of $\mathbf{v}^{(t)}$ to **decrease** $g(\mathbf{v}^{(t)})$.

If $\frac{\partial}{\partial v_k} g(\mathbf{v}^{(t)}) < 0$:

Increasing $\mathbf{v}^{(t)}$ in dimension k leads to an **decrease** in $g(\mathbf{v}^{(t)})$.

So **increase** the k th component of $\mathbf{v}^{(t)}$ to **decrease** $g(\mathbf{v}^{(t)})$.

If $\frac{\partial}{\partial v_k} g(\mathbf{v}^{(t)}) = 0$:

Increasing $\mathbf{v}^{(t)}$ in dimension k leads to **no change** in $g(\mathbf{v}^{(t)})$.

So **don't change** the k th component of $\mathbf{v}^{(t)}$.

Minimizing a Multivariate Function

Minimizing a Function of Multiple Variables

procedure MINIMIZE($g : \mathbb{R}^n \rightarrow \mathbb{R}$, $\mathbf{v} \in \mathbb{R}^n$, $\alpha \in \mathbb{R}^+$)

$\mathbf{v}^{(0)} \leftarrow \mathbf{v}; t \leftarrow 0$

while stopping conditions not met **do**

for each $k \in \{1, 2, \dots, n\}$ **do**

$$v_k^{(t+1)} \leftarrow v_k^{(t)} - \alpha \left[\frac{\partial}{\partial v_k} g(\mathbf{v}^{(t)}) \right]$$

$t \leftarrow t + 1$

This can be written more compactly using additional **vector** notation!

Gradients

Definition

For a function $g : \mathbb{R}^n \rightarrow \mathbb{R}$, the **gradient** of g , denoted ∇g , is the vector-valued function (i.e., a function that returns a vector) that returns the vector of partial derivatives of g with respect to each v_k evaluated at \mathbf{v} :

$$\nabla g(\mathbf{v}) = \begin{pmatrix} \frac{\partial}{\partial v_1} g(\mathbf{v}) \\ \frac{\partial}{\partial v_2} g(\mathbf{v}) \\ \vdots \\ \frac{\partial}{\partial v_k} g(\mathbf{v}) \end{pmatrix}$$

- The gradient describes the **shape** of the function in terms of the rates of change in output value per change in each input variable
- The gradient points in the direction of **steepest ascent**

Example of Gradients

Let $g : \mathbb{R}^2 \rightarrow \mathbb{R}$ be defined as $g(v_1, v_2) = 3v_1^2 + v_1v_2 + 2v_2 - 4v_2^2$.

Then:

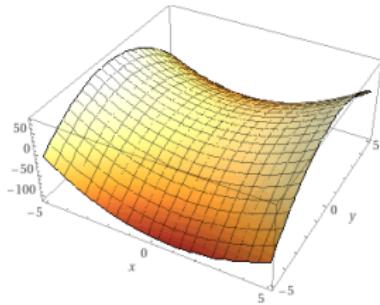
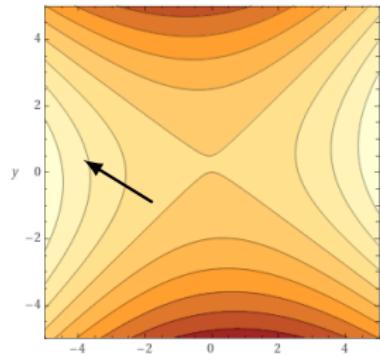
$$\frac{\partial}{\partial v_1} g(v_1, v_2) = 6v_1 + v_2$$

$$\frac{\partial}{\partial v_2} g(v_1, v_2) = v_1 + 2 - 8v_2$$

So $\nabla g(v_1, v_2) = \begin{pmatrix} 6v_1 + v_2 \\ v_1 + 2 - 8v_2 \end{pmatrix}$.

Evaluating the gradient at $(-2, -1)$ yields:

$$\nabla g(-2, -1) = \begin{pmatrix} 6(-2) - 1 \\ -2 + 2 - 8(-1) \end{pmatrix} = \begin{pmatrix} -13 \\ 8 \end{pmatrix}.$$



Improving Directions and Gradient Descent

For $g : \mathbb{R}^n \rightarrow \mathbb{R}$, the gradient ∇g can be used as an improving direction:

- If maximizing, then $\nabla g(\mathbf{v})$ is an improving direction
- If minimizing, then $-\nabla g(\mathbf{v})$ is an improving direction

Definition

Gradient descent is a first-order iterative optimization algorithm that minimizes a differentiable function by repeatedly moving in the direction of the negative gradient.

Gradient Descent

```
procedure GRADIENTDESCENT( $g : \mathbb{R}^n \rightarrow \mathbb{R}$ ,  $\mathbf{v} \in \mathbb{R}^n$ ,  $\alpha \in \mathbb{R}^+$ )
     $\mathbf{v}^{(0)} \leftarrow \mathbf{v}$ ;  $t \leftarrow 0$ 
    while stopping condition not met do
         $\mathbf{v}^{(t+1)} \leftarrow \mathbf{v}^{(t)} - \alpha \nabla g(\mathbf{v}^{(t)})$ 
         $t \leftarrow t + 1$ 
```

Stopping Conditions in Gradient Descent

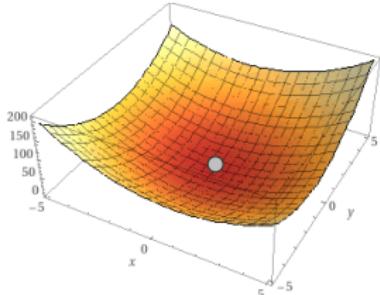
Using a stopping condition of minimal change between consecutive solutions, we have:

$$\left| \mathbf{v}^{(t+1)} - \mathbf{v}^{(t)} \right| \leq \epsilon \Leftrightarrow \left| -\alpha \nabla g(\mathbf{v}^{(t)}) \right| \leq \epsilon$$

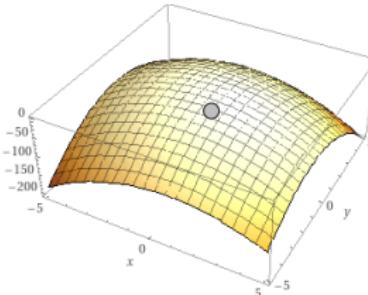
For $\alpha > 0$ and ϵ sufficiently small, this means $\nabla g(\mathbf{v}^{(t)}) \approx \mathbf{0}$, so gradient descent stops at a **stationary point**.

In higher dimensions, a stationary point can be:

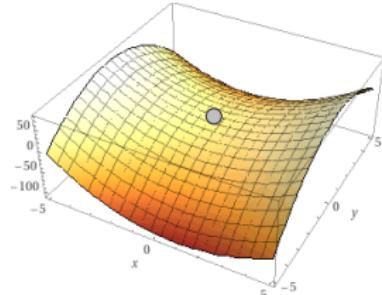
a local minimum



a local maximum



a saddle point



Cool Picture of Gradient Descent

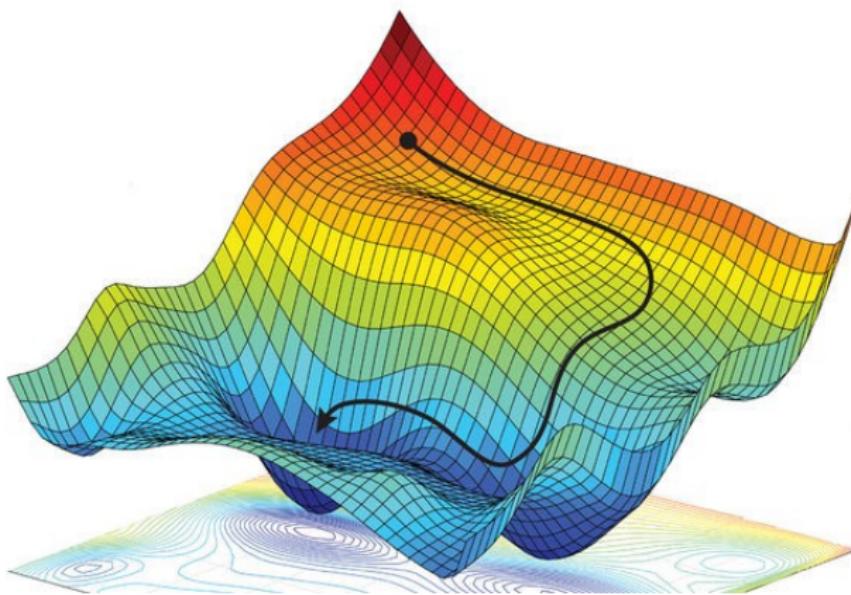


Image source: obtained here, which cites an earlier source with a broken link

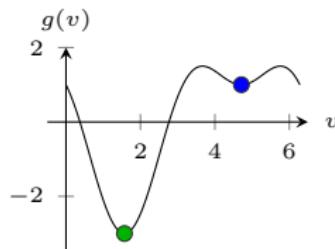
Local and Global Minima

Definition

A solution $\mathbf{v} \in \mathbb{R}^n$ to an unconstrained optimization problem is a **local minimum** if there exists an $\epsilon > 0$ such that for any $\mathbf{u} \in \mathbb{R}^n$ with $\|\mathbf{v} - \mathbf{u}\|_2 \leq \epsilon$, $g(\mathbf{v}) \leq g(\mathbf{u})$.

Definition

A solution \mathbf{v} is a **global minimum** if $g(\mathbf{v}) \leq g(\mathbf{u})$ for all $\mathbf{u} \in \mathbb{R}^n$.



Gradient descent converges to a **local minimum**.

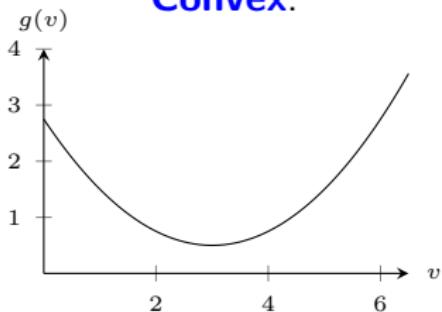
Convexity

Definition

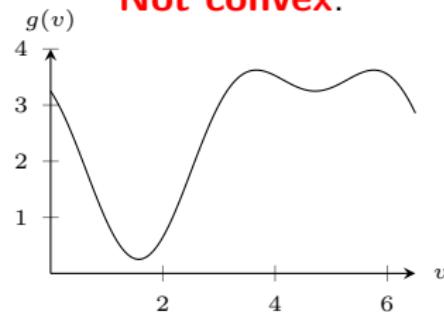
A function $g : \mathbb{R} \rightarrow \mathbb{R}$ is **convex** if and only if:

$$\forall v_1, v_2 \in \mathbb{R}, \forall \lambda \in [0, 1], g(\lambda v_1 + (1 - \lambda)v_2) \leq \lambda g(v_1) + (1 - \lambda)g(v_2).$$

Convex:



Not convex:



- For convex functions, local minima are global minima!
- For non-convex functions, local minima might not be global minima.

Lecture 03-3d: Gradient Descent for Multiple Linear Regression

Using Gradient Descent for Multiple Linear Regression

Recall the multiple linear regression optimization problem with average loss as the cost function and augmented zeroth attribute:

$$\begin{aligned} \min J(\mathbf{w}) &:= \frac{1}{N} \sum_{i=1}^N \left(y_i - \sum_{j=0}^p w_j x_{ij} \right)^2 \\ \text{s.t. } w_0, w_1, \dots, w_p &\in \mathbb{R} \end{aligned}$$

The function J is convex, so gradient descent can find the minimum!

We need the partial derivatives of $J(\mathbf{w})$ with respect to each weight w_k :

$$\frac{\partial}{\partial w_k} J(\mathbf{w}) = \frac{1}{N} \sum_{i=1}^N -2x_{ik} \left(y_i - \sum_{j=0}^p w_j x_{ij} \right).$$

Gradient Descent for Multiple Linear Regression

Algorithm 1 Gradient Descent for Multiple Linear Regression

procedure GRADIENTDESCENT($\{(\mathbf{x}_i, y_i)\}_{i=1}^N, \alpha$)

$w_0^{(0)}, w_1^{(0)}, \dots, w_p^{(0)} \leftarrow 0$ ▷ Initialize weights to all zeroes
 $t \leftarrow 0$

while stopping conditions not met **do**

for each $k \in \{0, 1, 2, \dots, p\}$ **do**

$$w_k^{(t+1)} = w_k^{(t)} - \alpha \left[\frac{1}{N} \sum_{i=1}^N -2x_{ik} \left(y_i - \sum_{j=0}^p w_j^{(t)} x_{ij} \right) \right]$$

$t \leftarrow t + 1$

With vector notation, we can write this more compactly (left as an exercise).

Efficiency of Gradient Descent for Multiple Linear Regression

In gradient descent, the update step for each weight w_k is

$$w_k^{(t+1)} = w_k^{(t)} - \alpha \left[\frac{1}{N} \sum_{i=1}^N -2x_{ik} \left(y_i - \sum_{j=0}^p w_j^{(t)} x_{ij} \right) \right].$$

Computing the gradient requires $O(Np)$ work **per iteration**, which gets expensive for large N . (A naive implementation is $O(Np^2)$, but this involves redundant calculations.)

If only a **single** data point, say \mathbf{x}_i , is used to *estimate* the gradient, the update step for each weight w_k becomes:

$$w_k^{(t+1)} = w_k^{(t)} - \alpha \left[-2x_{ik} \left(y_i - \sum_{j=0}^p w_j^{(t)} x_{ij} \right) \right].$$

This **only requires** $O(p)$ work per iteration.

Stochastic Gradient Descent

Using a **single** randomly selected data point to estimate the gradient in each iteration gives the (simple) **stochastic gradient descent** algorithm:

Algorithm 2 (Simple) Stochastic Gradient Descent

```
procedure STOCHASTICGRADIENTDESCENT(  $\{(\mathbf{x}_i, y_i)\}_{i=1}^N$ ,  $\alpha$  )
     $w_0^{(0)}, w_1^{(0)}, \dots, w_p^{(0)} \leftarrow 0$ 
     $t \leftarrow 0$ 
    while stopping conditions not met do
         $i \leftarrow$  random value from  $\{1, 2, \dots, N\}$ 
        for each  $k \in \{0, 1, 2, \dots, p\}$  do
             $w_k^{(t+1)} = w_k^{(t)} - \alpha \left[ -2x_{ik} \left( y_i - \sum_{j=0}^p w_j^{(t)} x_{ij} \right) \right]$ 
         $t \leftarrow t + 1$ 
```

Comparing Gradient Descent and Stochastic Gradient Descent

Batch gradient descent (GD)

- Uses all data points per iteration to calculate the gradient
- Good representation of gradient
- Each iteration is slow: $O(Np)$
- Takes fewer iterations to stop

Stochastic gradient descent (SGD)

- Uses a single data point per iteration to *estimate* the gradient
- Poor approximation of gradient
- Each iteration is fast: $O(p)$
- Takes more iterations to stop

Because we only use the gradient for small steps, the **advantages** of SGD generally outweigh its **disadvantages**.

Modifications to Stochastic Gradient Descent

To ensure that **all** data points are used, stochastic gradient descent is often implemented by repeatedly shuffling the data points and iterating through them in shuffled order:

Algorithm 3 Stochastic Gradient Descent

```
procedure STOCHASTICGRADIENTDESCENT(  $\{(\mathbf{x}_i, y_i)\}_{i=1}^N, \alpha$  )  
     $w_0^{(0)}, w_1^{(0)}, \dots, w_p^{(0)} \leftarrow 0$   
     $t \leftarrow 0$   
     $L \leftarrow (1, 2, \dots, N)$   
    while stopping conditions not met do  
        Randomly permute the elements of  $L$   
        for each  $i \in L$  do  
            for each  $k \in \{0, 1, 2, \dots, p\}$  do  
                 $w_k^{(t+1)} = w_k^{(t)} - \alpha \left[ -2x_{ik} \left( y_i - \sum_{j=0}^p w_j^{(t)} x_{ij} \right) \right]$   
         $t \leftarrow t + 1$ 
```

Mini-Batch Gradient Descent

A compromise between using all data points or using just a single data point is to use m data points to estimate the derivative at each iteration, where $1 \leq m \leq n$. This is **mini-batch gradient descent**:

Algorithm 4 Mini-Batch Gradient Descent

procedure MINIBATCHGRADIENTDESCENT($\{(\mathbf{x}_i, y_i)\}_{i=1}^N, \alpha, m$)

$w_0^{(0)}, w_1^{(0)}, \dots, w_p^{(0)} \leftarrow 0$

$t \leftarrow 0$

while stopping conditions not met **do**

 Divide $\{1, 2, \dots, N\}$ into random mini-batches of m data points

for each mini-batch B **do**

for each $k \in \{0, 1, 2, \dots, p\}$ **do**

$$w_k^{(t+1)} = w_k^{(t)} - \alpha \left[\frac{1}{|B|} \sum_{i \in B} -2x_{ik} \left(y_i - \sum_{j=0}^p w_j^{(t)} x_{ij} \right) \right]$$

$t \leftarrow t + 1$

For More Reading

General background:

- https://www.textbook.ds100.org/ch/11/gradient_descent.html

Gradient descent improvements:

- <https://ruder.io/optimizing-gradient-descent/index.html>
- <https://arxiv.org/abs/1609.04747>

Articles from *Towards Data Science* website:

- Gradient Descent Intro
- Gradient Descent Variants
- Adaptive learning

Unconstrained optimization:

- https://www.sandia.gov/~ktcarlb/opt_class/OPT_Lecture2.pdf

CS 457/557: Machine Learning

Lecture 03-4: Regularization and Feature Engineering

Lecture 03-4a: Regularization

Recap: Some Regression Models

Simple linear regression:

$$h(x_i) = w_0 + w_1 x_i$$

Polynomial regression:

$$h(x_i) = w_0 + w_1 x_i + w_2 x_i^2 + \dots + w_p x_i^p$$

Multiple linear regression:

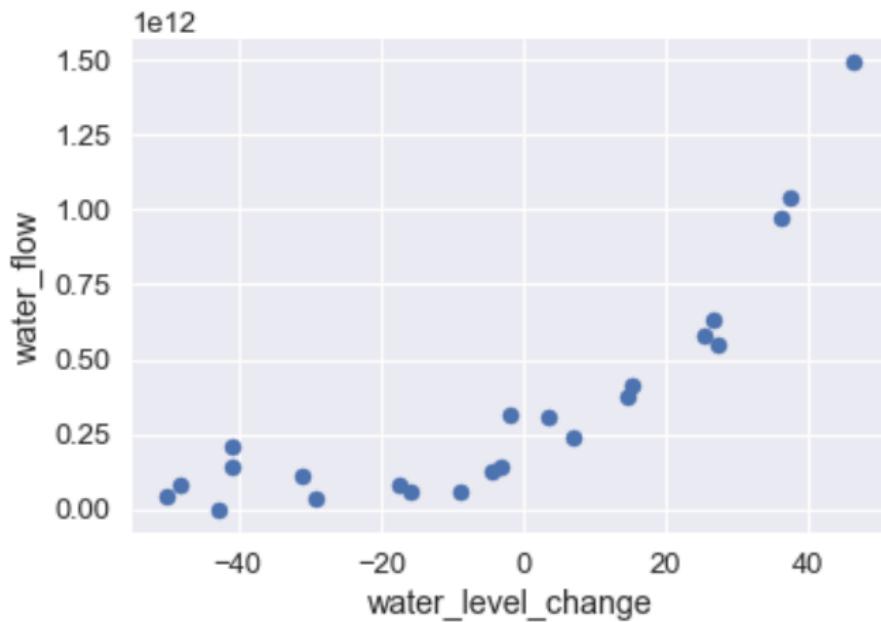
$$h(\mathbf{x}_i) = w_0 + w_1 x_{i1} + w_2 x_{i2} + \dots + w_p x_{ip}$$

Regression with two variables and second-order terms:

$$h(\mathbf{x}_i) = w_0 + w_1 x_{i1} + w_2 x_{i2} + w_3 x_{i1}^2 + w_4 x_{i2}^2 + w_5 x_{i1} x_{i2}$$

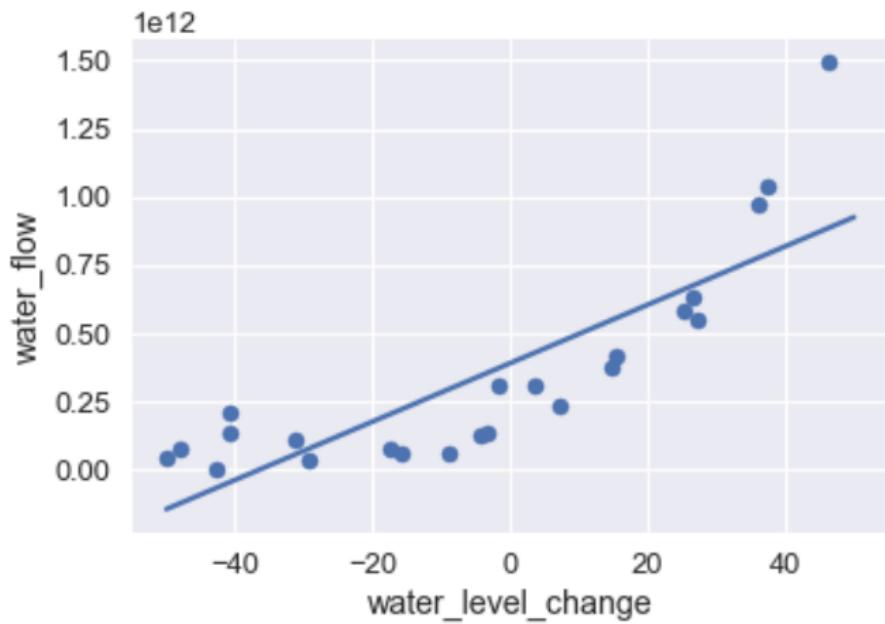
Even though we can consider hypothesis functions with higher-order terms, sometimes a **simpler** hypothesis function is **better**!

Motivating Example

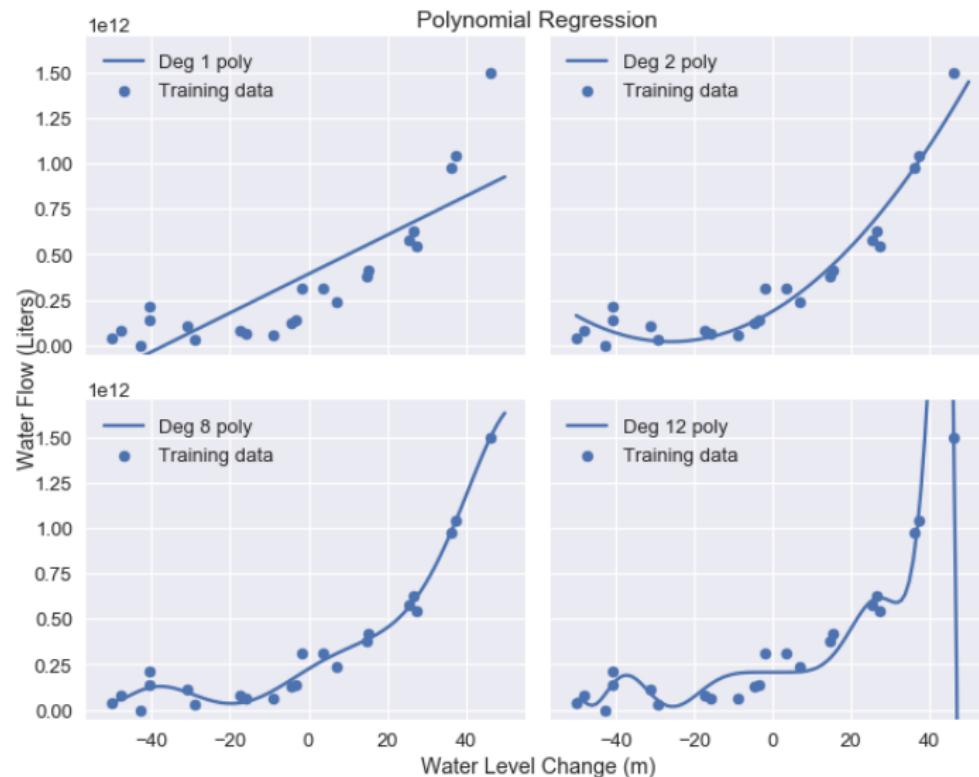


(Adapted from https://www.textbook.ds100.org/ch/16/reg_intuition.html)

Fitting a Linear Hypothesis Function



Using Hypothesis Functions with Higher-Order Terms



Limiting Complexity

Definition

Regularization incorporates secondary terms into the cost function in order to limit the complexity of the hypothesis function.

Regularization attempts to **balance** the **expressiveness** of a hypothesis function h with its **complexity**.

Instead of searching for

$$h^* = \arg \min_{h \in \mathcal{H}} \text{Loss}(h),$$

we want to find

$$h^* = \arg \min_{h \in \mathcal{H}} \text{Loss}(h) + \lambda \cdot \text{Complexity}(h)$$

where λ is a scaling factor.

Ridge Regression

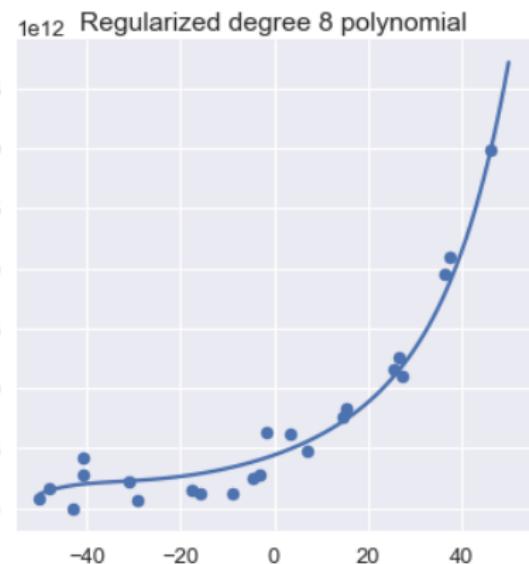
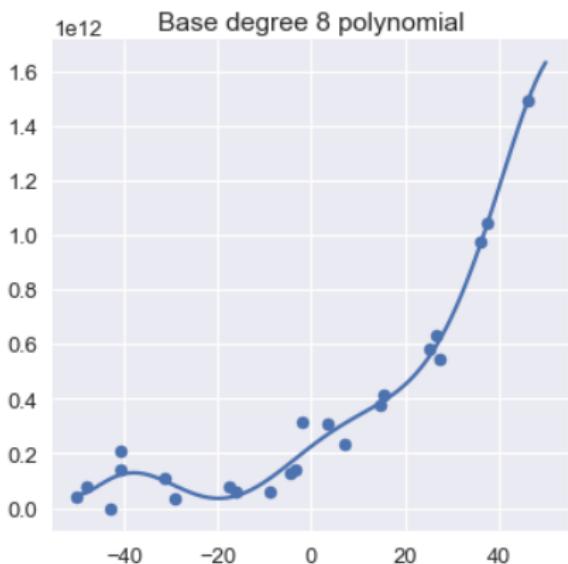
In regularized regression, the **complexity** of a hypothesis function h_w is based on the **magnitudes** of its weights w .

Ridge regression (Tikhonov regularization) uses the following **regularized** cost function for multiple linear regression:

$$J(w) = \frac{1}{N} \sum_{i=1}^N \left(y_i - w_0 - \sum_{j=1}^p w_j x_{ij} \right)^2 + \lambda \sum_{j=1}^p w_j^2$$

- Penalty term $\lambda \sum_{j=1}^p w_j^2$ penalizes large weights
- Intercept w_0 is **not** penalized
- λ is a **model hyperparameter** that needs to be specified by the user prior to learning

Using Regularization



Lecture 03-4b: Regularized Regression

Considerations with Ridge Regression

To find optimal weights \mathbf{w}^* in ridge regression, we must solve a **different** optimization problem compared to standard multiple linear regression.

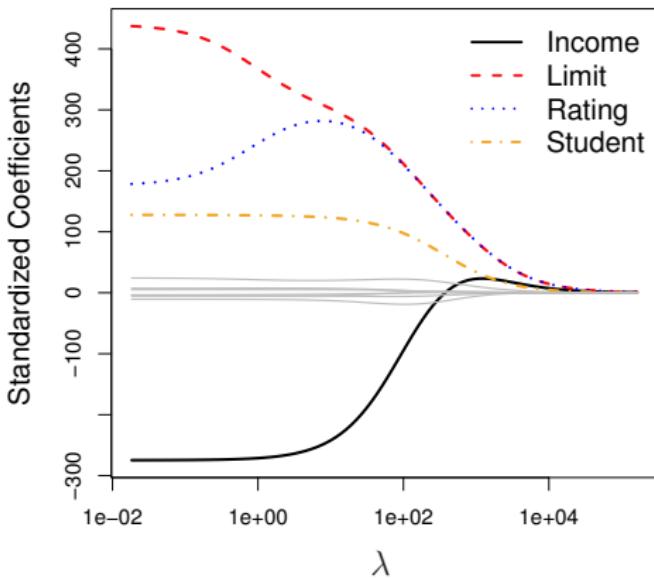
Some options:

- ① Analytical solution adapted from the normal equations
- ② Gradient descent with an appropriately modified gradient
(cost function remains convex!)

Different choices of λ lead to different optimal solutions \mathbf{w}^* :

- $\lambda = 0$ **reduces to** standard multiple linear regression
- As $\lambda \rightarrow \infty$, the weights get **increasingly penalized** and driven to 0

Ridge Regression in Action



Data set involves predicting if a person will default on a debt.

- Income, Limit, Rating, and Student are core attributes
- The gray lines are other less prominent attributes

As λ **increases**, the overall magnitude of the coefficients **decrease**.

Taken from “An Introduction to Statistical Learning, with applications in R” (Springer, 2013)
with permission from the authors: G. James, D. Witten, T. Hastie and R. Tibshirani.

Alternative Penalties

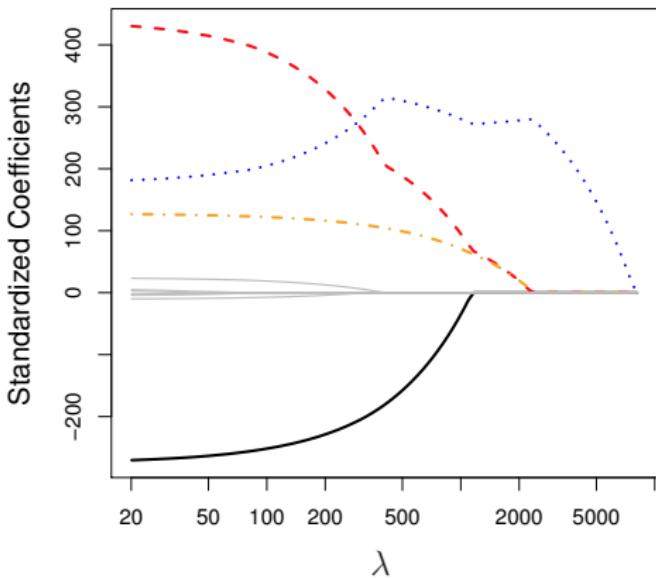
Ridge regression uses a sum of squared weights as the penalty term in its cost function:

$$\frac{1}{N} \sum_{i=1}^N \left(y_i - w_0 - \sum_{j=1}^p w_j x_{ij} \right)^2 + \lambda \sum_{j=1}^p w_j^2.$$

LASSO regression (Least Absolute Shrinkage and Selection Operator) uses an absolute value penalty term in its cost function:

$$\frac{1}{N} \sum_{i=1}^N \left(y_i - w_0 - \sum_{j=1}^p w_j x_{ij} \right)^2 + \lambda \sum_{j=1}^p |w_j|.$$

LASSO Regression in Action



As λ **increases**, the overall magnitude of the coefficients **decrease**.

- Coefficients eventually reach zero for λ sufficiently large
- For any particular value of λ , the features with non-zero coefficients are said to be “selected” by LASSO regression

Taken from “An Introduction to Statistical Learning, with applications in R” (Springer, 2013)
with permission from the authors: G. James, D. Witten, T. Hastie and R. Tibshirani.

Comparing Ridge and LASSO Regression

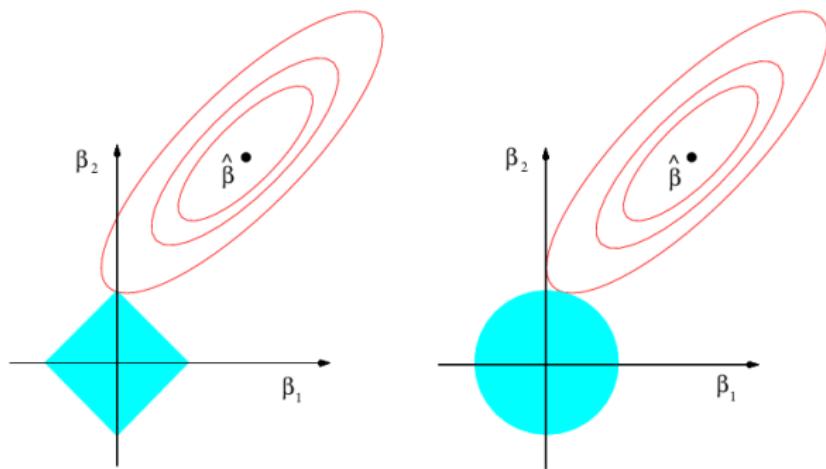
Ridge regression:

- Uses ℓ_2 norm for penalties: $\lambda \sum_{j=1}^p w_j^2$:
- Cost function **remains convex**
- Weights driven towards zero but not eliminated

LASSO regression:

- Uses ℓ_1 norm for penalties: $\lambda \sum_{j=1}^p |w_j|$:
- Cost function is **not convex**
- Will set some weights to zero (favors sparser models)

Why LASSO Regression Performs Attribute Selection



- $\hat{\beta}$ is the multiple linear regression solution (with two attributes)
- Each red contour represents a set of solutions with the same loss
- Blue portion is feasible region for LASSO (left) and ridge (right)
- Intersection of contours and feasible region is more likely to occur at corners with LASSO

Taken from "An Introduction to Statistical Learning, with applications in R" (Springer, 2013)
with permission from the authors: G. James, D. Witten, T. Hastie and R. Tibshirani.

Lecture 03-4c: Feature Engineering

Feature Engineering

Definition

Feature engineering is the process of transforming raw data into a data set that is suitable for machine learning methods.

Most machine learning methods expect data in the form $\{(\mathbf{x}_i, y_i)\}_{i=1}^N$, but real-world data often **is not** given to you this way!

- Data cleaning and formatting is a big part of applied ML & data science
- Attribute selection can often **make** or **break** your model
- For some ML methods, numerical scales need to be **comparable across attributes**, both for interpretation and numerical stability

Why Attribute Scales Matter

Example: Consider predicting house value from square footage and number of bedrooms.

Some typical attribute ranges:

- House value: \$100,000 – \$1,000,000
- Square footage: 500 – 5000
- Number of bedrooms: 1 – 10

In a linear regression model, the weights for the square footage and number of bedrooms attributes will:

- need to be **large** to produce estimates for the house value
- likely **differ** by one to two orders of magnitude
(this significantly impacts both ridge and LASSO regression)

Normalization

Definition

Normalization is the process of converting an actual range of values for a numerical attribute into a standard range of values, typically $[-1, 1]$ or $[0, 1]$.

For attribute j , let $L = \min_{i=1}^N x_{ij}$ and $U = \max_{i=1}^N x_{ij}$.

Then **min-max normalization (rescaling)** of attribute j transforms each attribute value x_{ij} into $x'_{ij} \in [0, 1]$ using:

$$x'_{ij} = \frac{x_{ij} - L}{U - L}.$$

To get $x'_{ij} \in [a, b]$ for any $a, b \in \mathbb{R}$, use $x'_{ij} = a + \frac{(x_{ij} - L)(b - a)}{U - L}$.

Standardization: Z-Score Normalization

Definition

Standardization (Z-score normalization) is the process of transforming attribute values to follow a standard normal distribution.

For attribute j , let $\hat{\mu}_j = \frac{1}{N} \sum_{i=1}^N x_{ij}$ and $s_j = \sqrt{\frac{1}{N-1} \sum_{i=1}^N (x_{ij} - \hat{\mu}_j)^2}$.

Then attribute j can be standardized by transforming each attribute value x_{ij} into x'_{ij} using

$$x'_{ij} = \frac{x_{ij} - \hat{\mu}_j}{s_j}.$$

A Z-scored attribute satisfies:

$$\frac{1}{N} \sum_{i=1}^N x'_{ij} = 0 \quad \text{and} \quad \frac{1}{N-1} \sum_{i=1}^N (x'_{ij} - \bar{x}'_j)^2 = 1.$$

Benefits of Feature Scaling

Benefits of feature scaling:

- All features are roughly on the same scale
- Magnitude of regression weights indicates relative importance
- Numerical calculations (e.g., in gradient descent) work better

In most cases, normalization is preferred over standardization.

Standardization is preferred when

- doing unsupervised learning;
- attribute values are close to normally distributed already;
- there are large outlier values for an attribute.

For attribute values that follow a power-law distribution with **very large outliers** (e.g., income, node degree in a network), using a sublinear transformation (e.g., log or square root) may be beneficial.

Some cool visualizations of feature scaling can be found [here](#).

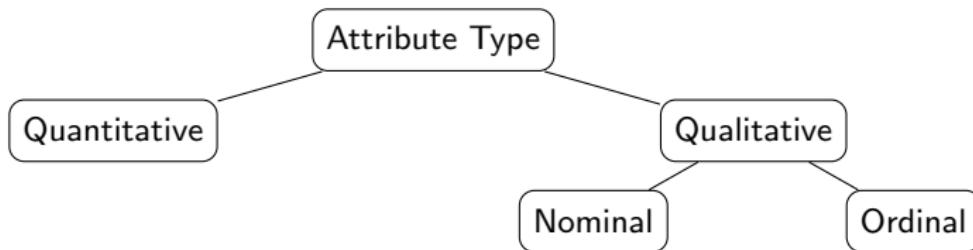
Lecture 03-4d: One-Hot Encoding

Linear Regression with Other Attribute Types

Consider linear regression with a hypothesis function $h(x) = w_0 + w_1x$:

- Changing x by δ corresponds to a change of δw_1 in the output.

This makes sense when attribute X is quantitative (numeric), but what about other attribute types?



- Nominal: e.g., beer type (stout, lager, ale, ...)
- Ordinal: e.g., highest degree obtained (None, HS, BS, MS, PhD)

Using Qualitative Attributes in Regression

With $h(x) = w_0 + w_1 w$, if the attribute X is ordinal (e.g., highest degree obtained), then the categories are **ordered** and we can map them to increasing numeric values, e.g.,

$$\text{None} = 0, \text{ HS} = 1, \text{ BS} = 2, \text{ MS} = 3, \text{ PhD} = 4$$

This mapping assumes the **change** from one category to the next is **constant**. If the output represents income, then each successive degree earned increases your income by w_1 . This is **not realistic!** (E.g., if $w_0 = 0$, then PhDs would earn 4 times the salary of someone with a high school degree!)

If the input attribute X is nominal (e.g., beer type), then there is **no ordering** to the categories, and numeric mappings don't make sense at all, e.g.,

$$\text{Stout} = 1, \text{ Lager} = 2, \text{ Ale} = 3, \dots$$

Does lager have twice the impact of stout? This **doesn't make sense!**

One-Hot Encoding

Definition

One-hot encoding transforms a qualitative attribute with k distinct categories into k separate indicator attributes, one for each category.

Example: If X_1 is highest degree obtained, with categories None, HS, BS, MS, PhD, then one-hot encoding adds the following features:

New derived attributes:

- $X_2 = (X_1 == \text{None})$
- $X_3 = (X_1 == \text{HS})$
- $X_4 = (X_1 == \text{BS})$
- $X_5 = (X_1 == \text{MS})$
- $X_6 = (X_1 == \text{PhD})$

	X_1						
alice	None						
bob	BS						
carol	PhD						
eve	MS						

⇒

	X_1	X_2	X_3	X_4	X_5	X_6
alice	None	1	0	0	0	0
bob	BS	0	0	1	0	0
carol	PhD	0	0	0	0	1
eve	MS	0	0	0	1	0

We need some additional work to handle qualitative **outputs** in regression, though!

CS 457/557: Machine Learning

Lecture 04-1: Model Selection

Lecture 04-1a: Evaluating Performance

Recap:

Supervised Learning Process

Supervised learning process:

- ① Start with a labeled data set $\{(\mathbf{x}_i, y_i)\}_{i=1}^N$
- ② Choose a **hypothesis space** \mathcal{H}
- ③ Choose a learning algorithm:
 - Loss function (e.g., squared loss)
 - Cost function (e.g., average loss)
 - Fitting method

With linear regression, we have already seen that there are **many options** for just the hypothesis space!

So we make our choices and learn a hypothesis function $h_{\mathbf{w}}$.

How do we know we made the **right** choices?

One possible way to answer this is by examining how well the learned hypothesis function $h_{\mathbf{w}}$ does at prediction!

Training Error

Definition

For a regression problem, the **training error** (training loss, training MSE (mean squared error)) of a hypothesis function $h_{\mathbf{w}}$ is

$$\text{Training error } (h_{\mathbf{w}}) = \frac{1}{N} \sum_{i=1}^N (y_i - h_{\mathbf{w}}(\mathbf{x}_i))^2.$$

Two issues with using training error to measure prediction quality:

- ① During learning, the machine **sees** the data on which it gets evaluated.
 - Evaluating a person using identical questions to ones that they studied beforehand doesn't provide much insight into what they really know!
- ② In general, simple hypothesis functions have **greater training error** than complex hypothesis functions!

Training Error in Linear Regression

For linear regression, the average loss cost function

$$\text{Cost}(h_{\mathbf{w}}) = \frac{1}{N} \sum_{i=1}^N (y_i - h_{\mathbf{w}}(\mathbf{x}_i))^2$$

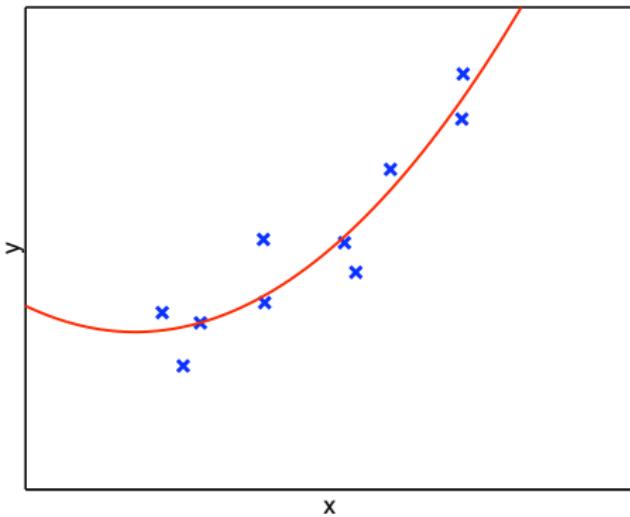
equals the training error. The optimal weight vector \mathbf{w}^* leads to a hypothesis function with the **smallest possible training error**.

Additionally, we can **never do worse** on training error by expanding the hypothesis space! For example, in single variable linear regression:

$$\min_{w_0, w_1} \frac{1}{N} \sum_{i=1}^N (y_i - w_0 - w_1 x_i)^2 \geq \min_{w_0, w_1, w_2} \frac{1}{N} \sum_{i=1}^N (y_i - w_0 - w_1 x_i - w_2 x_i^2)^2.$$

So the training error for a linear hypothesis function is **always at least as large** as the training error for a quadratic hypothesis function!

An Order-2 Solution

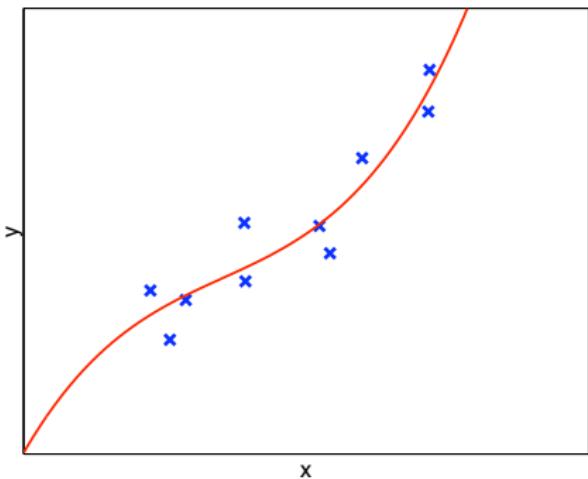


- With an order-2 function, we can fit our data somewhat better than with the original version

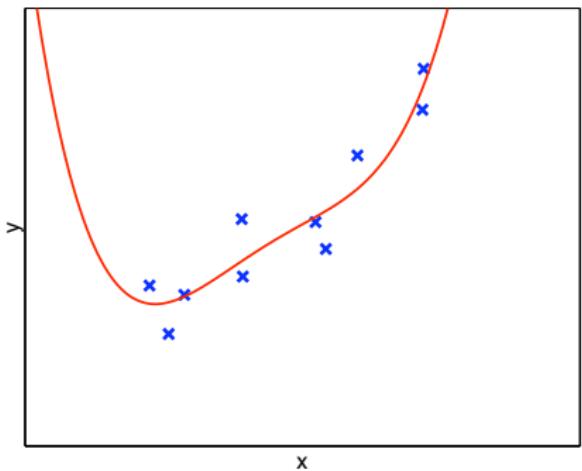
$$h(x) \leftarrow y = 0.73 + 1.74x + 0.68x^2$$

Higher-Order Fitting

Order-3 Solution

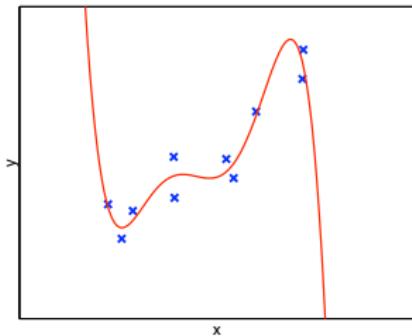


Order-4 Solution

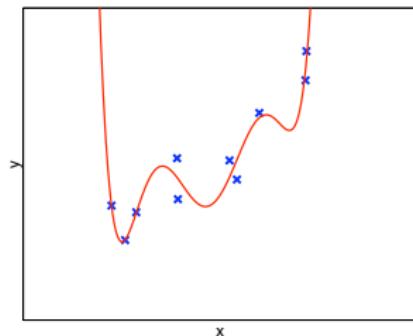


Even Higher-Order Fitting

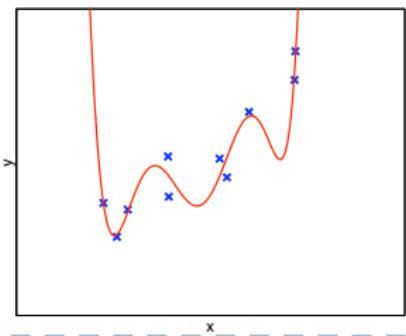
Order-5 Solution



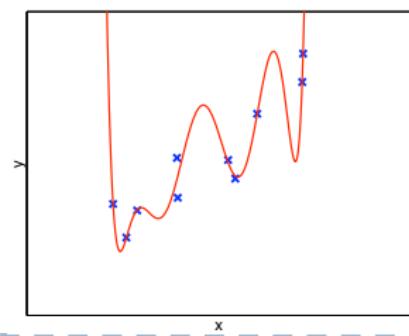
Order-6 Solution



Order-7 Solution

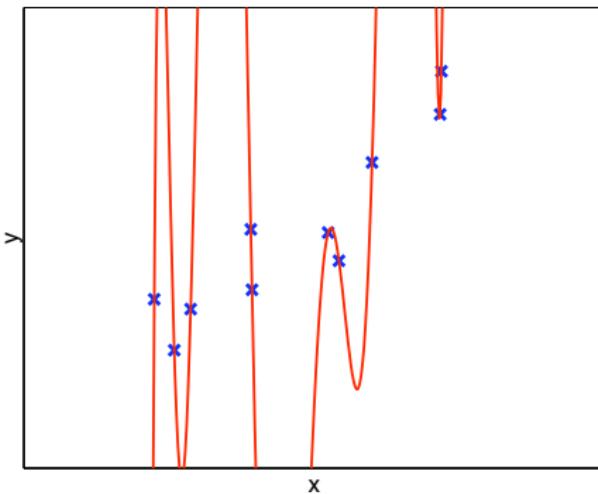


Order-8 Solution



The Risk of Overfitting

- ▶ An order-9 solution hits all the data points exactly, but is very “wild” at points that are not given in the data, with high variance
- ▶ This is a general problem for learning: if we **over-train**, we can end up with a function that is very precise on the data we already have, but will not predict accurately when used on new examples



Lecture 04-1b: Test Error

A Better Way to Measure Prediction Quality: Test Error

A better way to evaluate a hypothesis function h_w is using **new** data!

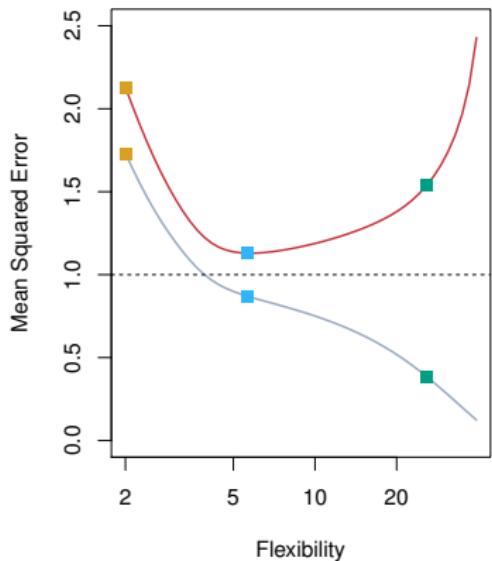
For simplicity, let \mathcal{P} be the set of all possible **new** data points (\mathbf{x}', y') . Then a hypothesis function's **test error** (generalization error, test MSE) is approximately

$$\text{Test error}(h_w) \approx \frac{1}{|\mathcal{P}|} \sum_{(\mathbf{x}', y') \in \mathcal{P}} (y' - h_w(\mathbf{x}'))^2.$$

I.e., the test error is the average of the squared prediction errors across all **out-of-sample** data points. Test error provides an indication of how well h_w **generalizes**.

In general, we have $\text{Training error}(h_w) \leq \text{Test error}(h_w)$, and sometimes this difference is significant.

Relationship Between Training and Test Error



- Gray curve is training error
- Red curve is test error
- Flexibility is a rough measure of the size of the hypothesis space

Training error **decreases** as flexibility **increases**.

Test error initially **decreases** as flexibility increases, but then **increases**.

Taken from "An Introduction to Statistical Learning, with applications in R" (Springer, 2013) with permission from the authors: G. James, D. Witten, T. Hastie and R. Tibshirani.

Insights from Training and Test Errors

What the training and test errors tell us about a hypothesis function:

Training error	Test error	Conclusion
High	High	Underfitting
High	Low	(should not happen)
Low	High	Overfitting
Low	Low	Good!

Definition

A hypothesis **underfits** when it fails to find a pattern in the data or does not represent the pattern well.

Definition

A hypothesis **overfits** when it follows the noise in the data instead of the data's overall pattern.

Underfitting and Overfitting



Lecture 04-1c: Test Error in Theory

Understanding Test Error

To understand the underlying components of test error, we need to shift to a **probabilistic perspective**!

Instead of treating the training set $\{(\mathbf{x}_i, y_i)\}_{i=1}^N$ as **fixed**, we treat the training set as being a **random sample**.

(That is, the training set could have been different.)

Let \mathcal{S} be the set of all possible training sets $\{(\mathbf{x}_i, y_i)\}_{i=1}^N$.

- The weights \mathbf{w} for $h_{\mathbf{w}}$ are **learned from a particular** training set $S \in \mathcal{S}$. To denote this dependence, we write $h_{\mathbf{w}_S}$.
- Changing the training set **changes** the learned weights!
(and hence changes the hypothesis function)

We want to see how a particular learning method (i.e., a process for constructing a hypothesis function) fares across different training sets.

Expected Test MSE

For a particular learning method, the **expected test MSE** for an arbitrary test point (\mathbf{x}_0, y_0) is $E \left[(y_0 - h_{\mathbf{w}_S}(\mathbf{x}_0))^2 \right]$, where the expectation is taken across *all possible training sets*. Intuitively,

$$E \left[(y_0 - h_{\mathbf{w}_S}(\mathbf{x}_0))^2 \right] \approx \frac{1}{|\mathcal{S}|} \sum_{S \in \mathcal{S}} (y_0 - h_{\mathbf{w}_S}(\mathbf{x}_0))^2.$$

With some work, we can **decompose** the expected test MSE into three core components:

$$E \left[(y_0 - h_{\mathbf{w}_S}(\mathbf{x}_0))^2 \right] = [\text{Bias}(h_{\mathbf{w}_S}(\mathbf{x}_0))]^2 + \text{Var}(h_{\mathbf{w}_S}(\mathbf{x}_0)) + \text{Var}(\epsilon)$$

where ϵ is the noise term.

Bias and Variance

We have

$$E \left[(y_0 - h_{\mathbf{w}_S}(\mathbf{x}_0))^2 \right] = [\text{Bias}(h_{\mathbf{w}_S}(\mathbf{x}_0))]^2 + \text{Var}(h_{\mathbf{w}_S}(\mathbf{x}_0)) + \text{Var}(\epsilon).$$

- (Squared) Model Bias: $[\text{Bias}(h_{\mathbf{w}_S}(\mathbf{x}_0))]^2$
 - Low if $h_{\mathbf{w}_S}$ matches f closely; high otherwise
- Model Variance: $\text{Var}(h_{\mathbf{w}_S}(\mathbf{x}_0))$
 - Low if learned weights \mathbf{w}_S don't change much across training sets
 - High if learned weights \mathbf{w}_S fluctuate across training sets
- Irreducible Error: $\text{Var}(\epsilon)$
 - Noise in the data generation and collection process

Conceptual Definitions

Definition

A learning method's **bias** is the error that is introduced by erroneous assumptions in the form of the hypothesis function.

A learning method with high bias tends to **underfit** the training data.

- Example: Regression with linear h_w when f is nonlinear
- Fix: Expand hypothesis space

Definition

A learning method's **variance** is a measure of how much the learned hypothesis function changes based on the training set used for learning.

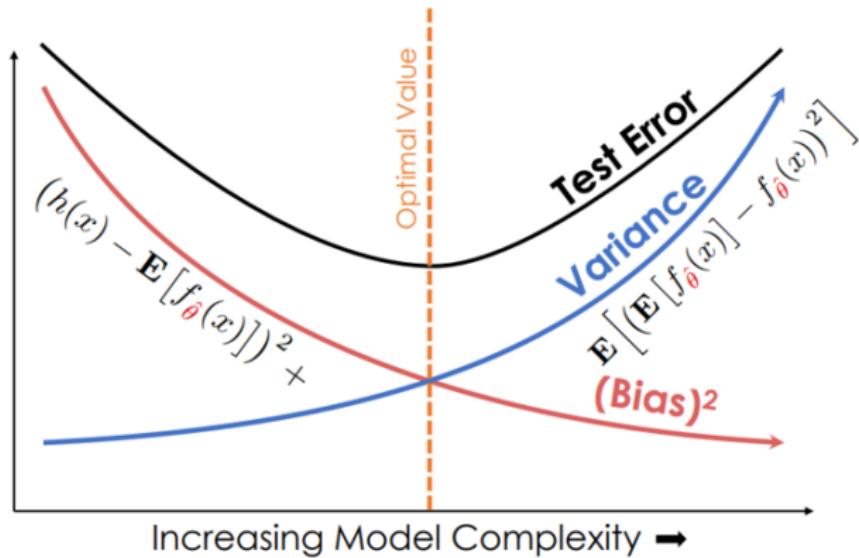
A learning method with high variance tends to **overfit** the training data.

- Example: Polynomial regression with high degree d
- Fixes: Lower max degree d ; add regularization; add training data!

Bias-Variance Tradeoff

Ideally, we want a learning method with **low bias and low variance**.

Unfortunately, there is often a tradeoff between these two!



Lecture 04-1d: Test Error in Practice

Test Error in Practice

Previously, we defined test error as

$$\text{Test error } (h_{\mathbf{w}}) \approx \frac{1}{|\mathcal{P}|} \sum_{(\mathbf{x}', y') \in \mathcal{P}} (y' - h_{\mathbf{w}}(\mathbf{x}'))^2,$$

where \mathcal{P} is the set of all possible **new** data points.

However, we can't use this to compute a hypothesis function's test error **without** knowing the true distribution of data points in \mathcal{P} .

Instead, we **attempt to estimate** a hypothesis function's test error by "creating" some **out-of-sample** data points from the data points that we already have.

Train-Test Split

Estimating the test error is often done as follows:

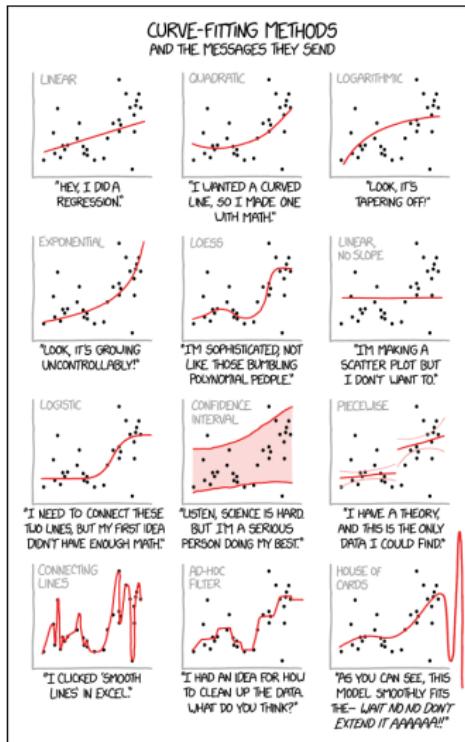
- ① Divide the sample $\{(\mathbf{x}_i, y_i)\}_{i=1}^N$ into two groups: a training set and a test set
- ② Learn a hypothesis function $h_{\mathbf{w}}$ from the training set
- ③ Estimate $h_{\mathbf{w}}$'s test error using the test set:

$$\widehat{\text{Test error}}(h_{\mathbf{w}}) = \frac{1}{|\text{Test}|} \sum_{i \in \text{Test}} (y_i - h_{\mathbf{w}}(\mathbf{x}_i))^2$$

Usually an 80/20 random split is used for training and testing data.

We will revisit this idea to see how it can be used to help us select a good hypothesis function in the next lecture!

Comic of the Day



<http://xkcd.com/2048/>

CS 457/557: Machine Learning

Lecture 04-2: Model Selection & Cross-Validation

Lecture 04-2a: Model Selection

Quick Recap and Terminology

Supervised learning process:

- ① Start with a labeled data set $\{(\mathbf{x}_i, y_i)\}_{i=1}^N$
- ② Choose a **hypothesis space** \mathcal{H}
- ③ Choose a learning algorithm:
 - Loss function (e.g., squared loss)
 - Cost function (e.g., average loss)
 - Fitting method

A set of choices gives us a particular **learning method**, or **model**, M .

When given data, the learning method **learns** a particular hypothesis function h from its hypothesis space. Learning is also called **fitting the model**.

We want to differentiate between the learning method itself and the learned hypothesis function. The terminology isn't always clear (e.g., sometimes "model" refers to the hypothesis space, sometimes it refers to the learned hypothesis function), but usually you can infer from context. We'll use "model" to refer to the learning method itself.

Test Error

We ultimately assess a particular learning method's performance through the **out-of-sample** performance, or **test error**, of its learned hypothesis function h :

$$\text{Test error}(h) \approx \frac{1}{|\mathcal{P}|} \sum_{(\mathbf{x}', y') \in \mathcal{P}} (y' - h(\mathbf{x}'))^2.$$

As \mathcal{P} , the set of all out-of-sample data points, is typically **unknown**, we **estimate** test error:

- ① Divide $\{(\mathbf{x}_i, y_i)\}_{i=1}^N$ into two groups: a training set and a test set
- ② Learn a hypothesis function h from the training set
- ③ Estimate test error of h using the test set:

$$\widehat{\text{Test error}}(h) = \frac{1}{|\text{Test}|} \sum_{i \in \text{Test}} (y_i - h(\mathbf{x}_i))^2$$

How can we use this to select the best model from a set of options?

A First Attempt at Model Selection

Suppose we are trying to decide between three different learning methods (models), $M^{(1)}$, $M^{(2)}$, and $M^{(3)}$.

Simple Model Selection Process: First Attempt

Split the sample $\{(\mathbf{x}_i, y_i)\}_{i=1}^N$ into a **training set** and a **test set**

for each $k \in \{1, 2, 3\}$ **do**

 Use $M^{(k)}$ to learn $h^{(k)}$ from training set

 Estimate test error for $h^{(k)}$ using

$$\widehat{\text{Test error}}\left(h^{(k)}\right) = \frac{1}{|\text{Test}|} \sum_{i \in \text{Test}} \left(y_i - h^{(k)}(\mathbf{x}_i)\right)^2$$

$$k^* \leftarrow \arg \min_k \widehat{\text{Test error}}\left(h^{(k)}\right)$$

Set final model $M^* \leftarrow M^{(k^*)}$

To be **statistically rigorous**, we need to do a bit more work...

Evaluating the Model Selection Process

Using the proposed selection process yields a final model M^* .

Natural question: Is M^* with learned hypothesis function h^* a **good** model?

- If h^* has good out-of-sample performance, then **yes**
- If h^* has bad out-of-sample performance, then **no**

What is h^* 's out-of-sample performance?

First guess:

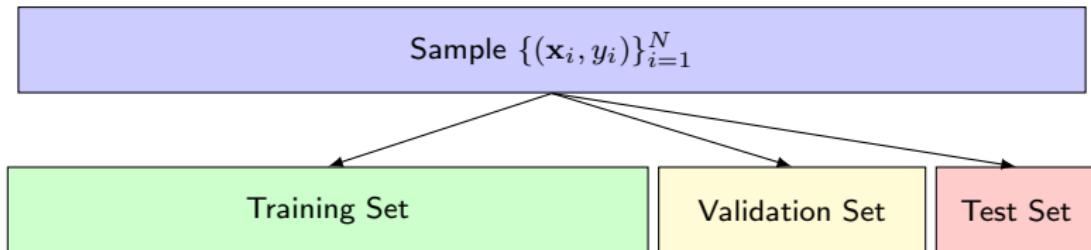
$$\widehat{\text{Test error}}(h^*) = \widehat{\text{Test error}}\left(h^{(k^*)}\right)$$

where k^* indicates the index of the best hypothesis.

However, this estimate of h^* 's test error is **biased** because we **looked at** the test data itself to choose h^* .

Training, Validation, and Test Sets

To get a **correct** estimate of the test error of the selected hypothesis, we need to divide the sample into **three** subsets:



- **Training Set**: used to learn each individual hypothesis
- **Validation Set (Hold-out Set)**: used during development to refine and compare learning methods (models) and select the best
- **Test Set**: used at the **end** to evaluate the final selected model

Avoids issues of “peeking” at the test set during model selection!

Model Selection: The Validation Set Approach

Simple Model Selection Process: Second Attempt

Split the sample $\{(\mathbf{x}_i, y_i)\}_{i=1}^N$ into a **training set**, a **validation set**, and a **test set**
for each $k \in \{1, 2, 3\}$ **do**

 Use $M^{(k)}$ to learn $h^{(k)}$ from training set

 Estimate **validation error** for $h^{(k)}$ using

$$\text{Validation error}\left(h^{(k)}\right) = \frac{1}{|\text{Validation}|} \sum_{i \in \text{Validation}} \left(y_i - h^{(k)}(\mathbf{x}_i)\right)^2$$

$$k^* \leftarrow \arg \min_k \text{Validation error}\left(h^{(k)}\right)$$

Set final model $M^* \leftarrow M^{(k^*)}$

Optional: Use M^* to learn h^* on **combined** training and validation set

Estimate test error of h^* as

$$\widehat{\text{Test error}}(h^*) = \frac{1}{|\text{Test}|} \sum_{i \in \text{Test}} (y_i - h^*(\mathbf{x}_i))^2$$

Lecture 04-2b: Model Selection Application

Model Selection Application

The validation set approach can be used for selecting **model hyperparameters**.

Motivating example: polynomial regression with a single input variable involves the model hyperparameter d , which determines the maximum degree polynomial within the hypothesis space.

$$\mathcal{H}^{(d)} = \left\{ h : \mathbb{R} \rightarrow \mathbb{R} \mid h(x) = w_0 + w_1x + w_2x^2 + \dots + w_dx^d \right\}$$

For any choice of $d \in \mathbb{Z}^+$, we get learning method $M^{(d)}$ which learns a hypothesis $h^{(d)} \in \mathcal{H}^{(d)}$.

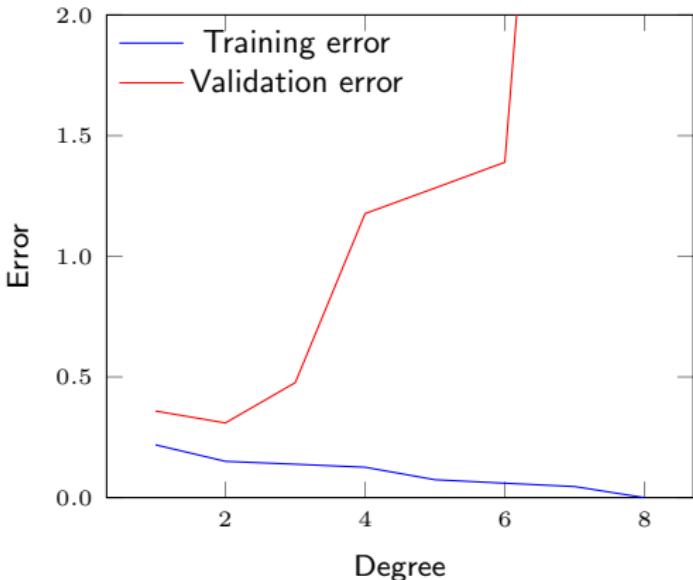
- If d is too small, we **underfit** (high model bias)
- If d is too big, we **overfit** (high model variance)

How do we determine the **optimal** choice for d ?

Example

Degree	Training error	Validation error
1	0.2188	0.3558
2	0.1504	0.3095
3	0.1384	0.4764
4	0.1259	1.1770
5	0.0742	1.2828
6	0.0598	1.3896
7	0.0458	38.819
8	0.0000	6097.5

- Optimal degree is 2 here
- $d > 2$ leads to progressively worse **overfitting**



What's in a Name?

Unfortunately, the names for these different subsets of data are not always used consistently:

AI:MA Textbook	Skiena Textbook	Cady Textbook
Training Set	Training Data	Training Set
Validation Set	Testing Data	Testing Set
Test Set	Evaluation Data	Validation Set

We'll use the naming scheme from the first column here.

"This is the most blatant example of the terminological confusion that pervades artificial intelligence research."

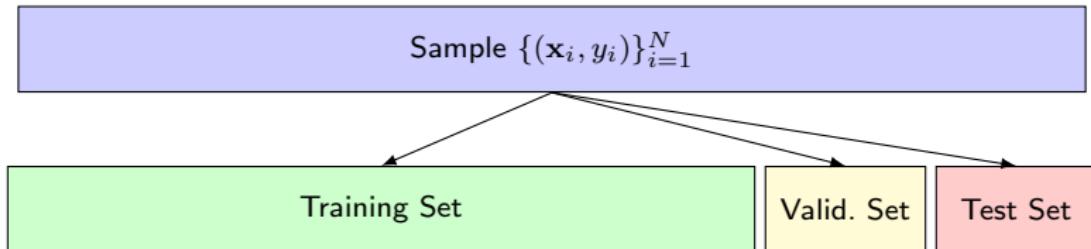
— B. Ripley, Pattern recognition and neural networks

https://en.wikipedia.org/wiki/Training,_validation,_and_test_sets

Lecture 04-2c: Leave-One-Out Cross-Validation

Getting More from the Sample

The recommended train-validation-test split is around 70/15/15:



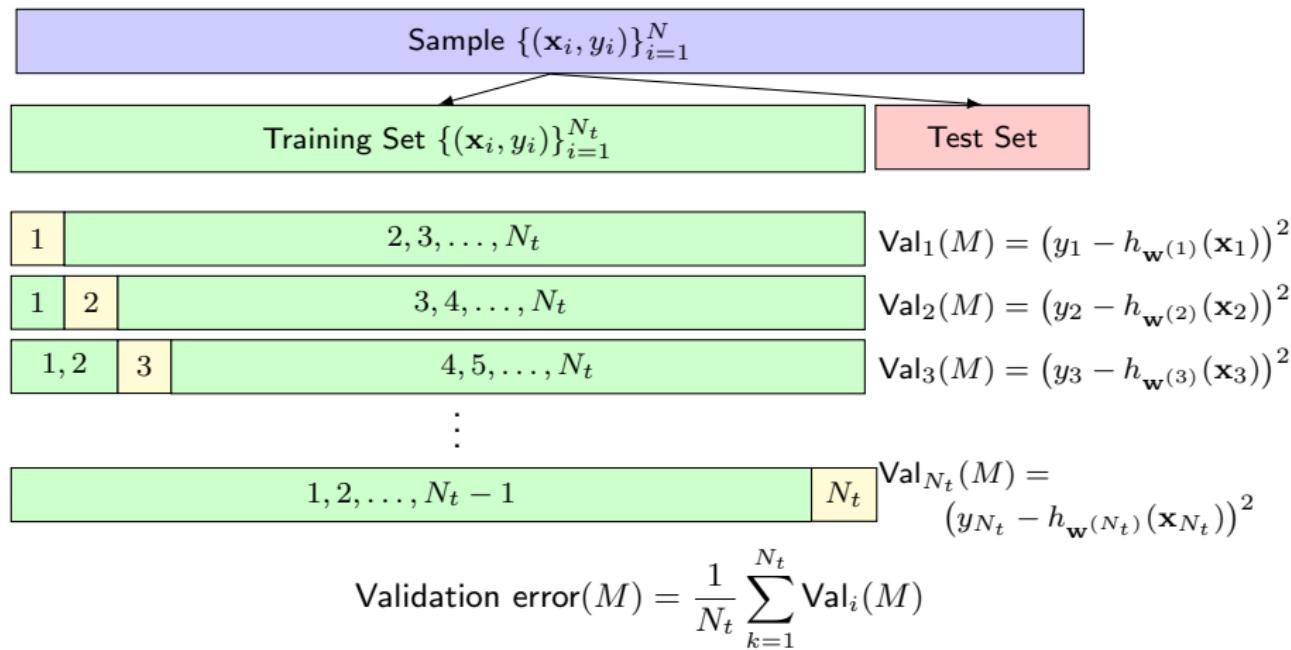
Validation error can be **variable** depending on the random split

- Increasing size of validation set decreases variability
- Decreasing size of training set increases model variability

Cross-validation refines the validation set approach to mitigate these issues by using parts of the training set for validation!

Leave-One-Out Cross-Validation

Leave-One-Out Cross-Validation (LOOCV) estimates the validation error for a learning method (model) M as:



Pseudocode for LOOCV

For a learning method (model) M :

Leave-One-Out Cross-Validation

procedure LOOCV(Model M ; Training set)

for each $i \in \text{Train}$ **do**

 Use M to learn $h^{(i)}$ on $\text{Train} - \{i\}$

$$\text{Val}_i(M) = (y_i - h^{(i)}(\mathbf{x}_i))^2$$

$$\text{Validation error}(M) = \frac{1}{|\text{Train}|} \sum_{i \in \text{Train}} \text{Val}_i(M)$$

return Validation error(M)

- The learned hypothesis function $h^{(i)}$ **changes** from one iteration to the next, based on which data point is left out of the training set
- Above implementation is computationally **prohibitive**

An Example of LOOCV

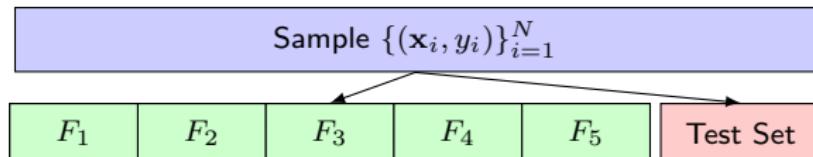
For a data set of 10 input-output pairs, LOOCV estimates the validation error of learning method (model) M as follows:

Iteration	Training set	Validation Set	Training error	Validation error
1	$\{(\mathbf{x}_i, y_i)\}_{i=1}^{10} - \{(\mathbf{x}_1, y_1)\}$	$\{(\mathbf{x}_1, y_1)\}$	0.4928	0.0044
2	$\{(\mathbf{x}_i, y_i)\}_{i=1}^{10} - \{(\mathbf{x}_2, y_2)\}$	$\{(\mathbf{x}_2, y_2)\}$	0.1995	0.1869
3	$\{(\mathbf{x}_i, y_i)\}_{i=1}^{10} - \{(\mathbf{x}_3, y_3)\}$	$\{(\mathbf{x}_3, y_3)\}$	0.3461	0.0053
4	$\{(\mathbf{x}_i, y_i)\}_{i=1}^{10} - \{(\mathbf{x}_4, y_4)\}$	$\{(\mathbf{x}_4, y_4)\}$	0.3887	0.8681
5	$\{(\mathbf{x}_i, y_i)\}_{i=1}^{10} - \{(\mathbf{x}_5, y_5)\}$	$\{(\mathbf{x}_5, y_5)\}$	0.2128	0.3439
6	$\{(\mathbf{x}_i, y_i)\}_{i=1}^{10} - \{(\mathbf{x}_6, y_6)\}$	$\{(\mathbf{x}_6, y_6)\}$	0.1996	0.1567
7	$\{(\mathbf{x}_i, y_i)\}_{i=1}^{10} - \{(\mathbf{x}_7, y_7)\}$	$\{(\mathbf{x}_7, y_7)\}$	0.5707	0.7205
8	$\{(\mathbf{x}_i, y_i)\}_{i=1}^{10} - \{(\mathbf{x}_8, y_8)\}$	$\{(\mathbf{x}_8, y_8)\}$	0.2661	0.0203
9	$\{(\mathbf{x}_i, y_i)\}_{i=1}^{10} - \{(\mathbf{x}_9, y_9)\}$	$\{(\mathbf{x}_9, y_9)\}$	0.3604	0.2033
10	$\{(\mathbf{x}_i, y_i)\}_{i=1}^{10} - \{(\mathbf{x}_{10}, y_{10})\}$	$\{(\mathbf{x}_{10}, y_{10})\}$	0.2138	1.0490
mean:			0.2188	0.3558

Lecture 04-2d: *k*-Fold Cross-Validation

k-Fold Cross-Validation

***k*-Fold Cross-Validation** (*k*-Fold CV) computes the validation error by:



F_1	F_2	F_3	F_4	F_5
F_1	F_2	F_3	F_4	F_5
F_1	F_2	F_3	F_4	F_5
F_1	F_2	F_3	F_4	F_5
F_1	F_2	F_3	F_4	F_5

$$\text{Val}_1(M) = \frac{1}{|F_1|} \sum_{i \in F_1} (y_i - h^{(1)}(\mathbf{x}_i))^2$$
$$\text{Val}_2(M) = \frac{1}{|F_2|} \sum_{i \in F_2} (y_i - h^{(2)}(\mathbf{x}_i))^2$$
$$\text{Val}_3(M) = \frac{1}{|F_3|} \sum_{i \in F_3} (y_i - h^{(3)}(\mathbf{x}_i))^2$$
$$\text{Val}_4(M) = \frac{1}{|F_4|} \sum_{i \in F_4} (y_i - h^{(4)}(\mathbf{x}_i))^2$$
$$\text{Val}_5(M) = \frac{1}{|F_5|} \sum_{i \in F_5} (y_i - h^{(5)}(\mathbf{x}_i))^2$$

$$\text{Validation error}(M) = \frac{1}{k} \sum_{i=1}^k \text{Val}_i(M)$$

Pseudocode for k -Fold CV

k -Fold CV Process

procedure $\text{kFOLDCV}(\text{Model } M; \text{Training set}; k)$

 Split training set into k folds, F_1, F_2, \dots, F_k

for each $j \in \{1, 2, \dots, k\}$ **do**

 Use M to learn $h^{(j)}$ on Train – F_j

$$\text{Val}_j(M) = \frac{1}{|F_j|} \sum_{i \in F_j} (y_i - h^{(j)}(\mathbf{x}_i))^2$$

$$\text{Validation error}(M) = \frac{1}{k} \sum_{j=1}^k \text{Val}_j(M)$$

return Validation error(M)

- Divides training set into k different validation tests, or folds
- On each iteration, all but one fold is used for training and the held-out fold is used for validation
- Average the validation errors from the k folds at the end

Notes on Cross-Validation Techniques

Both LOOCV and k -Fold CV:

- Assume the Train/Test split has been done properly beforehand
- Operate on the training set alone and repurpose it to get validation data
- Return estimates of a learning method's validation error

LOOCV:

- Once the train/test split is done, LOOCV is deterministic
- Computationally expensive in general
 - If model parameters are estimated using least squares, then LOOCV can be computed without repeatedly fitting the model!

k -Fold CV:

- k is typically 5 or 10; $k = |\text{Train}|$ yields LOOCV
- Typically faster computational performance compared to LOOCV
- Each training data point is used for validation once and for training $k - 1$ times

Lecture 04-2e: Cross-Validation Application

Cross-Validation Application: Selecting Model Hyperparameters

Cross-validation is primarily used during model development and refinement: it helps us figure out what is working and what isn't. Specifically, CV helps us tune our **model hyperparameters**. (Recall: model hyperparameters must be set **before** fitting)

Two examples:

- ① Polynomial regression with model hyperparameter d :

$$\mathcal{H}^{(d)} = \left\{ h : \mathbb{R} \rightarrow \mathbb{R} \mid h(x) = w_0 + w_1x + w_2x^2 + \dots + w_dx^d \right\}.$$

- ② Ridge regression with model hyperparameter λ :

$$\text{Cost}(h_{\mathbf{w}}) = \text{Loss}(h_{\mathbf{w}}) + \lambda \sum_{j=1}^p w_j^2$$

Cross-Validation Application: Selecting λ for Ridge Regression

Suppose we have a regression model M with parameters \mathbf{w} and want to use ridge regression with penalty hyperparameter λ .

To find the **right** choice of λ :

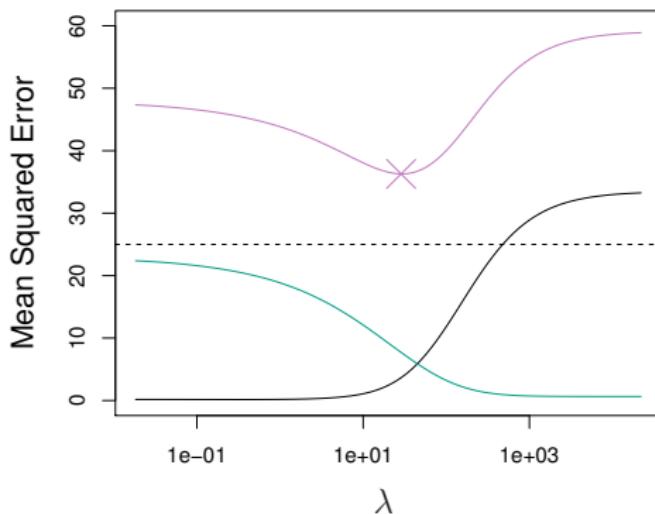
- ① Pick a grid of possible λ values, e.g., $\Lambda = \{0.01, 0.1, 1, 5, 10, 100\}$
- ② Compute the validation error for M with each $\lambda \in \Lambda$
- ③ Use λ with lowest validation error

Ridge Regression Hyperparameter Search

```
procedure SEARCHCV(Model  $M$ ; Training set;  $k$ ;  $\Lambda$ )
  for each  $\lambda \in \Lambda$  do
    Validation error  $(M^{(\lambda)}) = \text{kFOLDCV}(M^{(\lambda)}, \text{Training set}, k)$ 
     $\lambda^* \leftarrow \arg \min_{\lambda} \text{Validation error } (M^{(\lambda)})$ 
  Final model  $M^* \leftarrow M^{(\lambda^*)}$ 
```

Ridge Regression, Bias, Variance, and Test MSE

With ridge regression, the hyperparameter λ helps us balance between **underfitting** and **overfitting**!

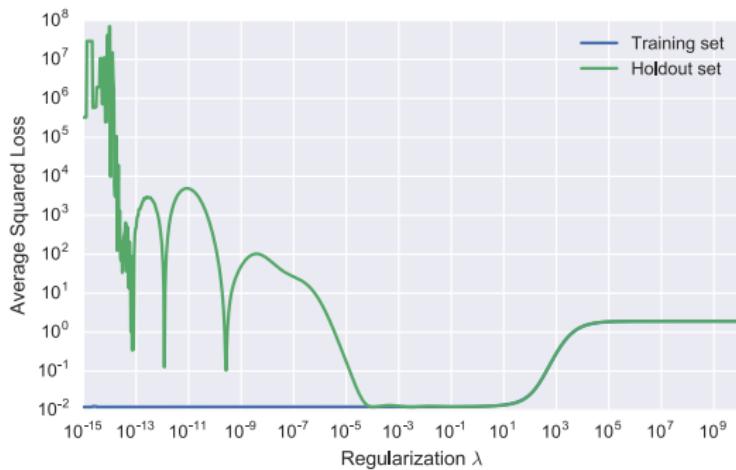


As λ **increases**:

- model bias (black line) **increases**
- model variance (green line) **decreases**
- test MSE (purple line) first decreases and then increases

Taken from "An Introduction to Statistical Learning, with applications in R" (Springer, 2013) with permission from the authors: G. James, D. Witten, T. Hastie and R. Tibshirani.

Training/cross-validation loss by regularization



CS 457/557: Machine Learning

Lecture 05-1: Linear Classification

Quick Recap

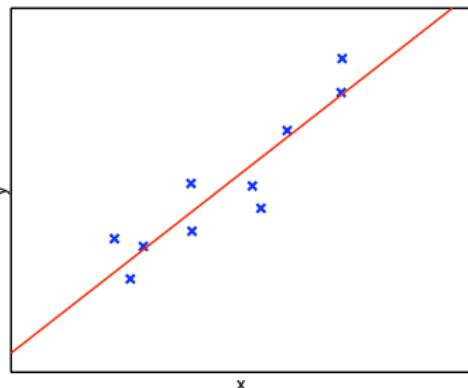
Supervised Learning Problem

Given a *labeled* data set $\{(\mathbf{x}_i, y_i)\}_{i=1}^N$ with $y = f(\mathbf{x})$ for some unknown function f , identify a **hypothesis function** h that approximates f well (i.e., $h(\mathbf{x}) \approx f(\mathbf{x})$ for all \mathbf{x}).

The **type of learning problem** we are solving depends on the type of the output values y :

- If y is continuous-valued, then we have a regression problem.
 - E.g., predicting house value
- If y has values from a finite discrete set, then we have a classification problem.
 - E.g., predicting spam or non-spam
 - Classifying hand-written digits as numbers 1–10

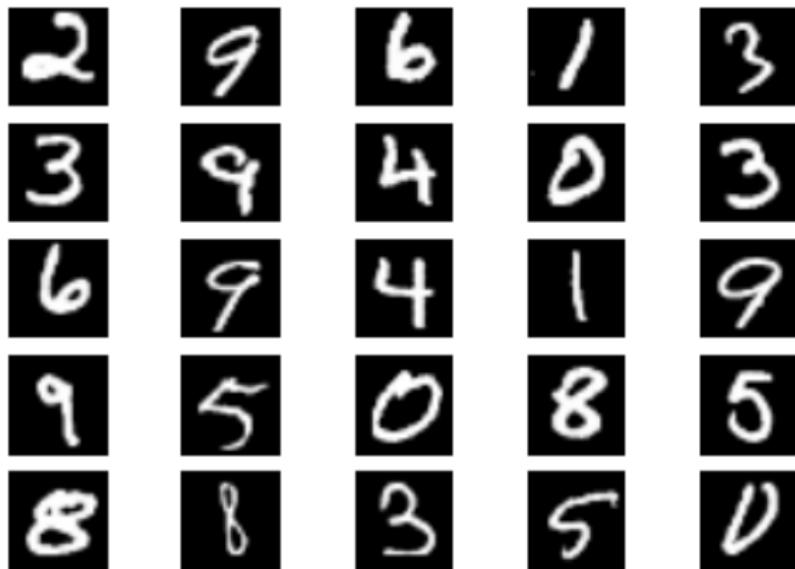
Review: Linear Regression



- ▶ We want to learn a **hypothesis function** h that minimizes our error relative to the actual output function f
- ▶ Often we will assume that this function h is linear, so the problem becomes finding a set of weights that minimize the error between f and our function:

$$h(x_1, x_2, \dots, x_n) = w_0 + w_1 x_1 + w_2 x_2 + \dots + w_n x_n$$

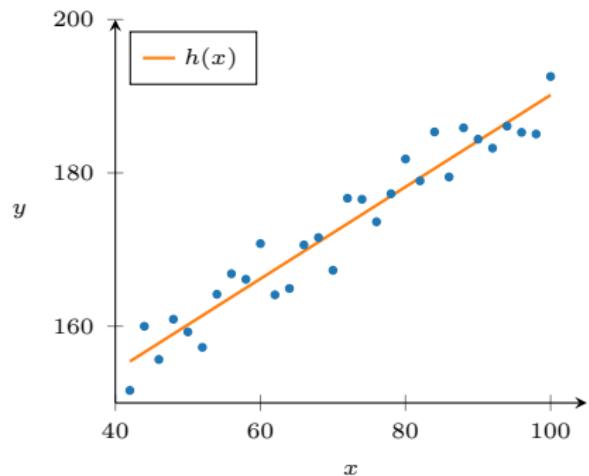
Classification Problems



- ▶ Often, we don't want a real-valued hypothesis function
- ▶ Instead, we want to divide inputs into distinct, discrete types, for example the digits 0–9

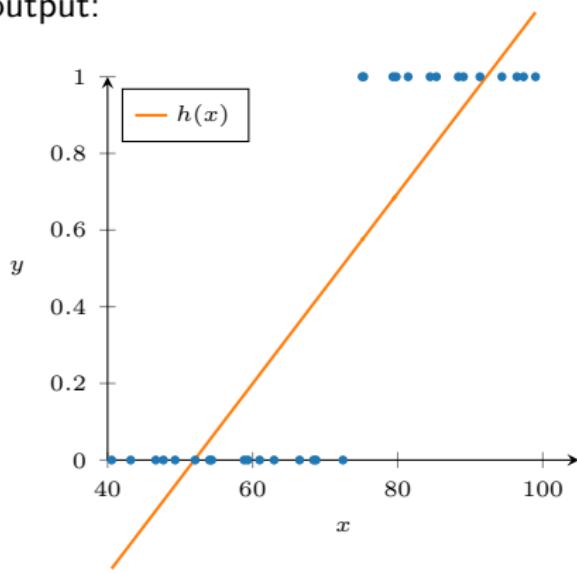
Comparing Regression and Classification

Regression with one input and one output:



We draw a line in input-output space to fit the data.

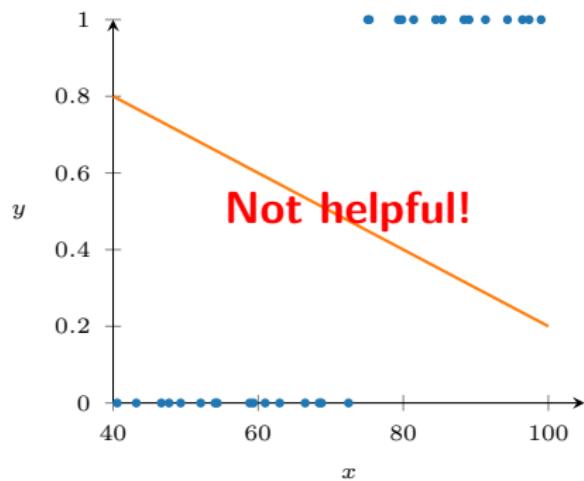
Classification with one input and one output:



Drawing a line to this data doesn't quite work...

From Trend Lines to Separating Lines

Instead of drawing a line to **fit** the data, we want to draw a line to **separate** the data!

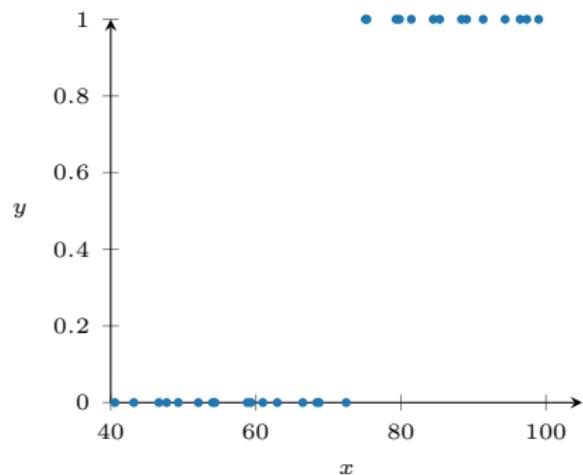


- Drawing a separating line in **input-output** space is **too easy**, and is **useless** for prediction
- Instead, we need to separate the classes in **input** space...

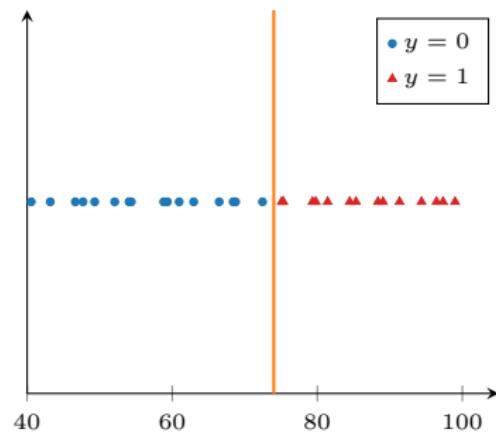
Separating Lines in Input Space

We want to draw a line to **separate** the data in the **input space**.

Input-output space:



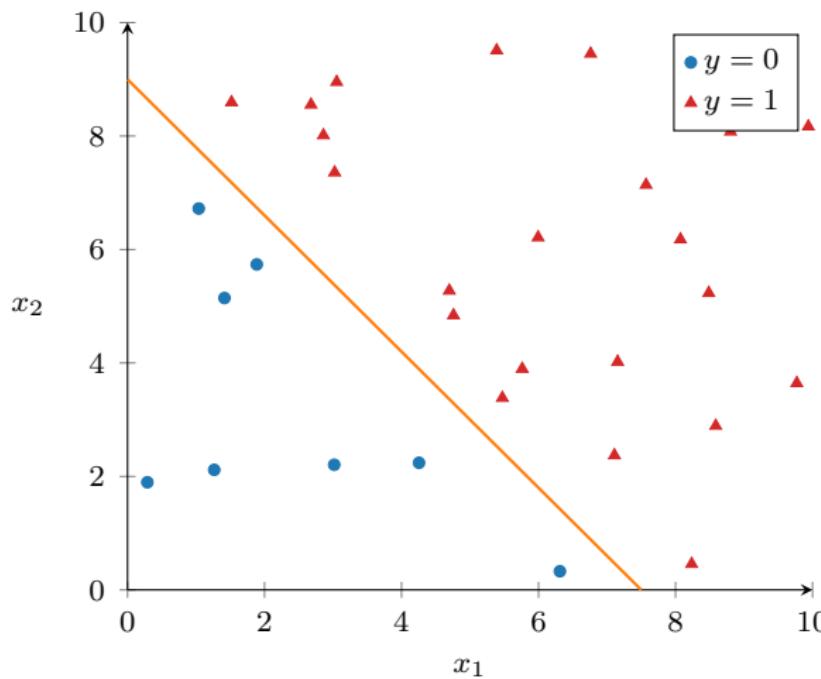
Input space with class markers:



The orange line (point) represents a **linear separator** that splits the data into two classes (usually called class 1 and class 0, or positive/negative).

Separating Lines with Two Input Attributes

We want to draw a line to **separate** the data in the **input space**.



Linear Separators

Notation:

- Input data has p features plus an augmented zeroth feature:

$$\mathbf{x} = (x_0, x_1, x_2, \dots, x_p) \quad \text{with} \quad x_0 = 1$$

- Any linear function of \mathbf{x} is defined by $p + 1$ weights,

$$\mathbf{w} = (w_0, w_1, w_2, \dots, w_p)$$

Linear functions can be written as $\mathbf{w} \cdot \mathbf{x} = \sum_{j=0}^p w_j x_j$.

Definition

A **linear separator** (boundary) defined by weight vector \mathbf{w} is the set of all points satisfying $\mathbf{w} \cdot \mathbf{x} = 0$ (equivalently, $w_0 + w_1 x_1 + \dots + w_p x_p = 0$).

Threshold Functions and Linear Separability

We can use a linear separator for classification by combining it with a **threshold function**:

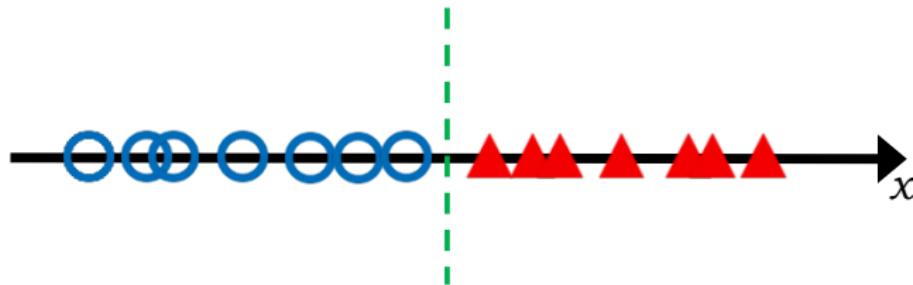
$$h_{\mathbf{w}}(\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{w} \cdot \mathbf{x} \geq 0 \\ 0 & \text{if } \mathbf{w} \cdot \mathbf{x} < 0 \end{cases}$$

(Outputs 1 and 0 are *arbitrary labels* for one of two possible classes.)

Definition

A data set $\{(\mathbf{x}_i, y_i)\}_{i=1}^N$ is **linearly separable** if there exists a weight vector \mathbf{w} such that $h_{\mathbf{w}}(\mathbf{x}_i) = y_i$ for all $i \in \{1, 2, \dots, N\}$.

From Regression to Classification

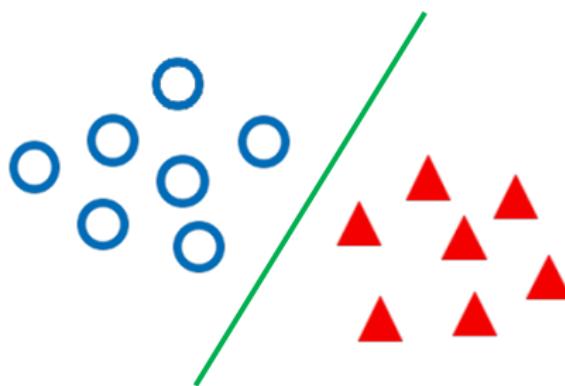


- ▶ Data is **linearly separable** if it can be divided into classes using a linear boundary:

$$\mathbf{w} \cdot \mathbf{x} = 0$$

- ▶ Such a boundary, in 1-dimensional space, is a **threshold value**

From Regression to Classification



- ▶ Data is **linearly separable** if it can be divided into classes using a linear boundary:
$$\mathbf{w} \cdot \mathbf{x} = 0$$
- ▶ Such a boundary, in 2-dimensional space, is a **line**

From Regression to Classification

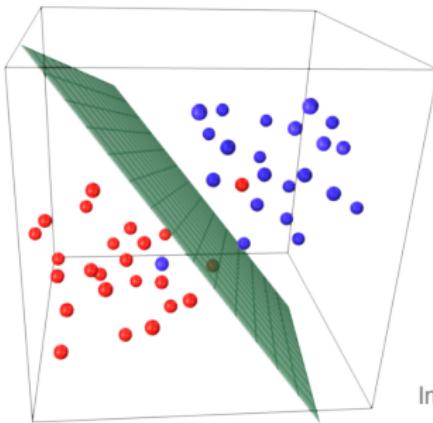


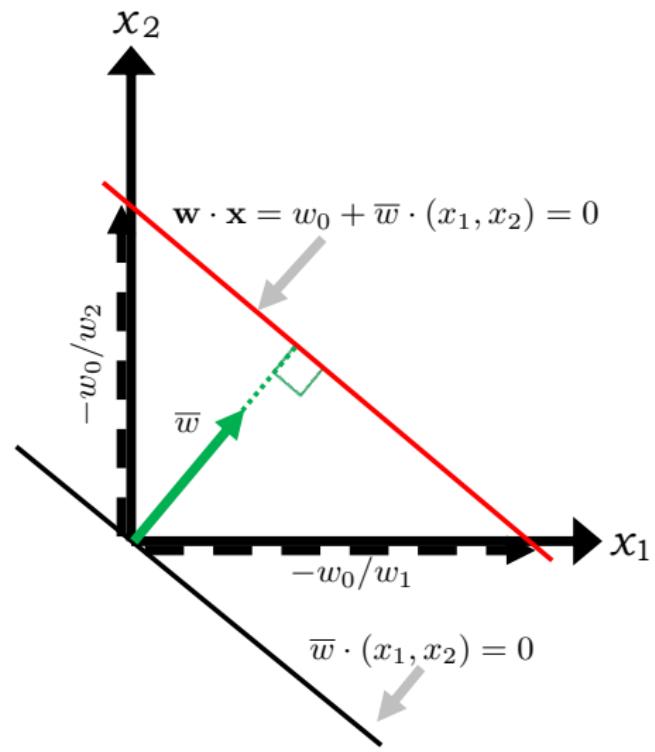
Image: R. Urtasun (U. of Toronto)

- ▶ Data is **linearly separable** if it can be divided into classes using a linear boundary:

$$\mathbf{w} \cdot \mathbf{x} = 0$$

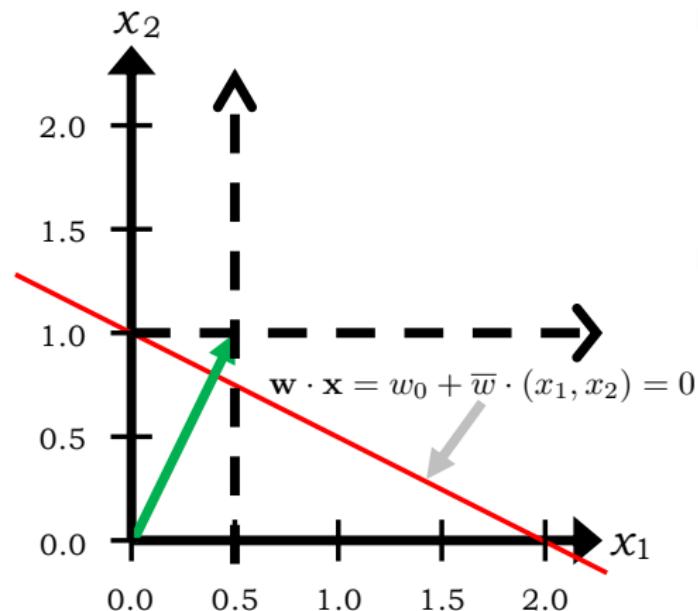
- ▶ Such a boundary, in 3-dimensional space, is a **plane**
- ▶ In higher dimensions, it is a **hyper-plane**

The Geometry of Linear Boundaries



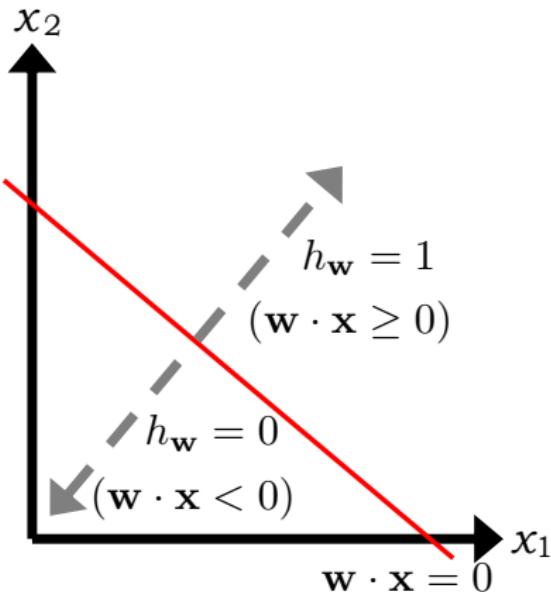
- ▶ Suppose we have 2-dimensional inputs $\mathbf{x} = (x_1, x_2)$
- ▶ The “real” weights $\bar{\mathbf{w}} = (w_1, w_2)$ define a **vector**
- ▶ The boundary where our linear function is zero,
 $\mathbf{w} \cdot \mathbf{x} = w_0 + \bar{\mathbf{w}} \cdot (x_1, x_2) = 0$ is an orthogonal line, parallel to $\bar{\mathbf{w}} \cdot (x_1, x_2) = 0$
- ▶ Its offset from origin is determined by w_0 (which is called the **bias weight**)

The Geometry of Linear Boundaries



- ▶ For example, with “real” weights:
 $\bar{w} = (w_1, w_2) = (0.5, 1.0)$
we get the vector shown
as a green arrow
- ▶ Then, for a bias weight
 $w_0 = -1.0$
the **boundary where our**
linear function is zero,
 $w \cdot x = w_0 + \bar{w} \cdot (x_1, x_2) = 0$
is the line shown in red,
crossing origin at $(2,0)$ & $(0,1)$

The Geometry of Linear Boundaries



- Once we have our linear boundary, data points are classified according to our threshold function

$$h_w = \begin{cases} 1 & w \cdot x \geq 0 \\ 0 & w \cdot x < 0 \end{cases}$$

Finding a Linear Separator

Viewing the task of finding a linear separator as an ML system:

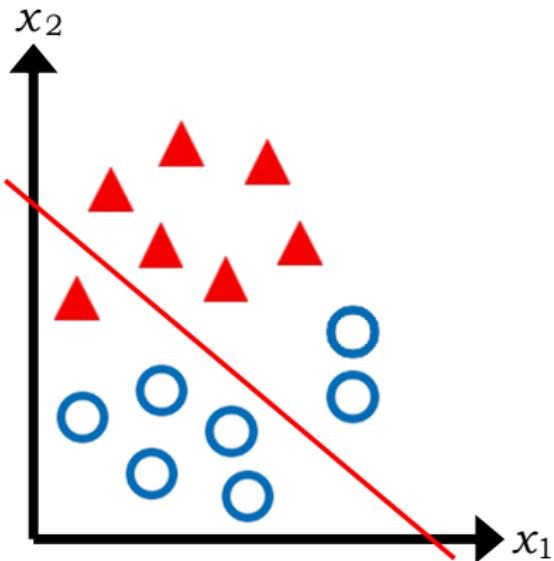
- Data: $\{(\mathbf{x}_i, y_i)\}_{i=1}^N$, with p input attributes
- Hypothesis space:

$$\mathcal{H} = \left\{ h : \mathbb{R}^p \rightarrow [0, 1] \mid h(\mathbf{x}) = \text{Threshold} \left(w_0 + \sum_{j=1}^p w_j x_j \right) \right\}$$

where $\text{Threshold}(t) = 1$ if $t \geq 0$ and 0 otherwise

- Learning algorithm:
 - Loss function: ???
 - Cost function: average loss (whatever that is)
 - Fitting method: ???

Zero-One Loss



- ▶ For a training set made up of input/output pairs,
 $\{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \dots, (\mathbf{x}_k, y_k)\}$
we could define the
zero/one loss

$$L(h_{\mathbf{w}}(\mathbf{x}_i), y_i) = \begin{cases} 0 & \text{if } h_{\mathbf{w}}(\mathbf{x}_i) = y_i \\ 1 & \text{if } h_{\mathbf{w}}(\mathbf{x}_i) \neq y_i \end{cases}$$

- ▶ Summed for the entire set, this is simply the **count** of examples that we get wrong

- ▶ In this example, if data-points marked **○** should be in class 0 (below the line) and those marked **▲** should be in class 1 (above the line) the loss would be equal to 3

Minimizing Zero/One Loss

With a cost function $J(\mathbf{w}) = \sum_{i=1}^N L_{0/1}(y_i, h_{\mathbf{w}}(\mathbf{x}_i))$, finding the weights that minimize $J(\mathbf{w})$ is **computationally difficult**: $J(\mathbf{w})$ is a piecewise constant function and not continuous, so gradient descent won't work.

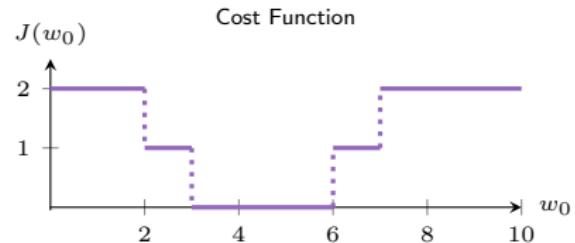
Example: Consider a simple data set with $h_{\mathbf{w}}(x) = \text{Threshold}(w_1 x - w_0)$ (*the change of sign of w_0 is for convenience here*) and w_1 fixed to 1.

Then $h_{\mathbf{w}}(x) = 1$ if $x \geq w_0$ and 0 if $x < w_0$.

Data set:



Search space for weights w_0 :



Perceptron Learning

Using 0/1 loss to update weights as in gradient descent is not feasible.

However, the following weight update rule, called the **perceptron learning rule**, works!

- ① Start with $t = 0$ and initial weights $\mathbf{w}^{(0)}$ (e.g., with each component drawn from $[-1, 1]$ uniformly)
- ② Choose an input \mathbf{x}_i that is **incorrectly classified**
- ③ Update components in weight vector for iteration $t + 1$ using

$$w_j^{(t+1)} \leftarrow w_j^{(t)} + \alpha (y_i - h_{\mathbf{w}^{(t)}}(\mathbf{x}_i)) \cdot x_{ij}$$

for all $j \in \{0, 1, 2, \dots, p\}$

- ④ Increment t and repeat Steps 2 and 3 until **no classification errors remain**

CS 457/557: Machine Learning

Lecture 05-2: Perceptron Learning

Lecture 05-2a: Perceptron Learning

Perceptron Learning

The **perceptron algorithm** iteratively updates weights using the **perceptron learning rule**:

- ① Start with $t = 0$ and initial weights $\mathbf{w}^{(0)}$ (e.g., with each component drawn from $[-1, 1]$ uniformly)
- ② Choose an input \mathbf{x}_i that is **incorrectly classified**
- ③ Update components in weight vector for iteration $t + 1$ using

$$w_j^{(t+1)} \leftarrow w_j^{(t)} + \alpha (y_i - h_{\mathbf{w}^{(t)}}(\mathbf{x}_i)) \cdot x_{ij}$$

for all $j \in \{0, 1, 2, \dots, p\}$

- ④ Increment t and repeat Steps 2 and 3 until **no classification errors remain**

Understanding Weight Updates

Update rule using an **incorrectly classified** input \mathbf{x}_i :

$$w_j^{(t+1)} \leftarrow w_j^{(t)} + \alpha (y_i - h_{\mathbf{w}^{(t)}}(\mathbf{x}_i)) \cdot x_{ij}$$

Understanding the update rule:

- If correct output should be **below** the boundary ($y_i = 0$) but current weights placed it **above** the boundary ($h_{\mathbf{w}^{(t)}}(\mathbf{x}_i) = 1$), then **subtract** each attribute value x_{ij} from the corresponding weight w_j (scaled by α)
- If correct output should be **above** the boundary ($y_i = 1$) but current weights placed it **below** the boundary ($h_{\mathbf{w}^{(t)}}(\mathbf{x}_i) = 0$), then **add** each attribute value x_{ij} to the corresponding weight w_j (scaled by α)

Perceptron Updates

The perceptron update rule shifts the weight vector positively or negatively, trying to get all data on the correct side of the linear decision boundary:

$$w_j^{(t+1)} \leftarrow w_j^{(t)} + \alpha (y_i - h_{\mathbf{w}^{(t)}}(\mathbf{x}_i)) \cdot x_{ij}$$

Example: At iteration t , we have $\mathbf{w}^{(t)} = (0.2, -2.5, 0.6)$ and we examine $\mathbf{x}_i = (1, 0.5, 0.4)$ with $y_i = 1$:

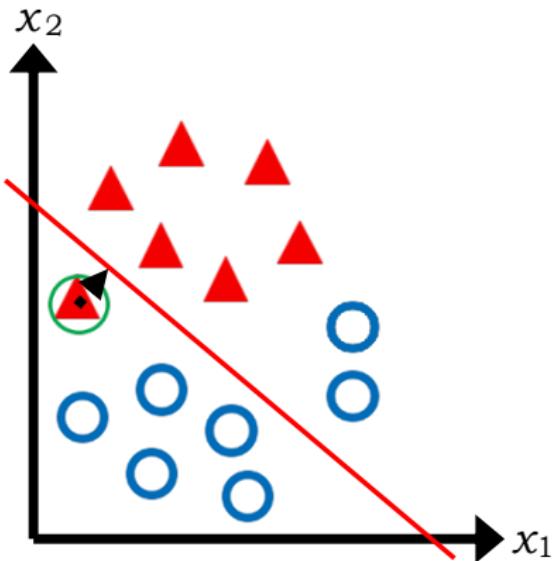
$$\begin{aligned}\mathbf{w}^{(t)} \cdot \mathbf{x}_i &= 0.2 + (-2.5 \cdot 0.5) + (0.6 \cdot 0.4) = -0.81, \\ \text{and } -0.81 < 0, \text{ so } h_{\mathbf{w}^{(t)}}(\mathbf{x}_i) &= 0.\end{aligned}$$

This means \mathbf{x}_i is **misclassified**. With $\alpha = 1$, weights get updated using:

$$\begin{aligned}w_0^{(t+1)} &\leftarrow (w_0^{(t)} + x_{i,0}) = (0.2 + 1) = 1.2 \\ w_1^{(t+1)} &\leftarrow (w_1^{(t)} + x_{i,1}) = (-2.5 + 0.5) = -2.0 \\ w_2^{(t+1)} &\leftarrow (w_2^{(t)} + x_{i,2}) = (0.6 + 0.4) = 1.0\end{aligned}$$

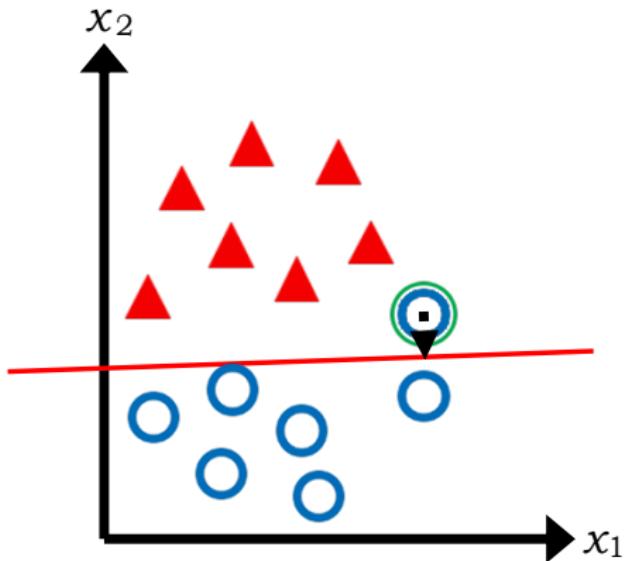
So $\mathbf{w}^{(t+1)} \cdot \mathbf{x}_i = 1.2 + (-2.0 \cdot 0.5) + (1.0 \cdot 0.4) = 0.6$, and $0.6 \geq 0$ so $h_{\mathbf{w}^{(t+1)}}(\mathbf{x}_i) = 1$.

Progress of Perceptron Learning



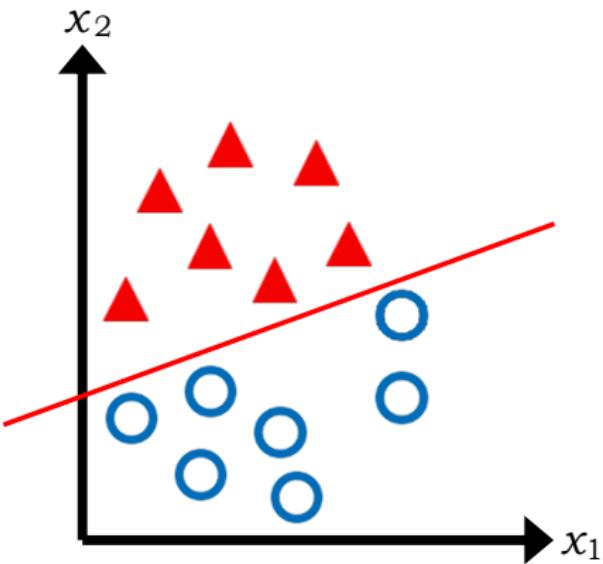
- ▶ For an example like this, we:
 1. Choose a mis-classified item (marked in green)
 2. Compute the weight updates, based on the “distance” away from the boundary (so weights shift more based upon errors in boundary placement that are more extreme)
- ▶ Here, this **adds** to each weight, changing the decision boundary
- ▶ In this example, data-points marked **○** should be in class 0 (below the line) and those marked **▲** should be in class 1 (above the line)

Progress of Perceptron Learning



- ▶ Once we get a new boundary, we repeat the process
 - 1. Choose a mis-classified item (marked in green)
 - 2. Compute the weight updates, based on the “distance” away from the boundary (so weights shift more based upon errors in boundary placement that are more extreme)
- ▶ Here, this **subtracts** from each weight, changing the decision boundary in the other direction
- ▶ In this example, data-points marked **O** should be in class 0 (below the line) and those marked **▲** should be in class 1 (above the line)

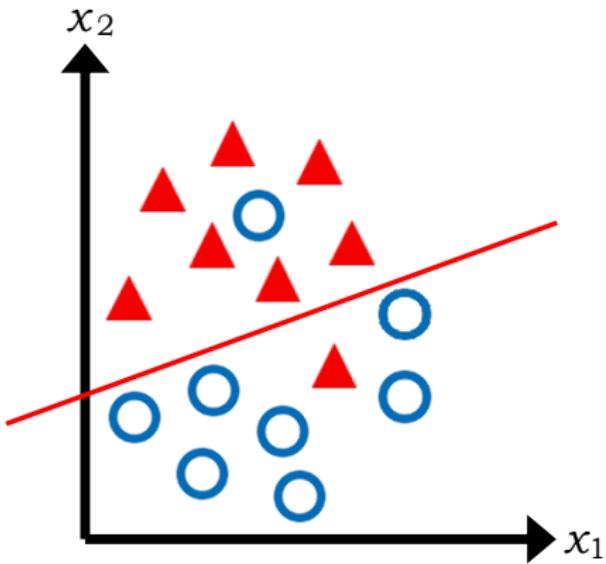
Linear Separability



- ▶ The process of adjusting weights stops when there is no classification error left
- ▶ A data-set is **linearly separable** if a linear separator exists for which there will be no error
- ▶ It is possible that there are **multiple** linear boundaries that achieve this
- ▶ It is also possible that there is **no such** boundary!

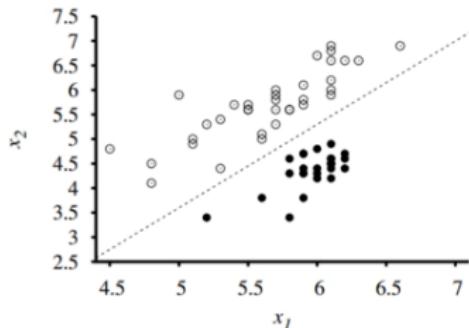
Lecture 05-2b: Linearly Inseparable Data

Linearly Inseparable Data

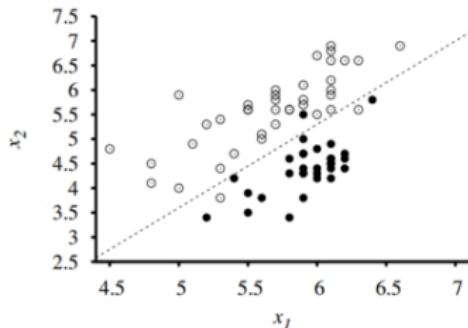


- ▶ Some data **can't** be separated using a linear classifier
 - ▶ Any line drawn will always leave some error
- ▶ The perceptron update method is guaranteed to eventually **converge** to an error-free boundary **if** such a boundary really exists
 - ▶ If it **doesn't** exist, then the most basic version of the algorithm will never terminate

Linearly Inseparable Data



(a)



(b)

Figure 18.15 (a) Plot of two seismic data parameters, body wave magnitude x_1 and surface wave magnitude x_2 , for earthquakes (white circles) and nuclear explosions (black circles) occurring between 1982 and 1990 in Asia and the Middle East (Kebeasy *et al.*, 1998). Also shown is a decision boundary between the classes. (b) The same domain with more data points. The earthquakes and explosions are no longer linearly separable.

- ▶ Unfortunately, data that can't be separated linearly is very common...

Modifying Perceptron Learning

To handle linearly inseparable data, we modify the perceptron algorithm:

- ① Start with $t = 0$ and initial weights $\mathbf{w}^{(0)}$
- ② Choose an input \mathbf{x}_i that is **incorrectly classified**
- ③ Update components in weight vector for iteration $t + 1$ using

$$w_j^{(t+1)} \leftarrow w_j^{(t)} + \alpha (y_i - h_{\mathbf{w}^{(t)}}(\mathbf{x}_i)) \cdot x_{ij}$$

for all $j \in \{0, 1, 2, \dots, p\}$

- ④ Increment t and repeat Steps 2 and 3 until
~~no classification errors remain~~ weights no longer change

To ensure that the weights eventually stop changing, we decrease α over time. As $\alpha \rightarrow 0$, the weights will **stop changing**, and the algorithm will converge. To get to a **least-error** weight vector, this decrease needs to happen **slowly**! E.g., $\alpha^{(t)} \leftarrow \frac{1000}{1000 + t}$.

Modifying Perceptron Learning

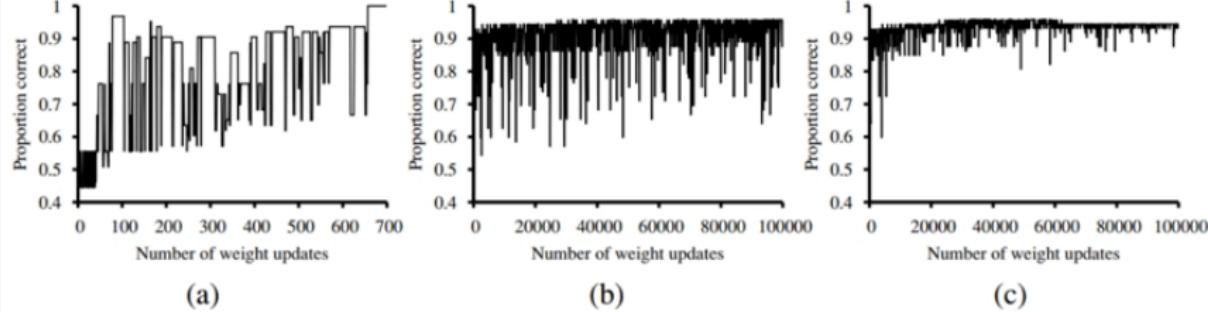


Figure 18.16 (a) Plot of total training-set accuracy vs. number of iterations through the training set for the perceptron learning rule, given the earthquake/explosion data in Figure 18.15(a). (b) The same plot for the noisy, non-separable data in Figure 18.15(b); note the change in scale of the x -axis. (c) The same plot as in (b), with a learning rate schedule $\alpha(t) = 1000/(1000 + t)$.

Lecture 05-2c: Perceptron Loss and Cost Functions

Perceptron Loss and Cost Functions

The perceptron learning rule updates weights from incorrect examples using

$$w_j^{(t+1)} \leftarrow w_j^{(t)} + \alpha (y_i - h_{\mathbf{w}^{(t)}}(\mathbf{x}_i)) \cdot x_{ij}.$$

Natural question: What cost function is the perceptron minimizing?

With a little work, we can derive the perceptron loss function as

$$L_\pi(y_i, h_{\mathbf{w}}(\mathbf{x}_i)) = \max\{0, (1 - 2y_i)\mathbf{w} \cdot \mathbf{x}_i\}$$

and the cost function being minimized as

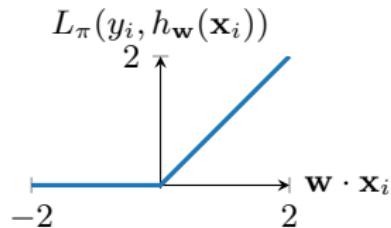
$$J(\mathbf{w}) = \sum_{i=1}^N L_\pi(y_i, h_{\mathbf{w}}(\mathbf{x}_i)).$$

Pictures

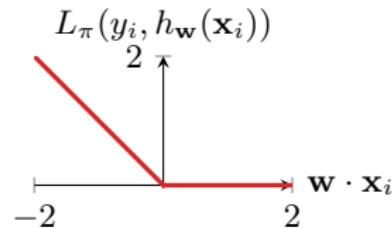
With perceptron loss, the further away a wrongly classified point is from the boundary, the larger the loss:

$$L_\pi(y_i, h_{\mathbf{w}}(\mathbf{x}_i)) = \max\{0, (1 - 2y_i)\mathbf{w} \cdot \mathbf{x}_i\}.$$

With $y_i = 0$, loss is $\max\{0, \mathbf{w} \cdot \mathbf{x}_i\}$



With $y_i = 1$, loss is $\max\{0, -\mathbf{w} \cdot \mathbf{x}_i\}$



Perceptron loss is also called **hinge loss** after the shape of the loss function.

Perceptron Cost Function

$$\begin{aligned} J(\mathbf{w}) &= \sum_{i=1}^N L_\pi(y_i, h_{\mathbf{w}}(\mathbf{x}_i)) \\ &= \sum_{i=1}^N \max\{0, (1 - 2y_i)\mathbf{w} \cdot \mathbf{x}_i\} \\ &= \sum_{i \in TP} \max\{0, (1 - 2y_i)\mathbf{w} \cdot \mathbf{x}_i\} + \sum_{i \in TN} \max\{0, (1 - 2y_i)\mathbf{w} \cdot \mathbf{x}_i\} \\ &\quad + \sum_{i \in FP} \max\{0, (1 - 2y_i)\mathbf{w} \cdot \mathbf{x}_i\} + \sum_{i \in FN} \max\{0, (1 - 2y_i)\mathbf{w} \cdot \mathbf{x}_i\} \\ &= \sum_{i \in TP} \max\{0, (-1)\mathbf{w} \cdot \mathbf{x}_i\} + \sum_{i \in TN} \max\{0, (1)\mathbf{w} \cdot \mathbf{x}_i\} \\ &\quad + \sum_{i \in FP} \max\{0, (1)\mathbf{w} \cdot \mathbf{x}_i\} + \sum_{i \in FN} \max\{0, (-1)\mathbf{w} \cdot \mathbf{x}_i\} \\ &= \sum_{i \in TP} 0 + \sum_{i \in TN} 0 + \sum_{i \in FP} \mathbf{w} \cdot \mathbf{x}_i + \sum_{i \in FN} -\mathbf{w} \cdot \mathbf{x}_i. \end{aligned}$$

(TP is true positives, TN is true negatives, FP is false positives, FN is false negatives)

Perceptron Update as Gradient Descent

The perceptron cost function can be rewritten as

$$J(\mathbf{w}) = \sum_{i \in TP} 0 + \sum_{i \in TN} 0 + \sum_{i \in FP} \mathbf{w} \cdot \mathbf{x}_i + \sum_{i \in FN} -\mathbf{w} \cdot \mathbf{x}_i.$$

If we look at a single example \mathbf{x}_i , we have:

$$\frac{\partial}{\partial w_j} J(\mathbf{w}) = \begin{cases} 0 & \text{if } i \in TP \text{ or } i \in TN \\ x_{ij} & \text{if } i \in FP \\ -x_{ij} & \text{if } i \in FN \end{cases}.$$

The generic gradient descent update rule is

$$w_j^{(t+1)} \leftarrow w_j^{(t)} - \alpha \frac{\partial}{\partial w_j} J(\mathbf{w}).$$

Combining this with above, we have:

- For $i \in FP$, update via: $w_j^{(t+1)} \leftarrow w_j^{(t)} - \alpha \frac{\partial}{\partial w_j} J(\mathbf{w}) \equiv w_j^{(t)} - \alpha x_{ij}$.
- For $i \in FN$, update via: $w_j^{(t+1)} \leftarrow w_j^{(t)} - \alpha \frac{\partial}{\partial w_j} J(\mathbf{w}) \equiv w_j^{(t)} + \alpha x_{ij}$.

The History of the Perceptron



Frank Rosenblatt
1928–1969

Rosenblatt's perceptron played an important role in the history of machine learning. Initially, Rosenblatt simulated the perceptron on an IBM 704 computer at Cornell in 1957, but by the early 1960s he had built special-purpose hardware that provided a direct, parallel implementation of perceptron learning. Many of his ideas were encapsulated in "Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms" published in 1962. Rosenblatt's work was criticized by Marvin Minsky, whose objections were published in the book "Perceptrons", co-authored with

Seymour Papert. This book was widely misinterpreted at the time as showing that neural networks were fatally flawed and could only learn solutions for linearly separable problems. In fact, it only proved such limitations in the case of single-layer networks such as the perceptron and merely conjectured (incorrectly) that they applied to more general network models. Unfortunately, however, this book contributed to the substantial decline in research funding for neural computing, a situation that was not reversed until the mid-1980s. Today, there are many hundreds, if not thousands, of applications of neural networks in widespread use, with examples in areas such as handwriting recognition and information retrieval being used routinely by millions of people.



Figure 4.8 Illustration of the Mark 1 perceptron hardware. The photograph on the left shows how the inputs were obtained using a simple camera system in which an input scene, in this case a printed character, was illuminated by powerful lights, and an image focussed onto a 20×20 array of cadmium sulphide photocells, giving a primitive 400 pixel image. The perceptron also had a patch board, shown in the middle photograph, which allowed different configurations of input features to be tried. Often these were wired up at random to demonstrate the ability of the perceptron to learn without the need for precise wiring, in contrast to a modern digital computer. The photograph on the right shows one of the racks of adaptive weights. Each weight was implemented using a rotary variable resistor, also called a potentiometer, driven by an electric motor thereby allowing the value of the weight to be adjusted automatically by the learning algorithm.

From: C. Bishop, *Pattern Recognition and Machine Learning*. Springer (2006).

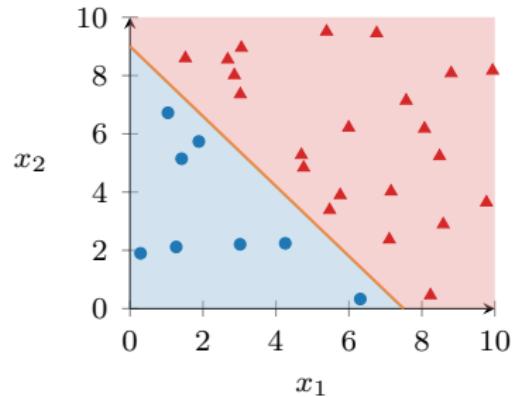
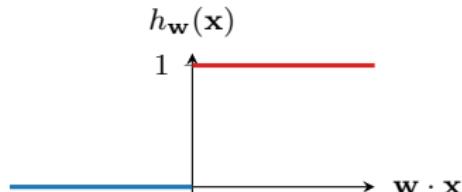
CS 457/557: Machine Learning

Lecture 05-3: Logistic Regression

Hard Thresholds are Hard!

The perceptron algorithm uses a **hard threshold** function for classification:

$$h_{\mathbf{w}}(\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{w} \cdot \mathbf{x} \geq 0 \\ 0 & \text{if } \mathbf{w} \cdot \mathbf{x} < 0 \end{cases}.$$



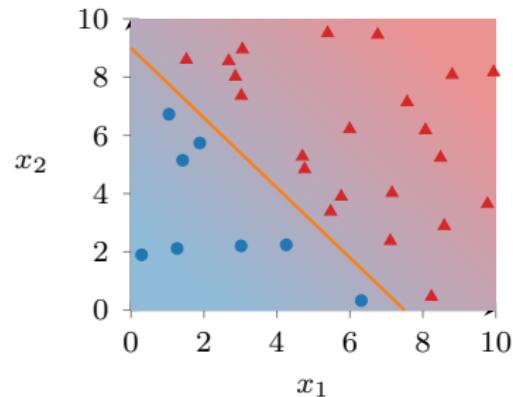
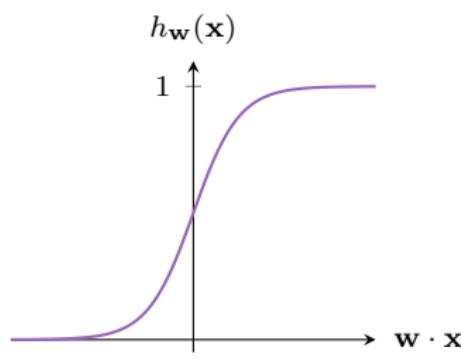
This gives a **yes/no** answer everywhere!

The threshold function itself is discontinuous (non-differentiable) at 0.

Soft Thresholds and the Logistic Function

An alternative **soft threshold** function returns values in the range $[0, 1]$. Such values can be interpreted as the probability of belonging to the positive class. The **logistic function** σ does this:

$$h_{\mathbf{w}}(\mathbf{x}) = \sigma(\mathbf{w} \cdot \mathbf{x}).$$

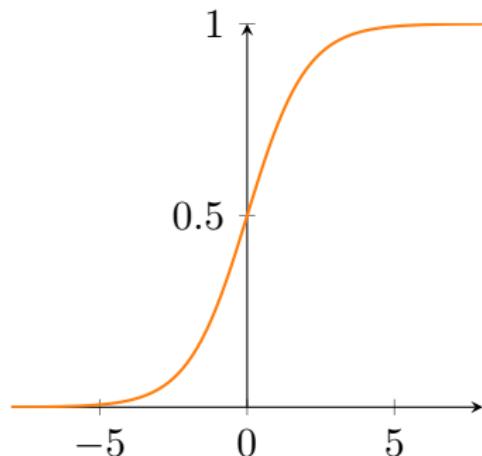


Decision boundary is all \mathbf{x} with $h_{\mathbf{w}}(\mathbf{x}) = 0.5$

Properties of the Logistic Function

The **standard logistic function** is $\sigma : \mathbb{R} \rightarrow [0, 1]$ is defined as

$$\sigma(t) = \frac{1}{1 + e^{-t}} = \frac{1}{1 + \exp(-t)}.$$



- σ is a **sigmoid** function
- Output is always in $[0, 1]$
- $\sigma(0) = 0.5$
- $\lim_{t \rightarrow \infty} \sigma(t) = 1$
- $\lim_{t \rightarrow -\infty} \sigma(t) = 0$
- σ is everywhere differentiable, with $\sigma'(t) = \sigma(t)(1 - \sigma(t))$

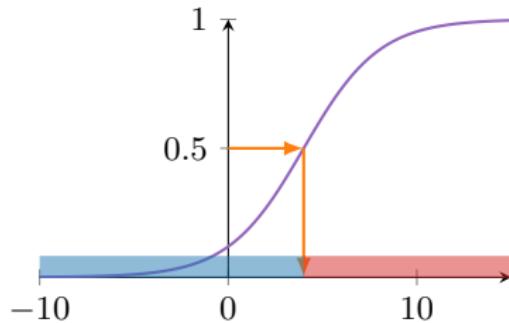
Logistic Regression for Classification

With the logistic function for thresholding, we have

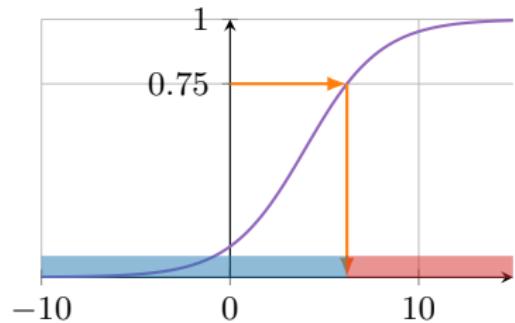
$$h_{\mathbf{w}}(\mathbf{x}) = \sigma(\mathbf{w} \cdot \mathbf{x}) = \frac{1}{1 + e^{-\mathbf{w} \cdot \mathbf{x}}}.$$

However, *by itself* this is **not** a classifier! To turn it into a **classifier** (i.e., an algorithm that produces labels), we add hard thresholding with a cutoff value t : Predicted label is 1 if $h_{\mathbf{w}}(\mathbf{x}) \geq t$, 0 otherwise.

$w_0 = -2, w_1 = 0.5$ with $t = 0.5$:

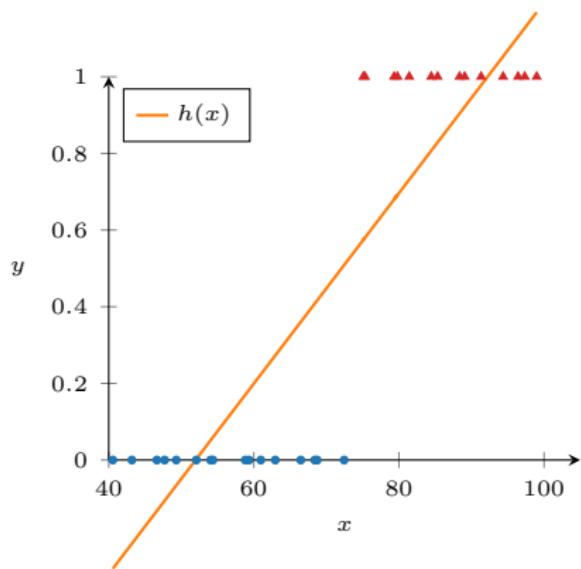


$w_0 = -2, w_1 = 0.5$ with $t = 0.75$:

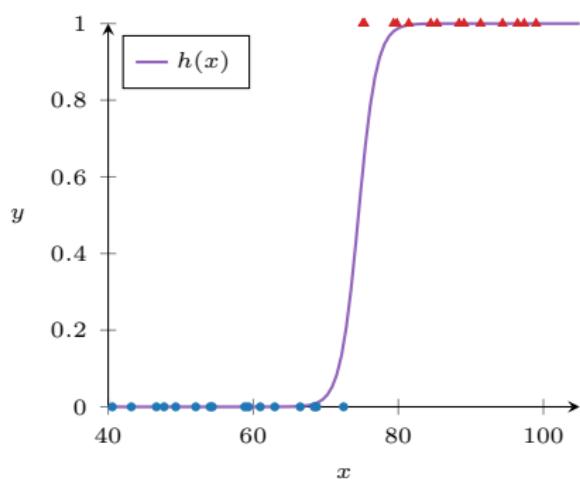


Example of Logistic Regression

Fitting a line to the data as in linear regression doesn't work:



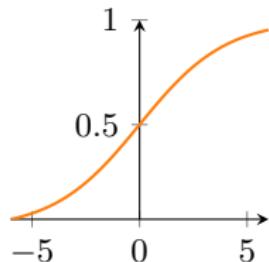
Fitting a linear function wrapped in a logistic function does work:



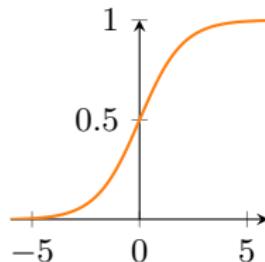
This is called **logistic regression!**

Some Parameter Choices for Logistic Regression

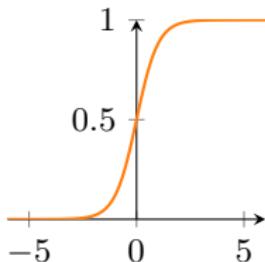
$w_0 = 0, w_1 = 0.5:$



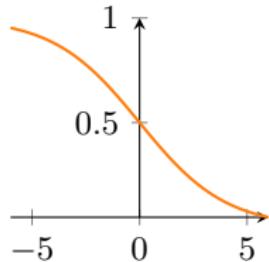
$w_0 = 0, w_1 = 1:$



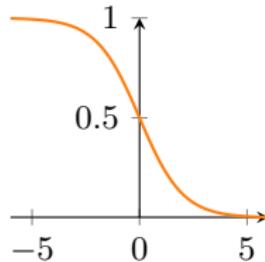
$w_0 = 0, w_1 = 2:$



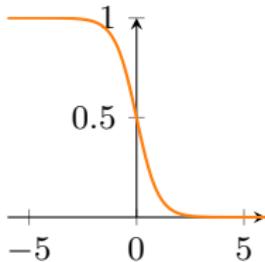
$w_0 = 0, w_1 = -0.5:$



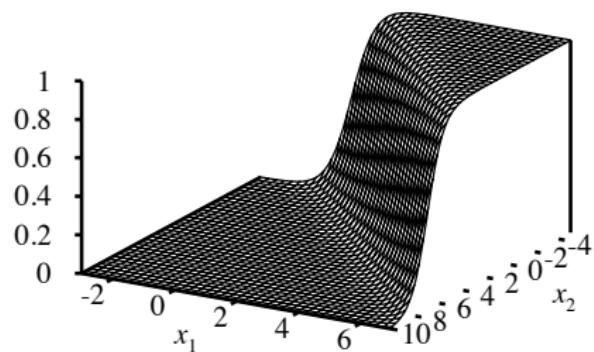
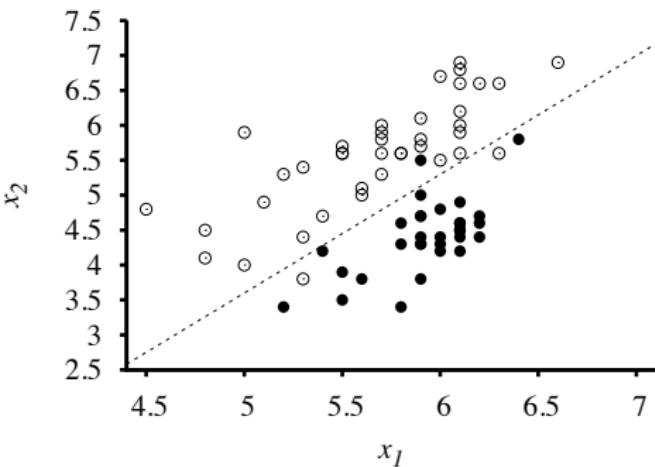
$w_0 = 0, w_1 = -1:$



$w_0 = 0, w_1 = -2:$



Applying the Logistic



- When we have data that is not linearly separable, our hard threshold still has to make a hard decision
- With the logistic, we get a smooth surface where things close to the boundary between classes are only *probably* in one or the other

Lecture 05-3b: Loss Functions for Logistic Regression

Logistic Regression Components

Logistic regression as an ML system:

- Data: $\{(\mathbf{x}_i, y_i)\}_{i=1}^N$, with p input attributes
- Hypothesis space:

$$\mathcal{H} = \left\{ h : \mathbb{R}^p \rightarrow [0, 1] \mid h(\mathbf{x}) = \sigma \left(w_0 + \sum_{j=1}^p w_j x_j \right) \right\}$$

where $\sigma(t) = \frac{1}{1+e^{-t}}$

- Learning algorithm:
 - Loss function: ???
 - Cost function: average loss (whatever that is)
 - Fitting method: ???

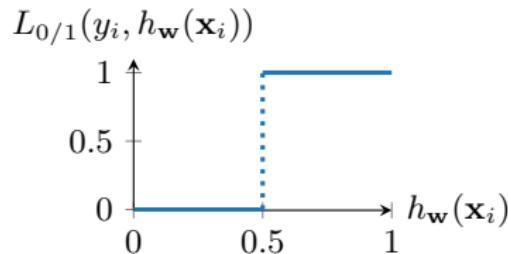
Loss Functions for Logistic Regression:

0/1 Loss

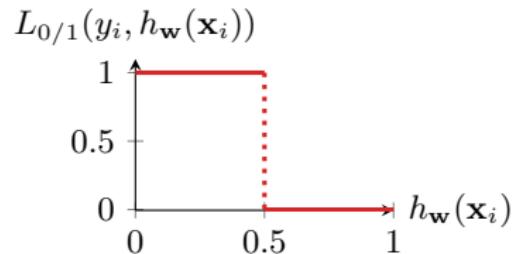
We could adapt the 0/1 loss function for logistic regression by using a hard threshold with cutoff t :

$$L_{0/1}(y_i, h_{\mathbf{w}}(\mathbf{x}_i)) = \begin{cases} 0 & \text{if } y_i = 1 \text{ and } h_{\mathbf{w}}(\mathbf{x}_i) \geq t \\ 0 & \text{if } y_i = 0 \text{ and } h_{\mathbf{w}}(\mathbf{x}_i) < t \\ 1 & \text{otherwise} \end{cases}$$

With $y_i = 0$ and $t = 0.5$, loss is:



With $y_i = 1$ and $t = 0.5$, loss is:



- Hard to optimize; Penalizes near-misses and absolute misses the same
- Comparing the predicted *probability* to the actual label gives better granularity!

Loss Functions for Logistic Regression: Squared Loss

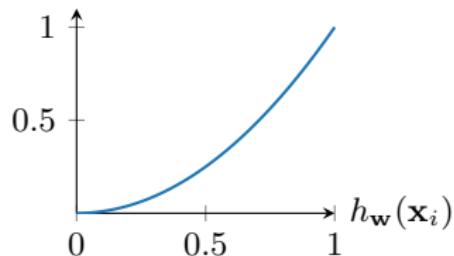
We can also use squared loss for logistic regression:

$$L_2(y_i, h_{\mathbf{w}}(\mathbf{x}_i)) = (y_i - h_{\mathbf{w}}(\mathbf{x}_i))^2$$

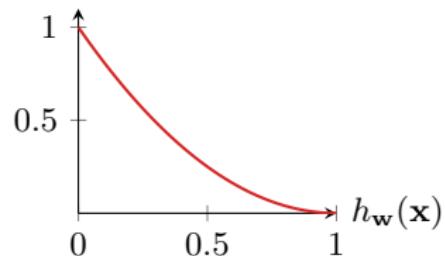
With $y_i = 0$, loss is $(h_{\mathbf{w}}(\mathbf{x}_i))^2$

With $y_i = 1$, loss is $(1 - h_{\mathbf{w}}(\mathbf{x}_i))^2$

$$L_2(y_i, h_{\mathbf{w}}(\mathbf{x}_i))$$



$$L_2(y_i, h_{\mathbf{w}}(\mathbf{x}_i))$$



- $h_{\mathbf{w}}(\mathbf{x}_i) \in [0, 1]$, so loss is bounded between 0 and 1
- Does not lead to a convex objective for finding \mathbf{w}
(gradient descent is not guaranteed to converge to global minimum)

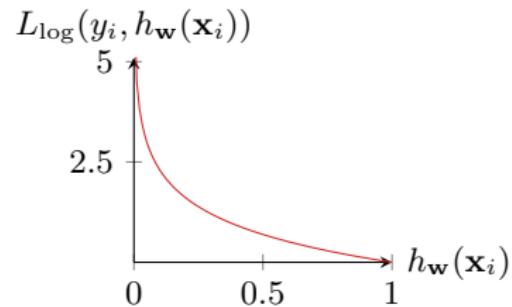
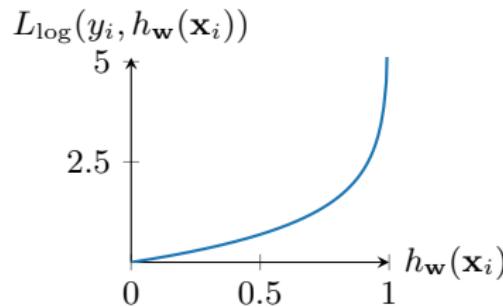
Loss Functions for Logistic Regression: Log-Loss

Another loss function is **log-loss (cross entropy loss)**:

$$L_{\text{log}}(y_i, h_{\mathbf{w}}(\mathbf{x}_i)) = \begin{cases} -\log(h_{\mathbf{w}}(\mathbf{x}_i)) & \text{if } y_i = 1 \\ -\log(1 - h_{\mathbf{w}}(\mathbf{x}_i)) & \text{if } y_i = 0 \end{cases}$$
$$= -1(y_i \log(h_{\mathbf{w}}(\mathbf{x}_i)) + (1 - y_i) \log(1 - h_{\mathbf{w}}(\mathbf{x}_i)))$$

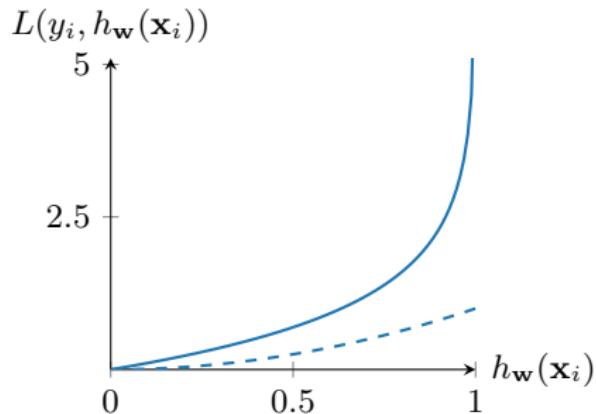
Recall: $\log_b x = y$ if and only if $b^y = x$. Typically the natural logarithm is used here.

With $y_i = 0$, loss is $-\ln(1 - h_{\mathbf{w}}(\mathbf{x}_i))$: With $y_i = 1$, loss is $-\ln(h_{\mathbf{w}}(\mathbf{x}_i))$:

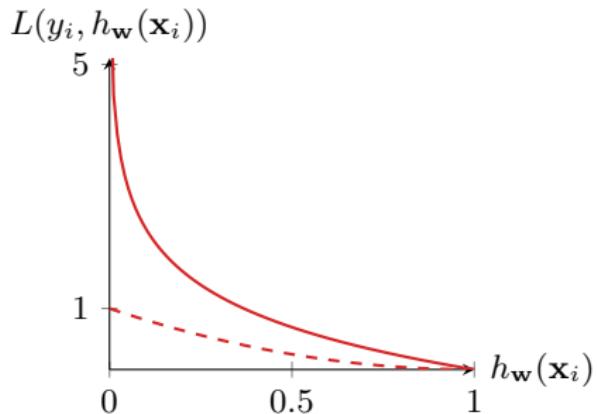


Comparing Squared Loss and Log-Loss

With $y_i = 0$:



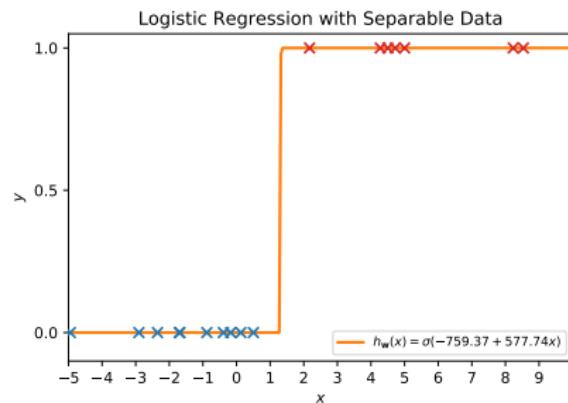
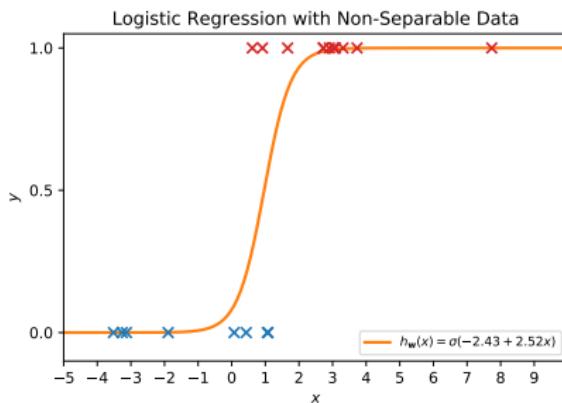
With $y_i = 1$:



- Log-loss is shown as solid lines, squared loss is shown as dashed lines
- Log-loss penalizes wrong answers **more severely** than squared loss
- Log-loss leads to a **convex** cost function!
- Log-loss also has connections to maximum likelihood estimation

Logistic Regression with Regularization

Using average log-loss as the cost function gives us a **convex** function, which we can minimize with gradient descent. There is one issue though...



With separable data, making the curve steeper always reduces the cost! Gradient descent can make the curve steeper by increasing the magnitude of the weights, so it will choose to do so unless we **penalize** large weights using **regularization**.

Logistic Regression Components, Revisited

Logistic regression an ML system:

- Data: $\{(\mathbf{x}_i, y_i)\}_{i=1}^N$, with p input attributes
- Hypothesis space:

$$\mathcal{H} = \left\{ h : \mathbb{R}^p \rightarrow [0, 1] \mid h(\mathbf{x}) = \sigma \left(w_0 + \sum_{j=1}^p w_j x_j \right) \right\}$$

where $\sigma(t) = \frac{1}{1+e^{-t}}$

- Learning algorithm:
 - Loss function: **log-loss**
 - Cost function: average loss **with regularization**
 - Fitting method: **gradient descent**
(we just need the partial derivatives of the cost function!)

Another Look at the Cost Function

With some more algebra (and augmented zeroth attribute):

$$\begin{aligned} J(\mathbf{w}) &= \frac{1}{N} \sum_{i=1}^N L_{\log}(y_i, h_{\mathbf{w}}(\mathbf{x}_i)) \\ &= \frac{1}{N} \sum_{i=1}^N - \left[y_i \ln \left(\frac{1}{1 + e^{-(\mathbf{w} \cdot \mathbf{x}_i)}} \right) + (1 - y_i) \ln \left(1 - \frac{1}{1 + e^{-(\mathbf{w} \cdot \mathbf{x}_i)}} \right) \right] \\ &= \vdots \\ &= -\frac{1}{N} \sum_{i=1}^N \left[y_i (\mathbf{w} \cdot \mathbf{x}_i) + \ln \left(\frac{1}{1 + e^{(\mathbf{w} \cdot \mathbf{x}_i)}} \right) \right] \\ &= -\frac{1}{N} \sum_{i=1}^N [y_i (\mathbf{w} \cdot \mathbf{x}) + \ln(\sigma(-\mathbf{w} \cdot \mathbf{x}))] \end{aligned}$$

- Easier to take the derivative of the above for gradient descent

History of Logistic Regression (1838–1847)

- ▶ The logistic function and its name come from three papers by Pierre François Verhulst (right), a statistician and student of Alphonse Quetelet (left)
- ▶ They were interested in modeling human population growth, which will tend to grow exponentially unless checked, but has an upper bound (**equilibrium**) at which it maxes out and stops growing
- ▶ The Sigmoid curve was a good fit for real population data for France, Belgium, and Russia up to the year 1833



History of Logistic Regression (20th C.)

- ▶ The logistic was re-discovered by Raymond Pearl (left) and Lowell Reed (right) in the 1920's
- ▶ They later discovered Verhulst's earlier work, and credited him, but his logistic terminology didn't really catch on until the work of others, after WWII
- ▶ Pearl and collaborators went on to apply the logistic curve to models of human and fruit fly populations, as well as to the growth of cantaloupes
- ▶ In the 40's and 50's, statisticians working to model **bioassay** (effects of medicines and other substances on living tissues) popularized the use of the logistic and its name
- ▶ Due to computational conveniences, this became more popular than other models



CS 457/557: Machine Learning

Lecture 05-4: Evaluating Classifiers

Lecture 05-4a: Evaluating Classifiers

Evaluating a Classifier

Definition

Two metrics for evaluating a classifier are:

$$\text{accuracy} = \frac{\# \text{ of points classified correctly}}{\# \text{ of points total}}$$

$$\text{error} = \frac{\# \text{ of points classified incorrectly}}{\# \text{ of points total}} = 1 - \text{accuracy}$$

Example: Classifying emails as **spam** (1) or **ham** (0)

- Sample contains 100 emails, 5 of which are **spam**

	Accuracy	Error
Classifier 1: nothing is spam	0.95	0.05
Classifier 2: everything is spam	0.05	0.95

Misclassification Errors

		Truth (y_i)	
		1	0
Prediction (\hat{y}_i)	1	TP: True positives	FP: False positives
	0	FN: False negatives	TN: True negatives

$$\text{accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}} = \frac{\text{TP} + \text{TN}}{n}$$

$$\text{precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

(of points that were **predicted** positive,
what proportion actually were positive?)

$$\text{recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

(of points that were **actually** positive,
what proportion were predicted correctly?)

Confusion Matrices

Example revisited: Classifying emails as **spam** (1) or **ham** (0)

- Sample contains 100 emails, 5 of which are **spam**
- Classifier 1: nothing is spam
- Classifier 2: everything is spam
- Classifier 3: emails containing “money” are spam
(6 legit emails, 3 actual spam emails)

	y_i	
	1	0
\hat{y}_i	1	TP
	0	FN
		FP
		TN

Definition

A **confusion matrix** is a table containing counts of true positives, true negatives, false positives, and false negatives for a classifier.

	y_i	
Clf. 1	1	0
\hat{y}_i	1	0
	5	95

	y_i	
Clf. 2	1	0
\hat{y}_i	1	0
	5	95

	y_i	
Clf. 3	1	0
\hat{y}_i	1	0
	3	6
	2	89

Precision & Recall

	Accuracy	Precision	Recall
	$\frac{TP+TN}{n}$	$\frac{TP}{TP+FP}$	$\frac{TP}{TP+FN}$
Clf. 1	$\frac{0 + 95}{100} = 0.95$	$\frac{0}{0 + 0} = \text{undef}$	$\frac{0}{0 + 5} = 0$
Clf. 2	$\frac{5 + 0}{100} = 0.05$	$\frac{5}{5 + 95} = 0.05$	$\frac{5}{5 + 0} = 1.00$
Clf. 3	$\frac{3 + 89}{100} = 0.92$	$\frac{3}{3 + 6} = 0.33$	$\frac{3}{3 + 2} = 0.60$
Clf. 1	$\frac{y_i}{1 \quad 0}$	$\frac{y_i}{1 \quad 0}$	$\frac{y_i}{1 \quad 0}$
\hat{y}_i	$\begin{matrix} 1 & 0 & 0 \\ 0 & 5 & 95 \end{matrix}$	$\begin{matrix} 1 & 5 & 95 \\ 0 & 0 & 0 \end{matrix}$	$\begin{matrix} 1 & 3 & 6 \\ 0 & 2 & 89 \end{matrix}$

Precision & Recall for a Threshold Classifier

For a soft threshold classifier, precision and recall **depend** on the cutoff value t :

$$\hat{y} = \begin{cases} 1 & \text{if } h_{\mathbf{w}}(\mathbf{x}) \geq t \\ 0 & \text{otherwise} \end{cases}$$

- If $t = 0$: Everything is spam! Precision is low, but recall is 100%!
- If $t = 1$: Nothing is spam Precision is high, but recall is low

Precision and recall are **inversely** related: improving one generally decreases the other.

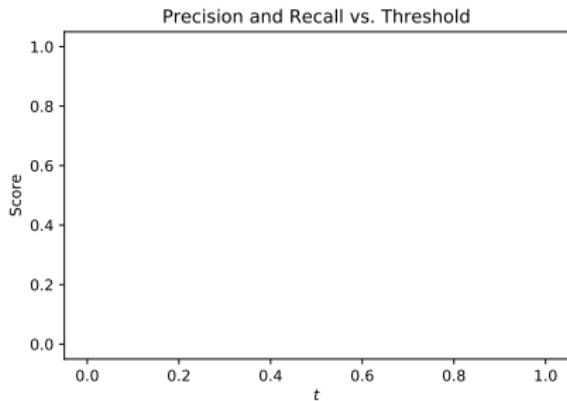
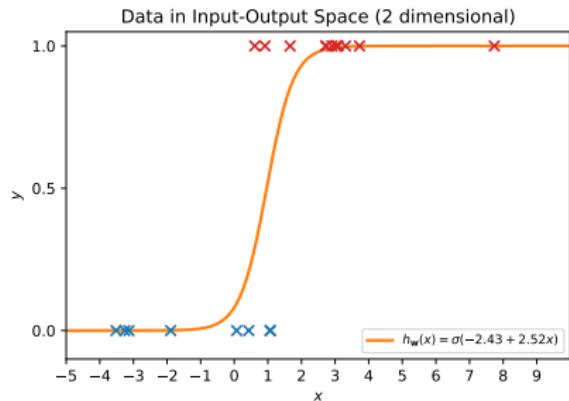
Precision penalizes **false positives**; Recall penalizes **false negatives**.

- Criminal trial: avoid false positives (want high precision)
- Disease screening: avoid false negatives (want high recall)

Lecture 05-4b: Evaluation using Curves

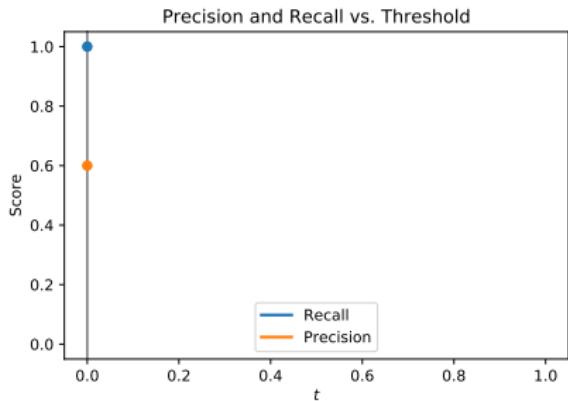
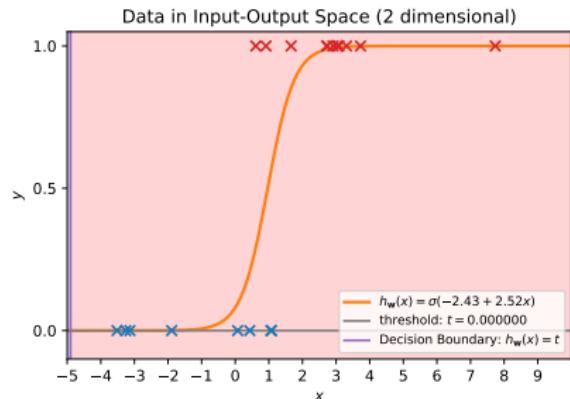
Precision & Recall vs. Threshold

A soft threshold classifier is really a **family** of classifiers defined by the choice of cutoff value t . How does each classifier in the family do?



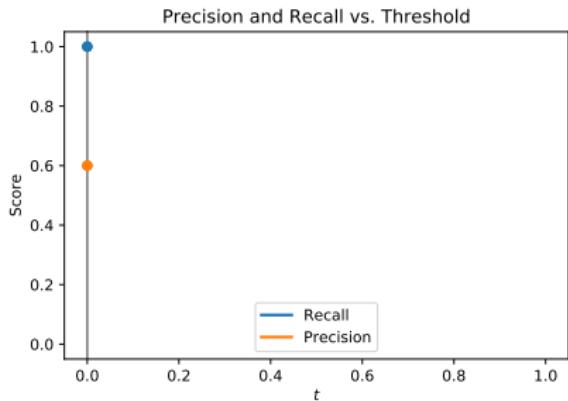
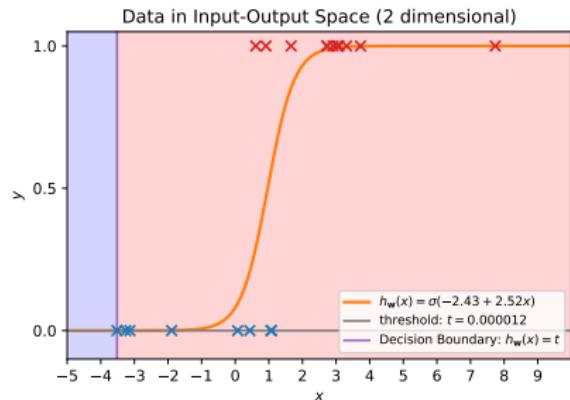
Precision & Recall vs. Threshold

A soft threshold classifier is really a **family** of classifiers defined by the choice of cutoff value t . How does each classifier in the family do?



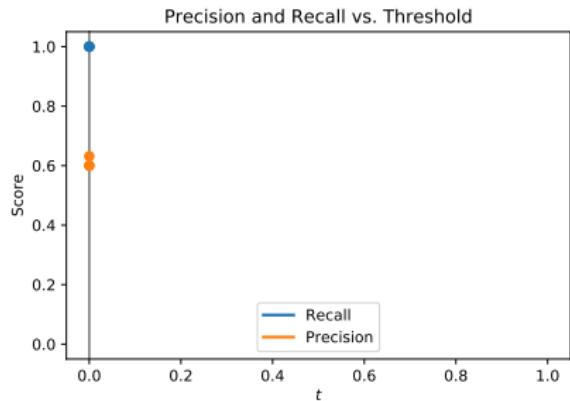
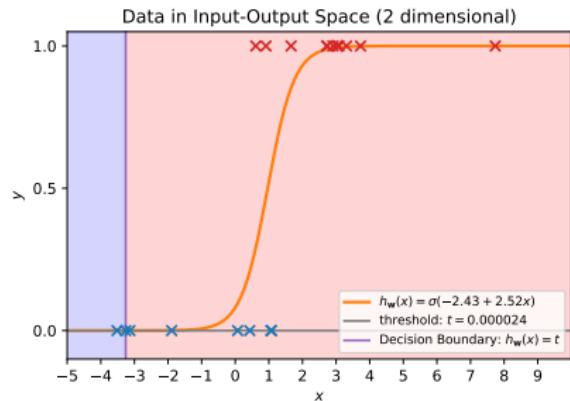
Precision & Recall vs. Threshold

A soft threshold classifier is really a **family** of classifiers defined by the choice of cutoff value t . How does each classifier in the family do?



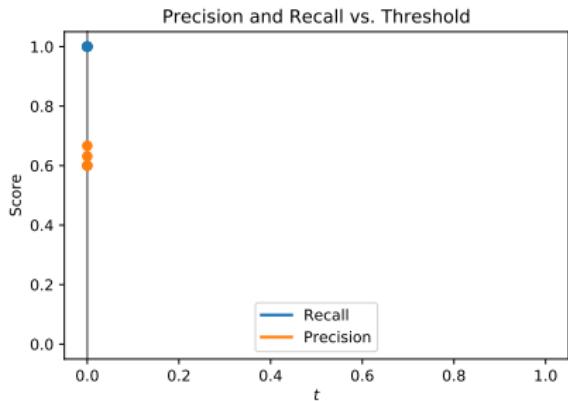
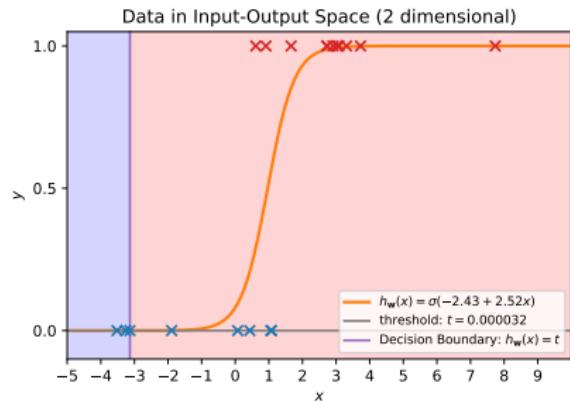
Precision & Recall vs. Threshold

A soft threshold classifier is really a **family** of classifiers defined by the choice of cutoff value t . How does each classifier in the family do?



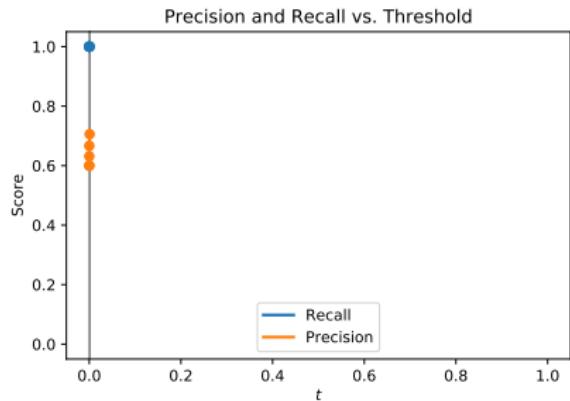
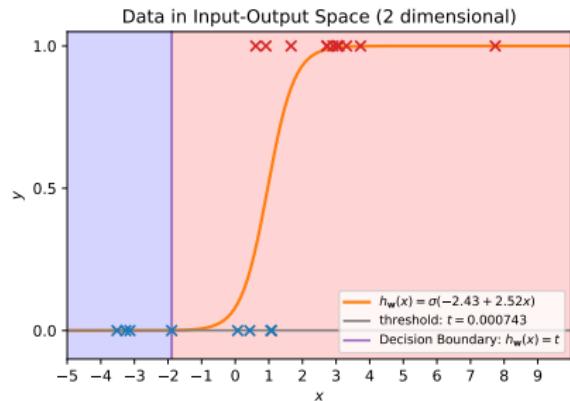
Precision & Recall vs. Threshold

A soft threshold classifier is really a **family** of classifiers defined by the choice of cutoff value t . How does each classifier in the family do?



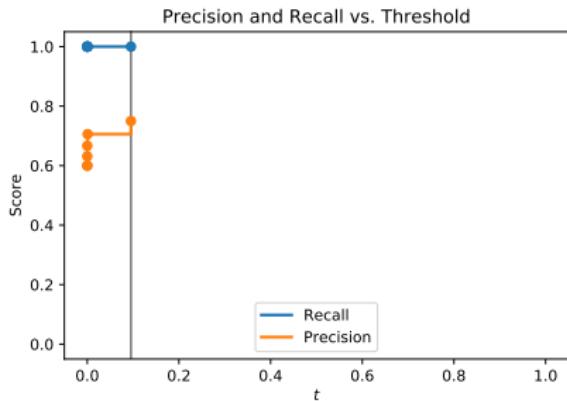
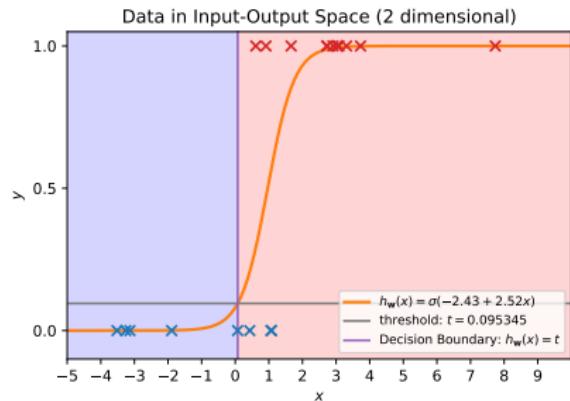
Precision & Recall vs. Threshold

A soft threshold classifier is really a **family** of classifiers defined by the choice of cutoff value t . How does each classifier in the family do?



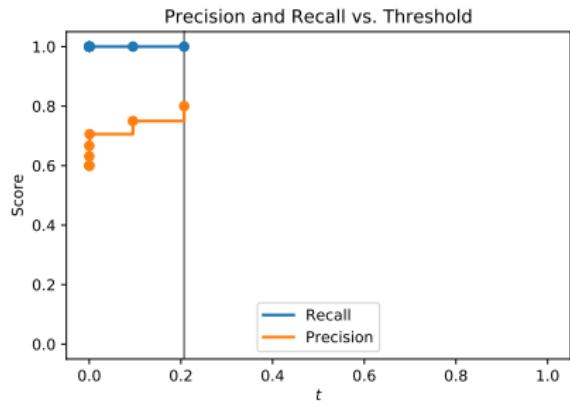
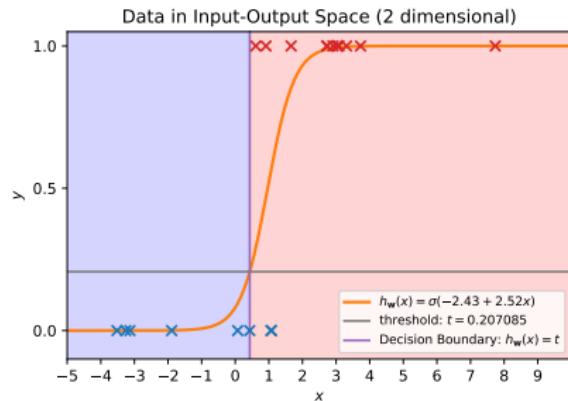
Precision & Recall vs. Threshold

A soft threshold classifier is really a **family** of classifiers defined by the choice of cutoff value t . How does each classifier in the family do?



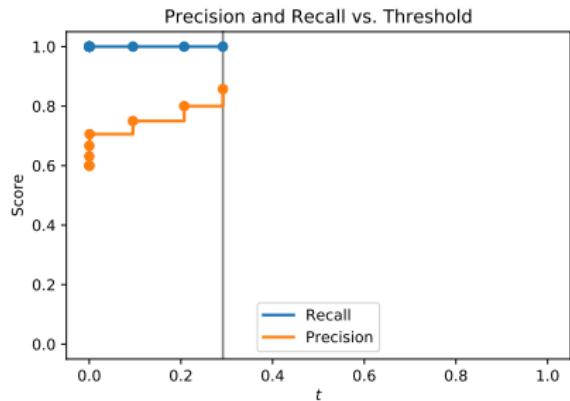
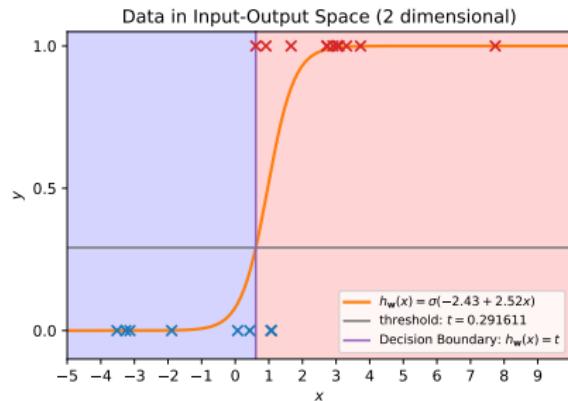
Precision & Recall vs. Threshold

A soft threshold classifier is really a **family** of classifiers defined by the choice of cutoff value t . How does each classifier in the family do?



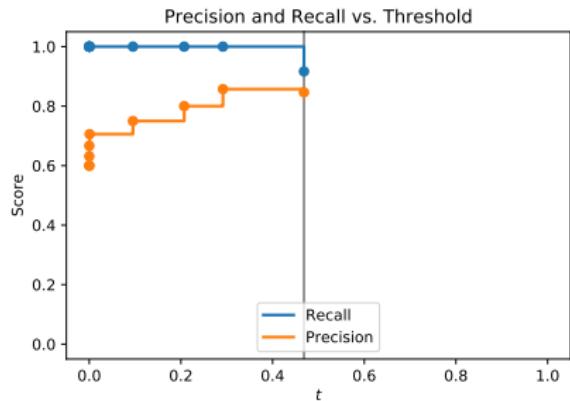
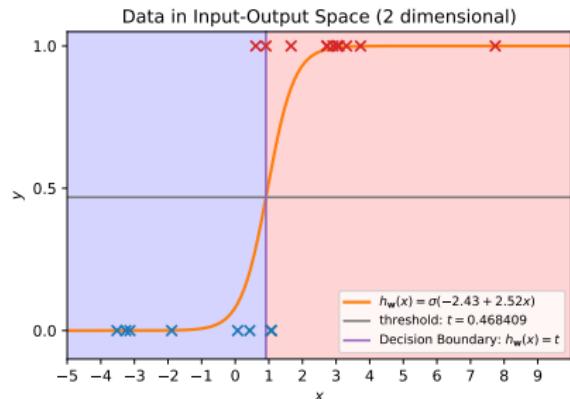
Precision & Recall vs. Threshold

A soft threshold classifier is really a **family** of classifiers defined by the choice of cutoff value t . How does each classifier in the family do?



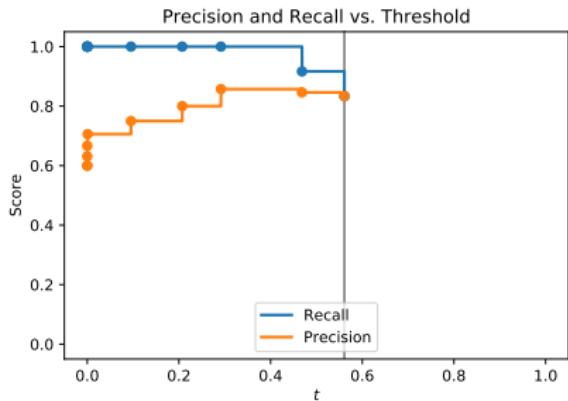
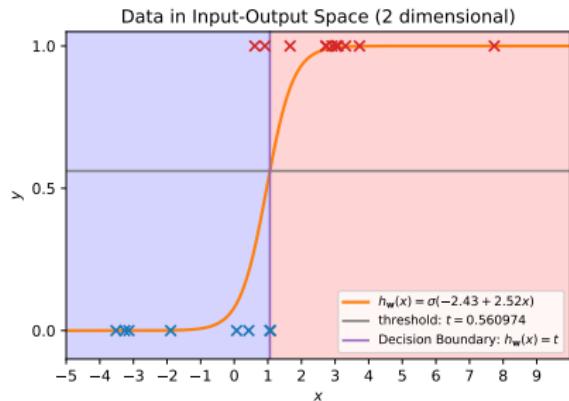
Precision & Recall vs. Threshold

A soft threshold classifier is really a **family** of classifiers defined by the choice of cutoff value t . How does each classifier in the family do?



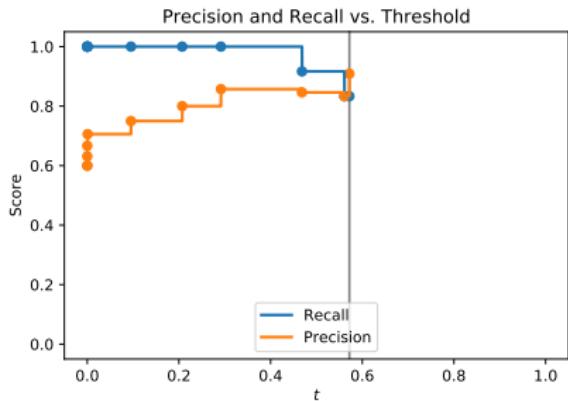
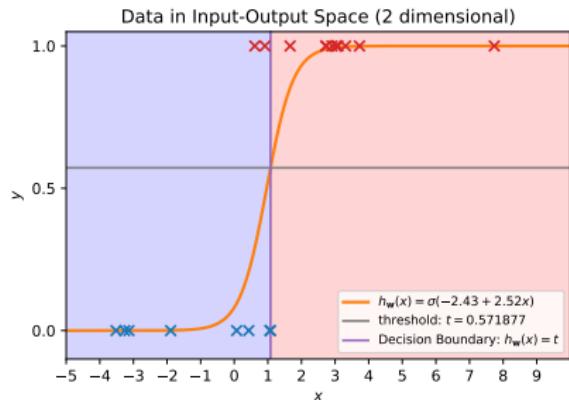
Precision & Recall vs. Threshold

A soft threshold classifier is really a **family** of classifiers defined by the choice of cutoff value t . How does each classifier in the family do?



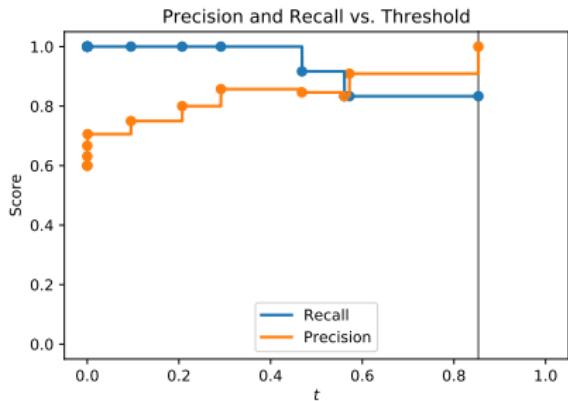
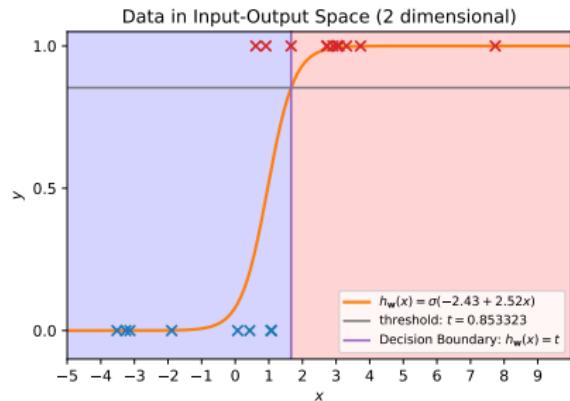
Precision & Recall vs. Threshold

A soft threshold classifier is really a **family** of classifiers defined by the choice of cutoff value t . How does each classifier in the family do?



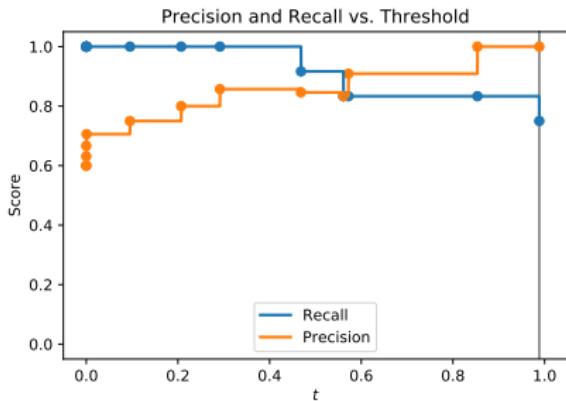
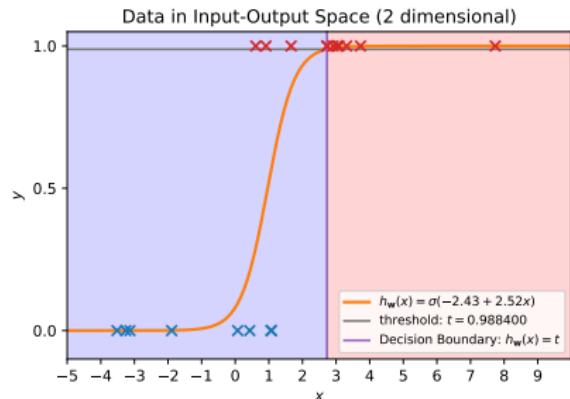
Precision & Recall vs. Threshold

A soft threshold classifier is really a **family** of classifiers defined by the choice of cutoff value t . How does each classifier in the family do?



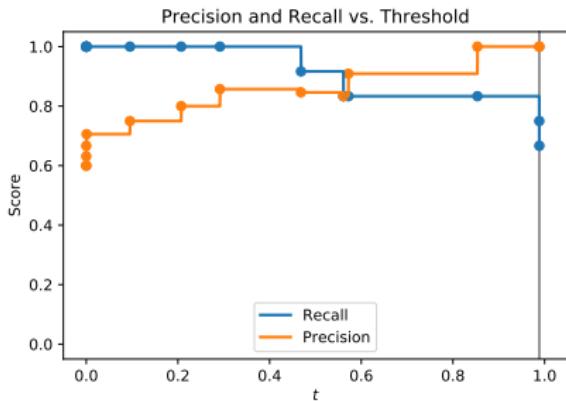
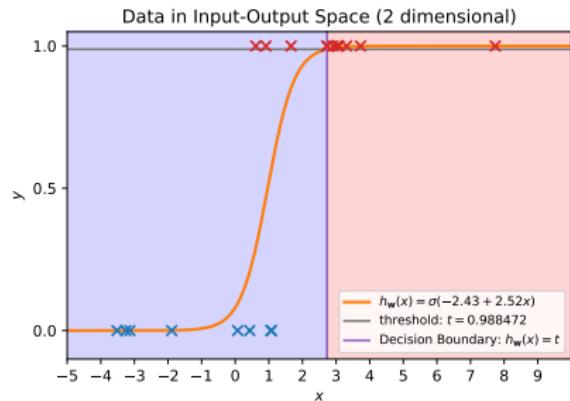
Precision & Recall vs. Threshold

A soft threshold classifier is really a **family** of classifiers defined by the choice of cutoff value t . How does each classifier in the family do?



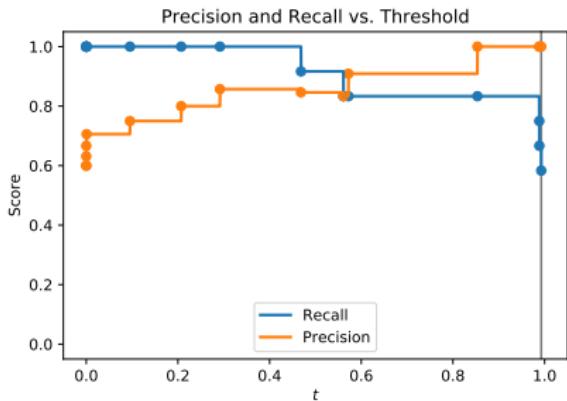
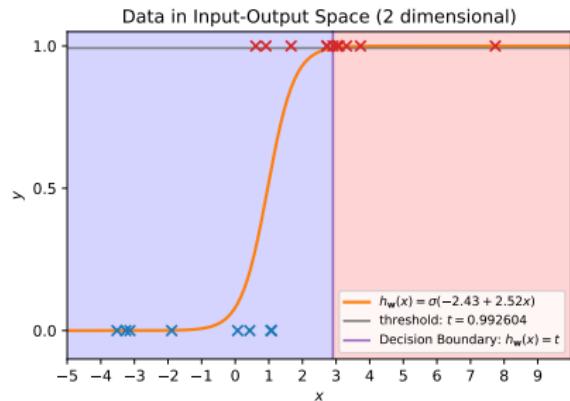
Precision & Recall vs. Threshold

A soft threshold classifier is really a **family** of classifiers defined by the choice of cutoff value t . How does each classifier in the family do?



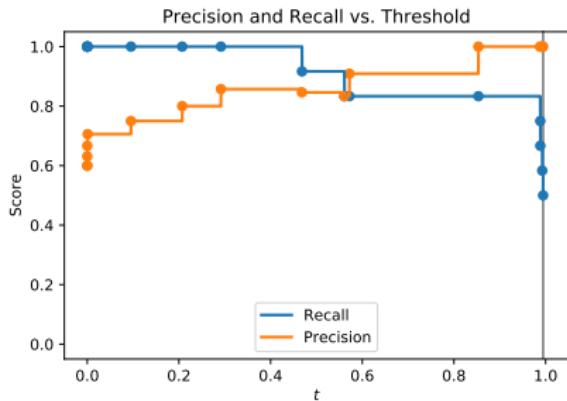
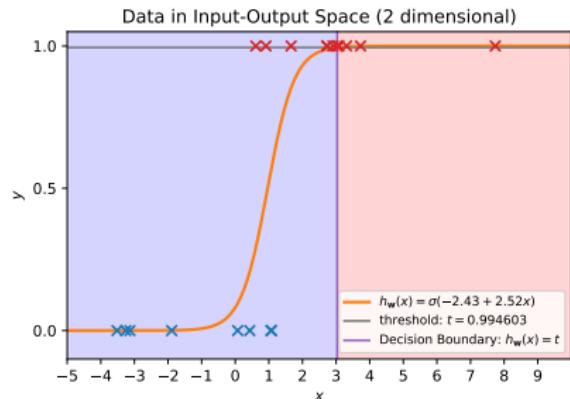
Precision & Recall vs. Threshold

A soft threshold classifier is really a **family** of classifiers defined by the choice of cutoff value t . How does each classifier in the family do?



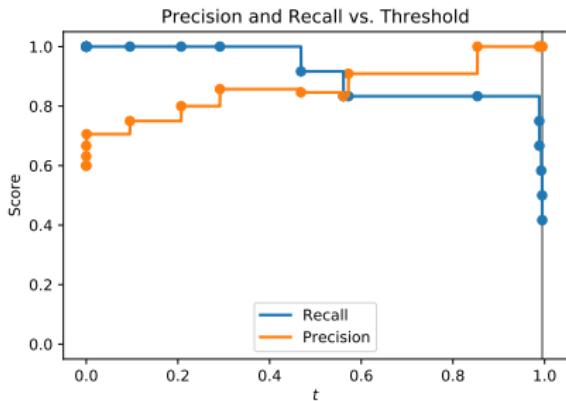
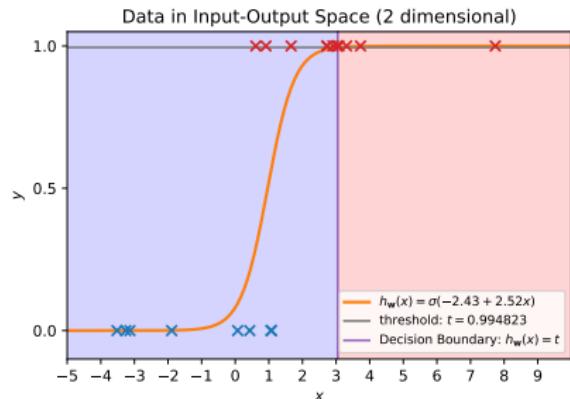
Precision & Recall vs. Threshold

A soft threshold classifier is really a **family** of classifiers defined by the choice of cutoff value t . How does each classifier in the family do?



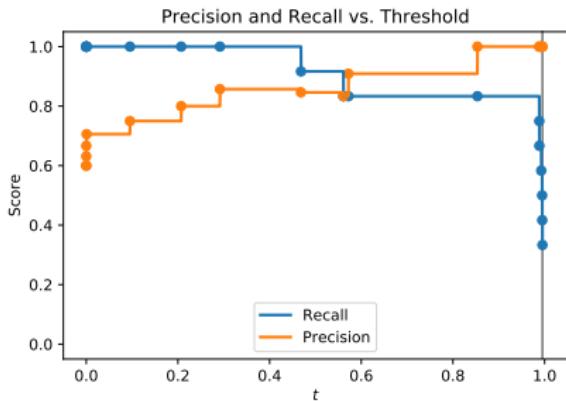
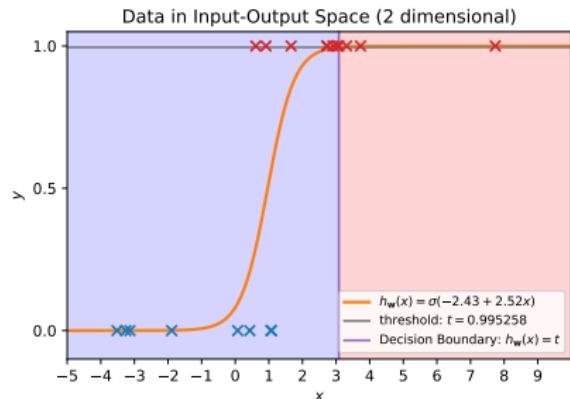
Precision & Recall vs. Threshold

A soft threshold classifier is really a **family** of classifiers defined by the choice of cutoff value t . How does each classifier in the family do?



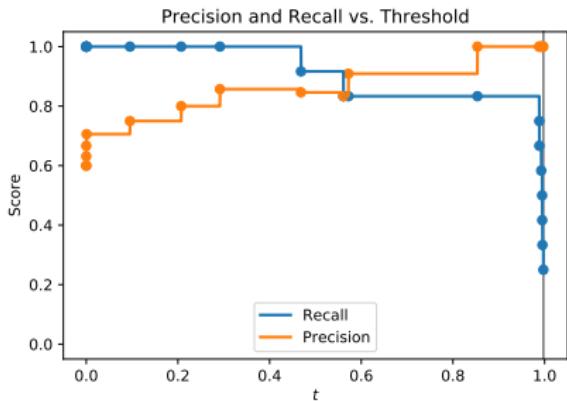
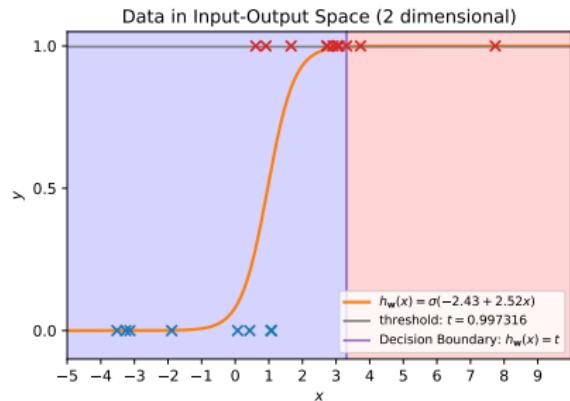
Precision & Recall vs. Threshold

A soft threshold classifier is really a **family** of classifiers defined by the choice of cutoff value t . How does each classifier in the family do?



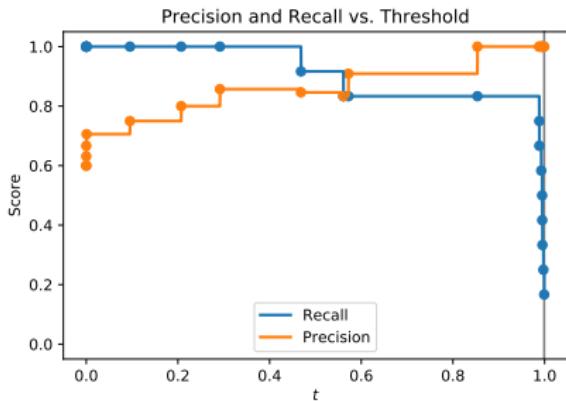
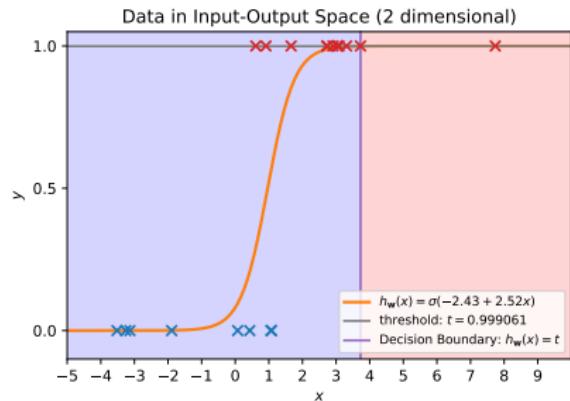
Precision & Recall vs. Threshold

A soft threshold classifier is really a **family** of classifiers defined by the choice of cutoff value t . How does each classifier in the family do?



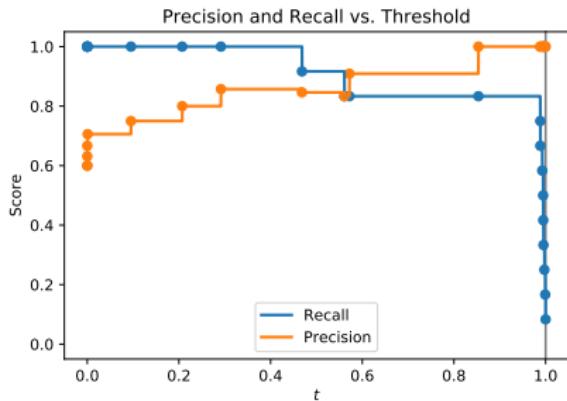
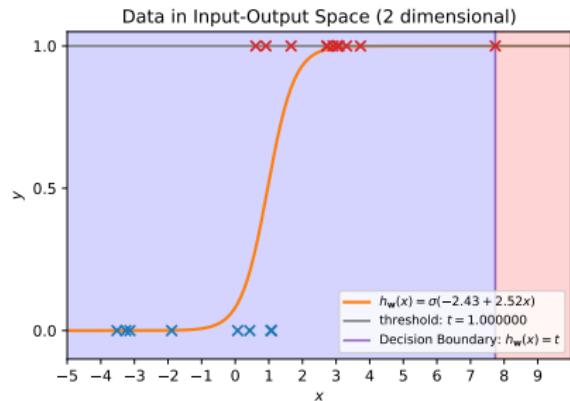
Precision & Recall vs. Threshold

A soft threshold classifier is really a **family** of classifiers defined by the choice of cutoff value t . How does each classifier in the family do?



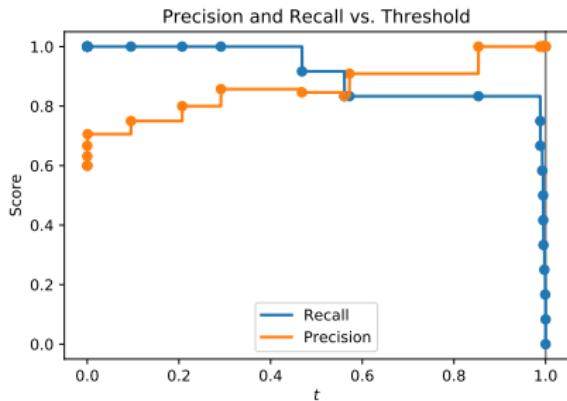
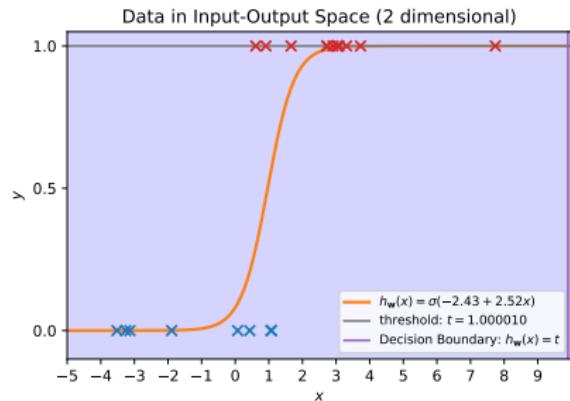
Precision & Recall vs. Threshold

A soft threshold classifier is really a **family** of classifiers defined by the choice of cutoff value t . How does each classifier in the family do?



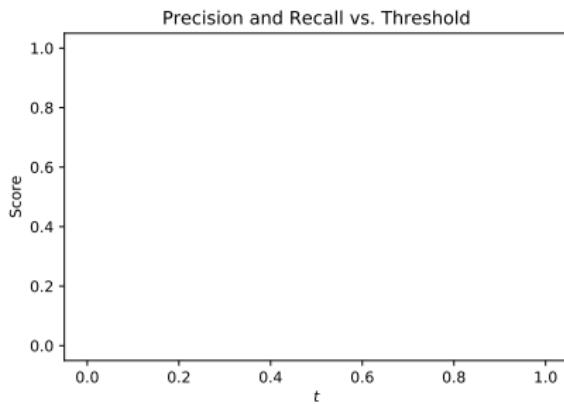
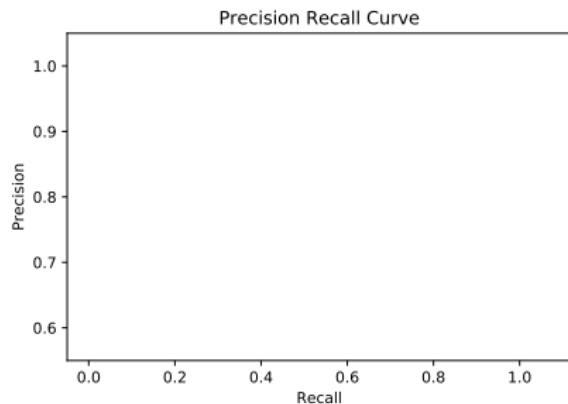
Precision & Recall vs. Threshold

A soft threshold classifier is really a **family** of classifiers defined by the choice of cutoff value t . How does each classifier in the family do?



Precision Recall Curve

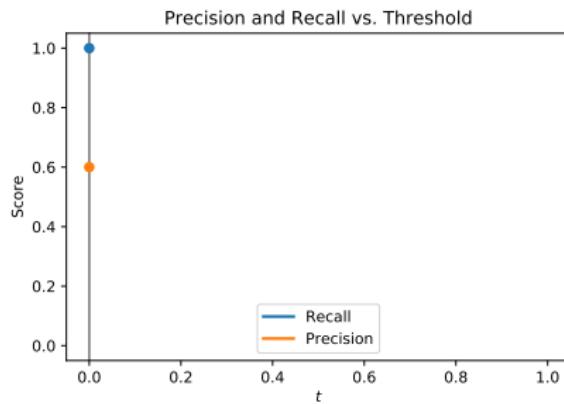
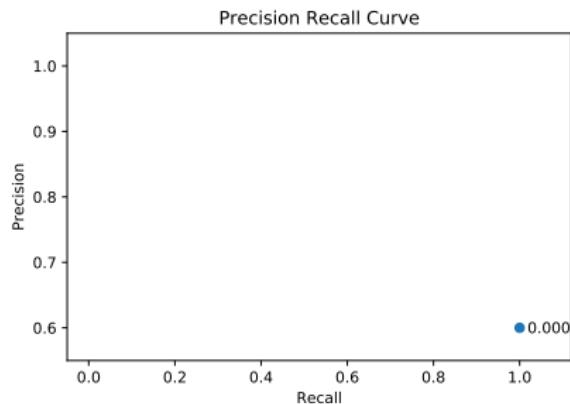
For any cutoff value t , we can also plot the associated (precision, recall) pairs as points in two dimensional space:



- Precision-recall curve is parameterized by threshold
- Area under the precision-recall curve gives us a metric for a classifier that is **independent** of threshold value

Precision Recall Curve

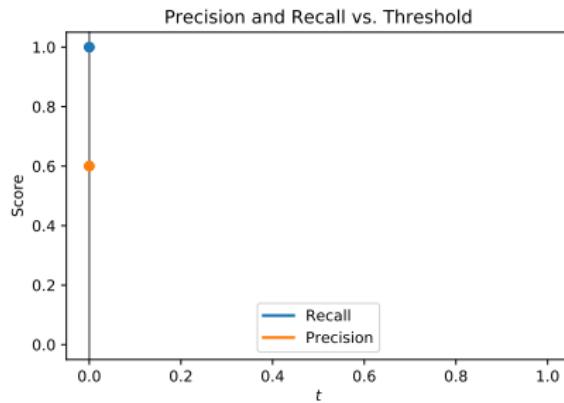
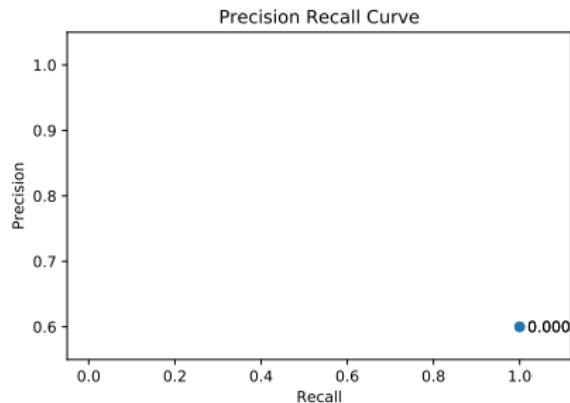
For any cutoff value t , we can also plot the associated (precision, recall) pairs as points in two dimensional space:



- Precision-recall curve is parameterized by threshold
- Area under the precision-recall curve gives us a metric for a classifier that is **independent** of threshold value

Precision Recall Curve

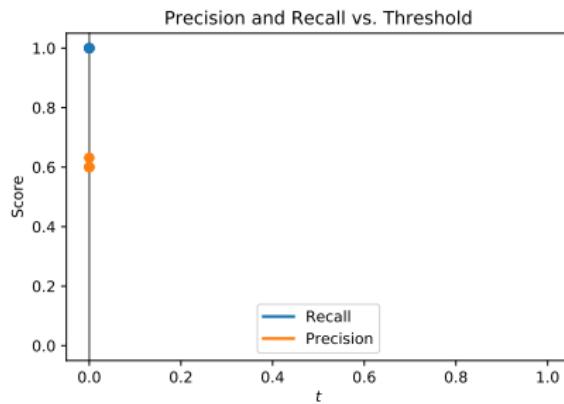
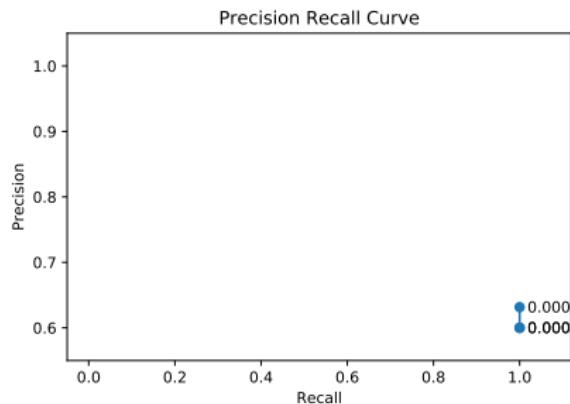
For any cutoff value t , we can also plot the associated (precision, recall) pairs as points in two dimensional space:



- Precision-recall curve is parameterized by threshold
- Area under the precision-recall curve gives us a metric for a classifier that is **independent** of threshold value

Precision Recall Curve

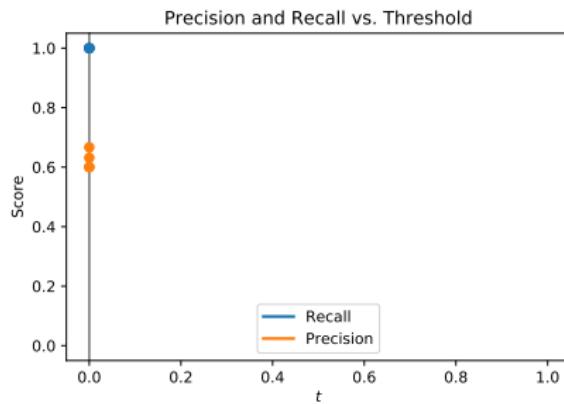
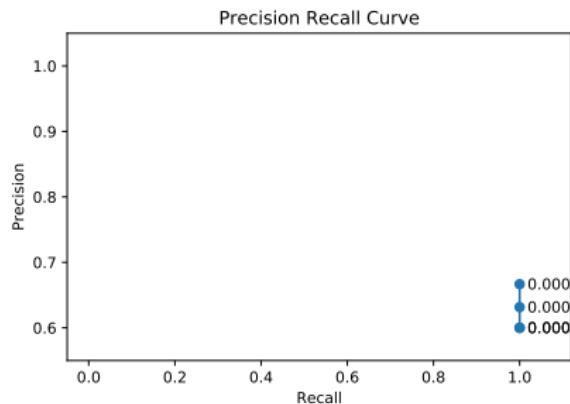
For any cutoff value t , we can also plot the associated (precision, recall) pairs as points in two dimensional space:



- Precision-recall curve is parameterized by threshold
- Area under the precision-recall curve gives us a metric for a classifier that is **independent** of threshold value

Precision Recall Curve

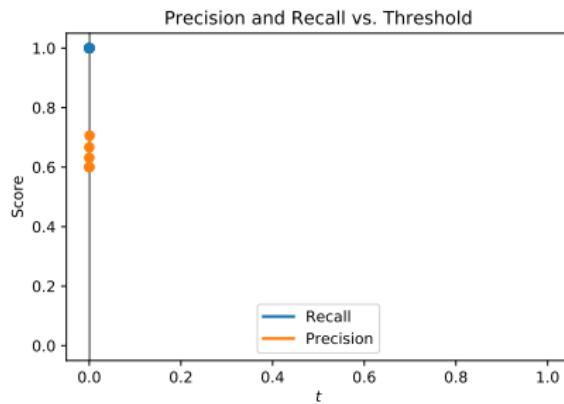
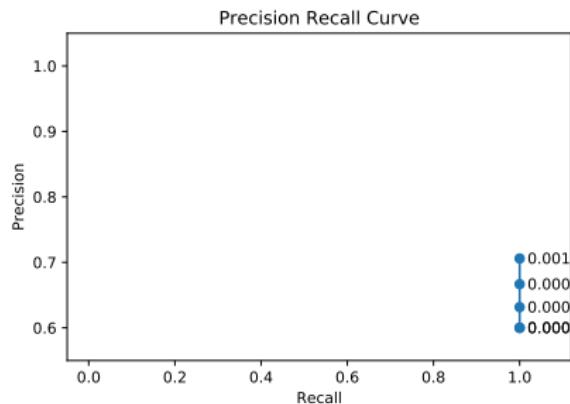
For any cutoff value t , we can also plot the associated (precision, recall) pairs as points in two dimensional space:



- Precision-recall curve is parameterized by threshold
- Area under the precision-recall curve gives us a metric for a classifier that is **independent** of threshold value

Precision Recall Curve

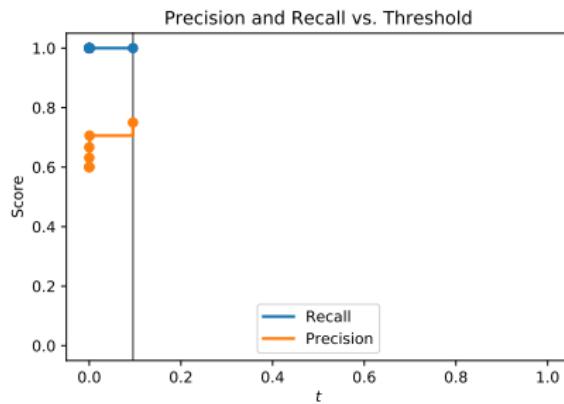
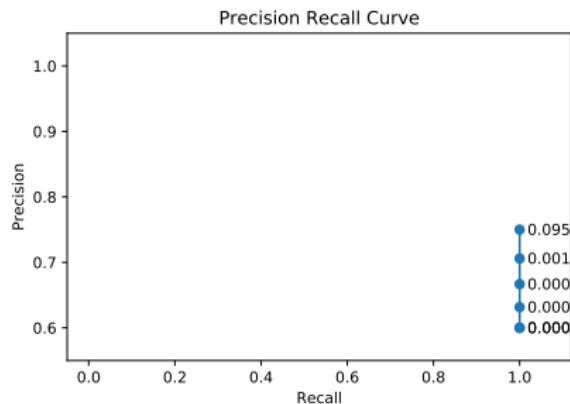
For any cutoff value t , we can also plot the associated (precision, recall) pairs as points in two dimensional space:



- Precision-recall curve is parameterized by threshold
- Area under the precision-recall curve gives us a metric for a classifier that is **independent** of threshold value

Precision Recall Curve

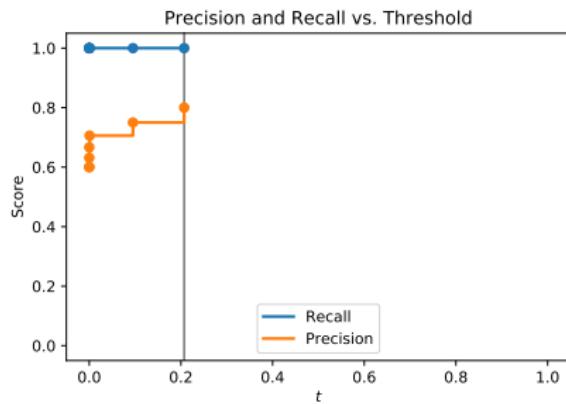
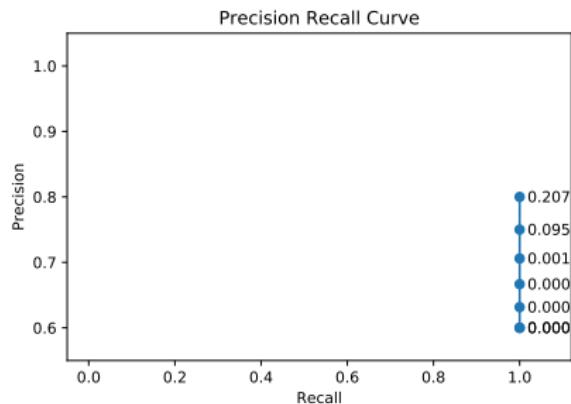
For any cutoff value t , we can also plot the associated (precision, recall) pairs as points in two dimensional space:



- Precision-recall curve is parameterized by threshold
- Area under the precision-recall curve gives us a metric for a classifier that is **independent** of threshold value

Precision Recall Curve

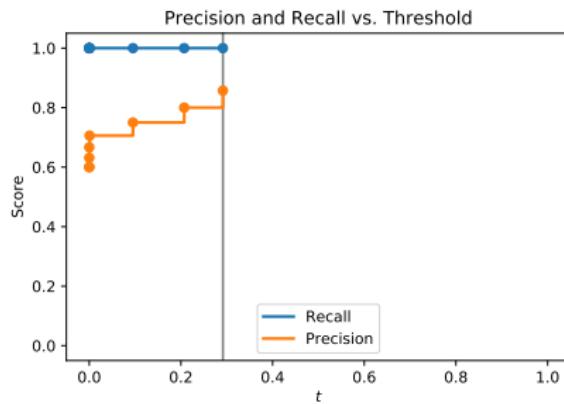
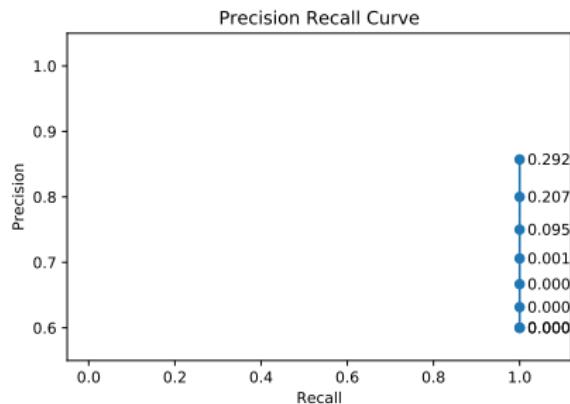
For any cutoff value t , we can also plot the associated (precision, recall) pairs as points in two dimensional space:



- Precision-recall curve is parameterized by threshold
- Area under the precision-recall curve gives us a metric for a classifier that is **independent** of threshold value

Precision Recall Curve

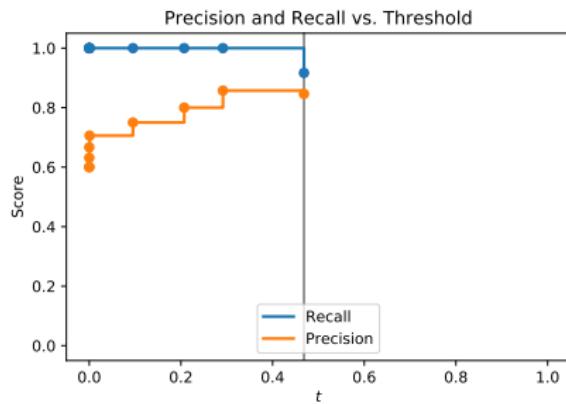
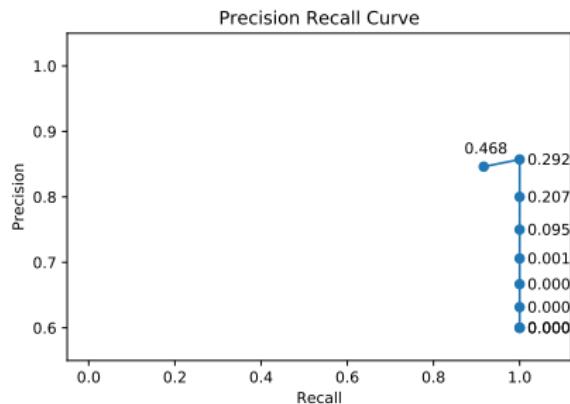
For any cutoff value t , we can also plot the associated (precision, recall) pairs as points in two dimensional space:



- Precision-recall curve is parameterized by threshold
- Area under the precision-recall curve gives us a metric for a classifier that is **independent** of threshold value

Precision Recall Curve

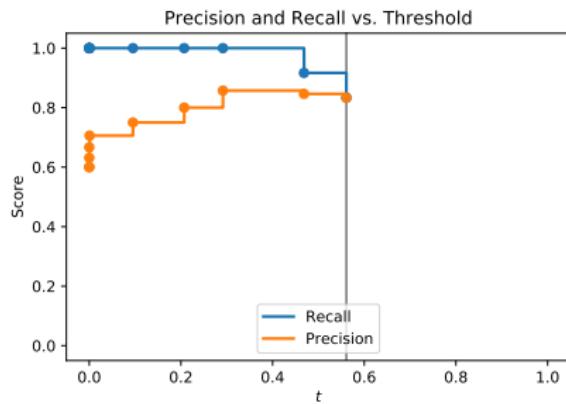
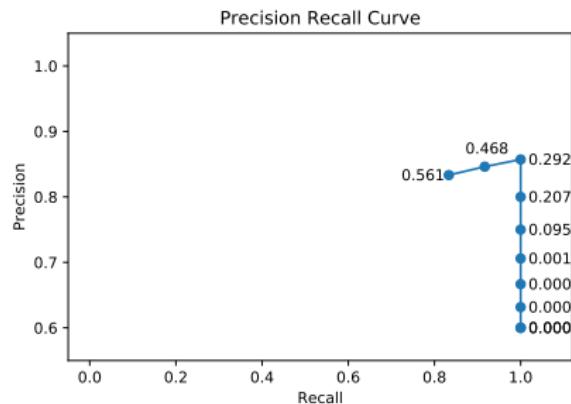
For any cutoff value t , we can also plot the associated (precision, recall) pairs as points in two dimensional space:



- Precision-recall curve is parameterized by threshold
- Area under the precision-recall curve gives us a metric for a classifier that is **independent** of threshold value

Precision Recall Curve

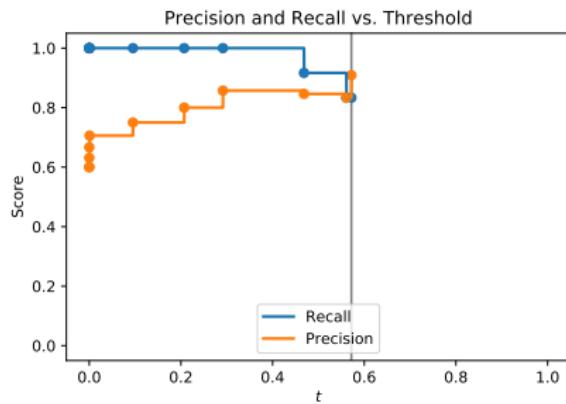
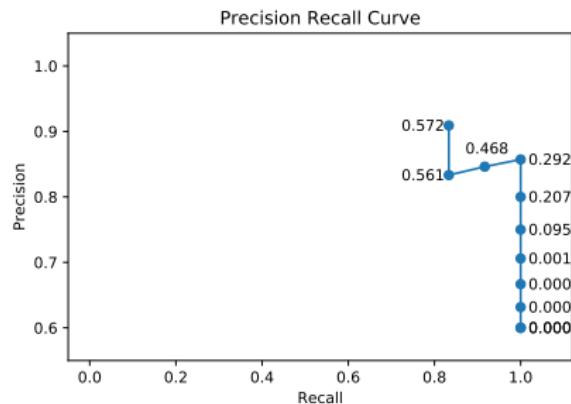
For any cutoff value t , we can also plot the associated (precision, recall) pairs as points in two dimensional space:



- Precision-recall curve is parameterized by threshold
- Area under the precision-recall curve gives us a metric for a classifier that is **independent** of threshold value

Precision Recall Curve

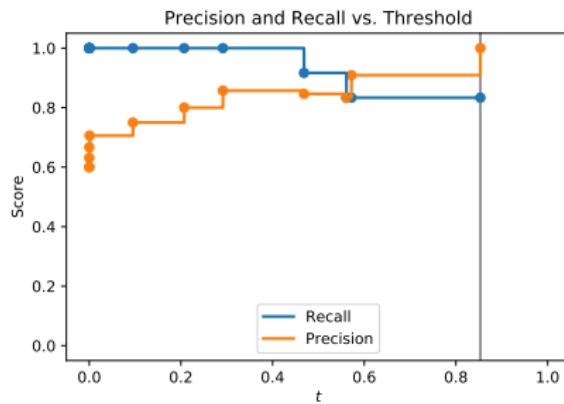
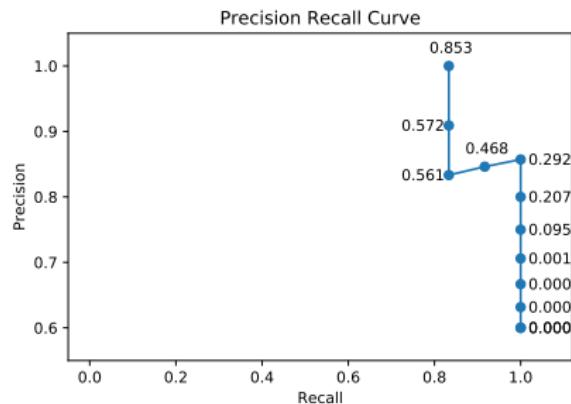
For any cutoff value t , we can also plot the associated (precision, recall) pairs as points in two dimensional space:



- Precision-recall curve is parameterized by threshold
- Area under the precision-recall curve gives us a metric for a classifier that is **independent** of threshold value

Precision Recall Curve

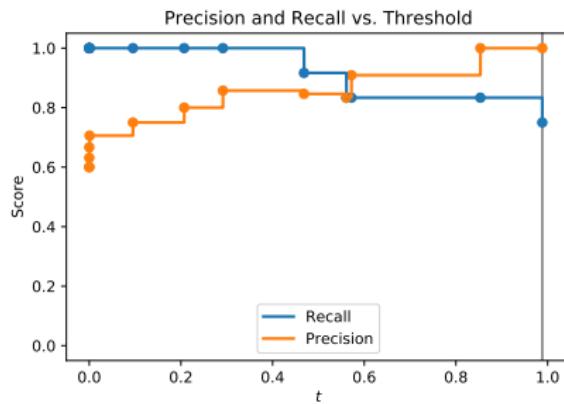
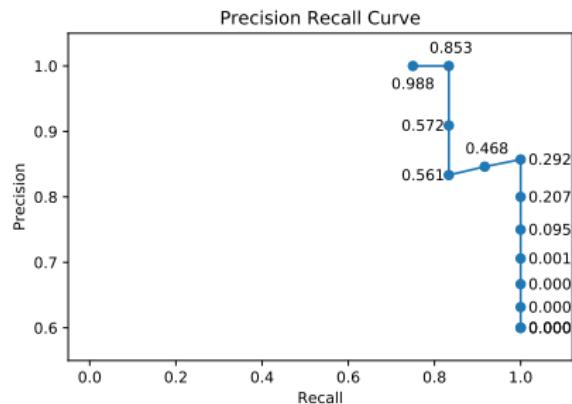
For any cutoff value t , we can also plot the associated (precision, recall) pairs as points in two dimensional space:



- Precision-recall curve is parameterized by threshold
- Area under the precision-recall curve gives us a metric for a classifier that is **independent** of threshold value

Precision Recall Curve

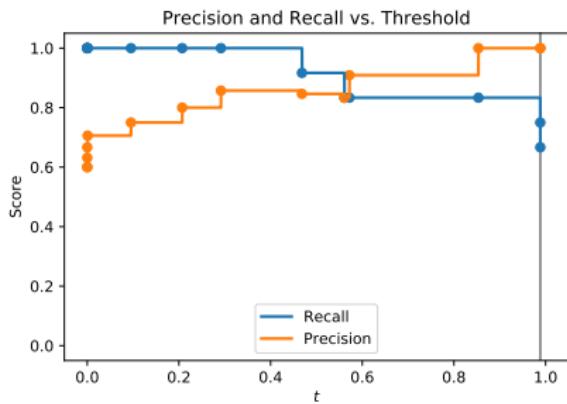
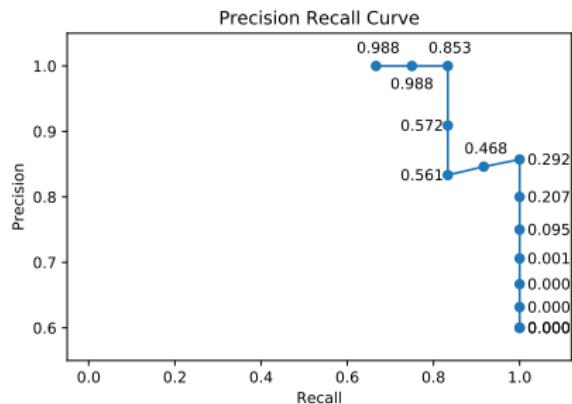
For any cutoff value t , we can also plot the associated (precision, recall) pairs as points in two dimensional space:



- Precision-recall curve is parameterized by threshold
- Area under the precision-recall curve gives us a metric for a classifier that is **independent** of threshold value

Precision Recall Curve

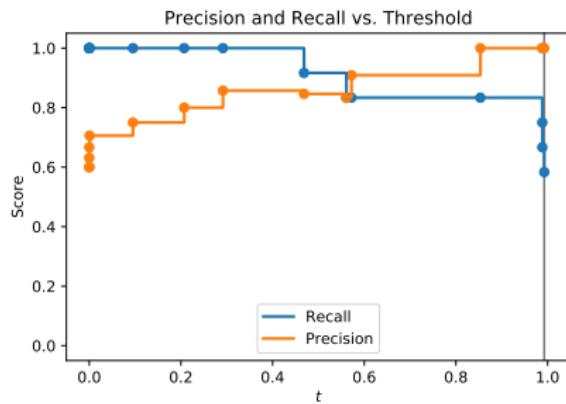
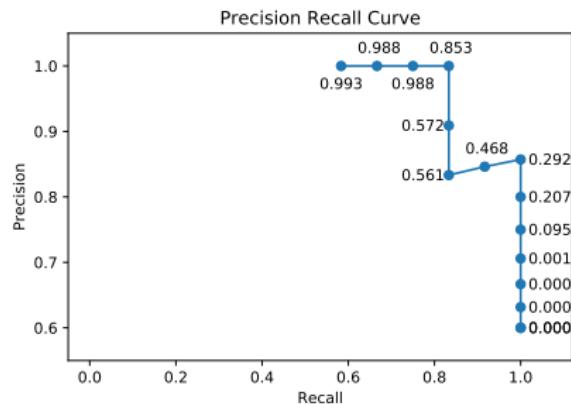
For any cutoff value t , we can also plot the associated (precision, recall) pairs as points in two dimensional space:



- Precision-recall curve is parameterized by threshold
- Area under the precision-recall curve gives us a metric for a classifier that is **independent** of threshold value

Precision Recall Curve

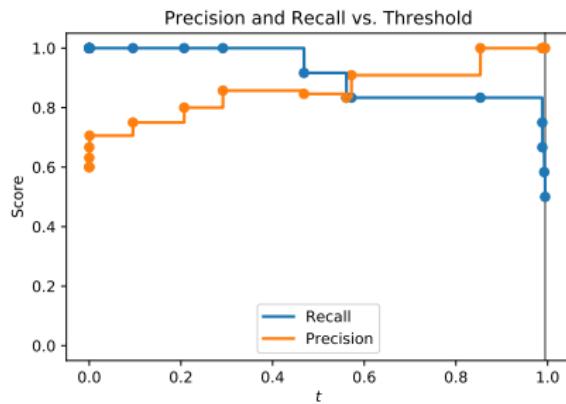
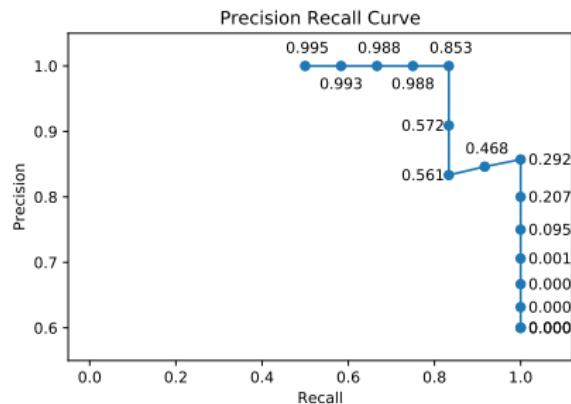
For any cutoff value t , we can also plot the associated (precision, recall) pairs as points in two dimensional space:



- Precision-recall curve is parameterized by threshold
- Area under the precision-recall curve gives us a metric for a classifier that is **independent** of threshold value

Precision Recall Curve

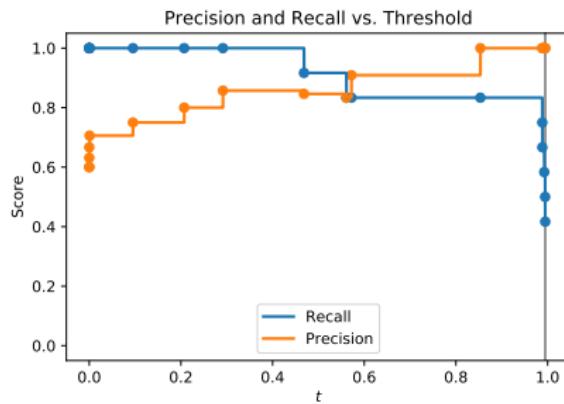
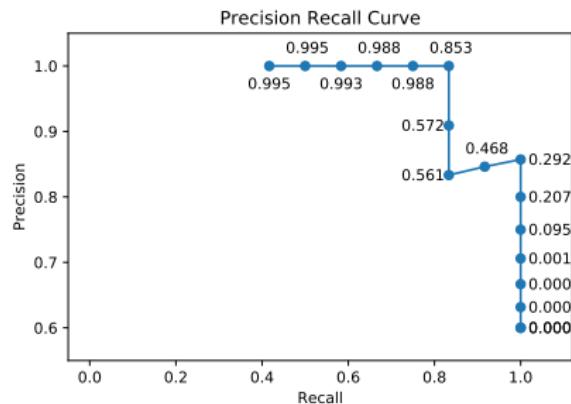
For any cutoff value t , we can also plot the associated (precision, recall) pairs as points in two dimensional space:



- Precision-recall curve is parameterized by threshold
- Area under the precision-recall curve gives us a metric for a classifier that is **independent** of threshold value

Precision Recall Curve

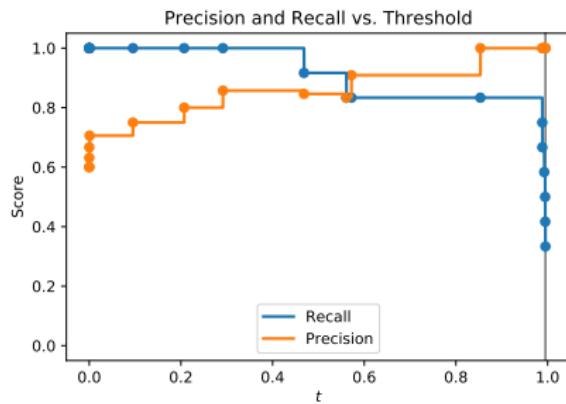
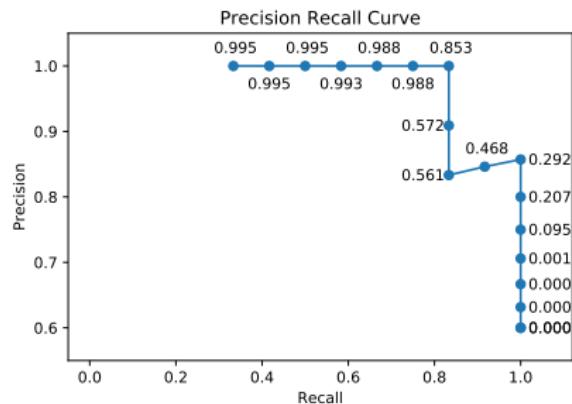
For any cutoff value t , we can also plot the associated (precision, recall) pairs as points in two dimensional space:



- Precision-recall curve is parameterized by threshold
- Area under the precision-recall curve gives us a metric for a classifier that is **independent** of threshold value

Precision Recall Curve

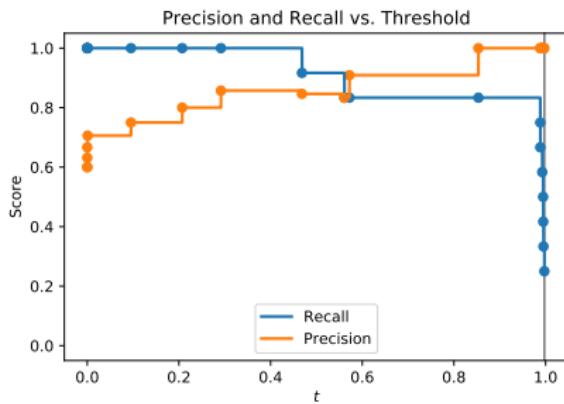
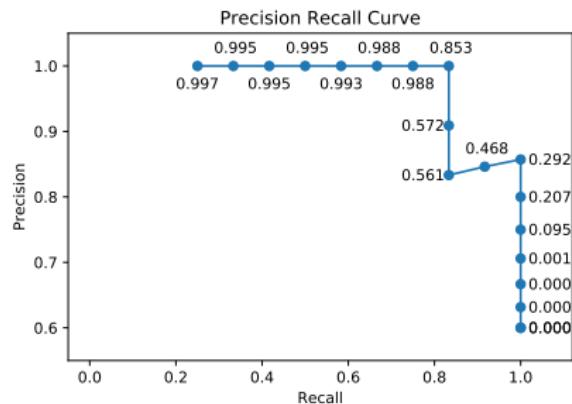
For any cutoff value t , we can also plot the associated (precision, recall) pairs as points in two dimensional space:



- Precision-recall curve is parameterized by threshold
- Area under the precision-recall curve gives us a metric for a classifier that is **independent** of threshold value

Precision Recall Curve

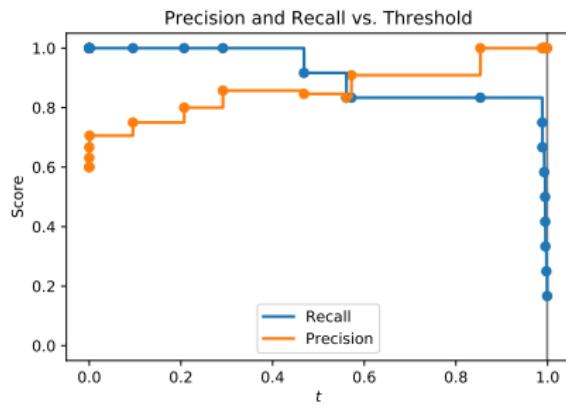
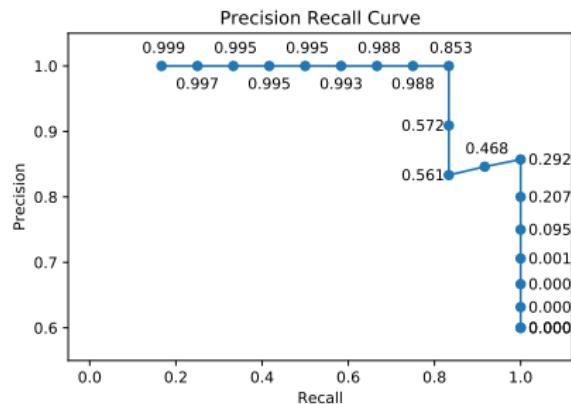
For any cutoff value t , we can also plot the associated (precision, recall) pairs as points in two dimensional space:



- Precision-recall curve is parameterized by threshold
- Area under the precision-recall curve gives us a metric for a classifier that is **independent** of threshold value

Precision Recall Curve

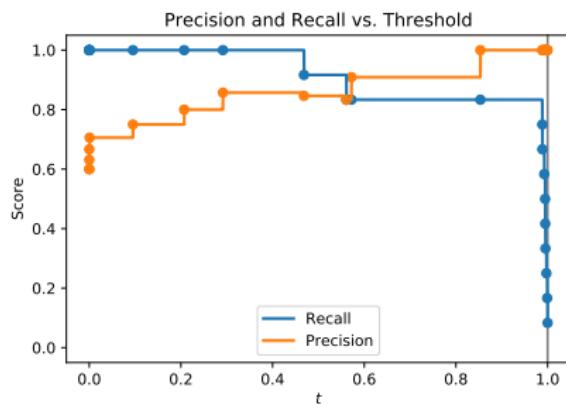
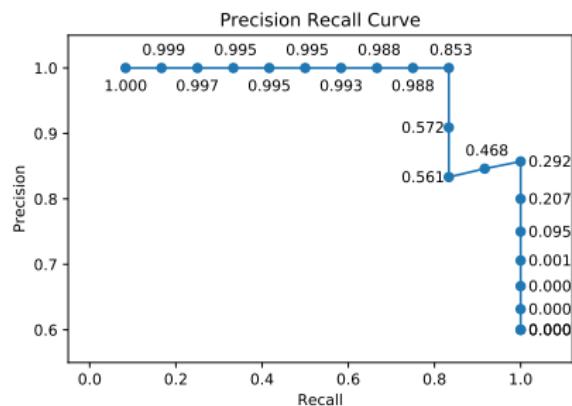
For any cutoff value t , we can also plot the associated (precision, recall) pairs as points in two dimensional space:



- Precision-recall curve is parameterized by threshold
- Area under the precision-recall curve gives us a metric for a classifier that is **independent** of threshold value

Precision Recall Curve

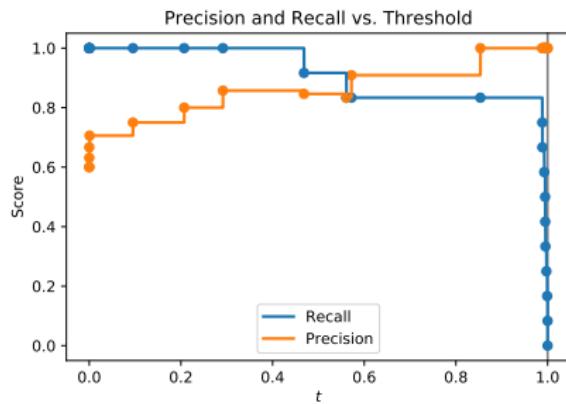
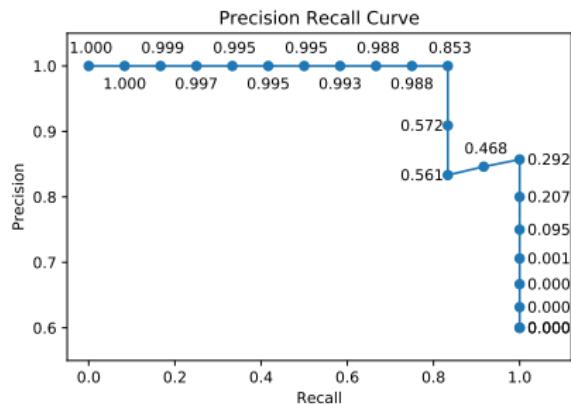
For any cutoff value t , we can also plot the associated (precision, recall) pairs as points in two dimensional space:



- Precision-recall curve is parameterized by threshold
- Area under the precision-recall curve gives us a metric for a classifier that is **independent** of threshold value

Precision Recall Curve

For any cutoff value t , we can also plot the associated (precision, recall) pairs as points in two dimensional space:



- Precision-recall curve is parameterized by threshold
- Area under the precision-recall curve gives us a metric for a classifier that is **independent** of threshold value

Other Metrics for Evaluating Classifiers

The **F-score** (F1-score) balances precision and recall in a **single** number:

$$F = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

False Positive Rate (FPR):

$$\mathbf{FPR} = \frac{\text{FP}}{\text{FP} + \text{TN}}$$

True Positive Rate (TPR):

$$\mathbf{TPR} = \frac{\text{TP}}{\text{TP} + \text{FN}} = \mathbf{recall}$$

		y_i	
		1	0
\hat{y}_i	1	TP	FP
	0	FN	TN

ROC Curves

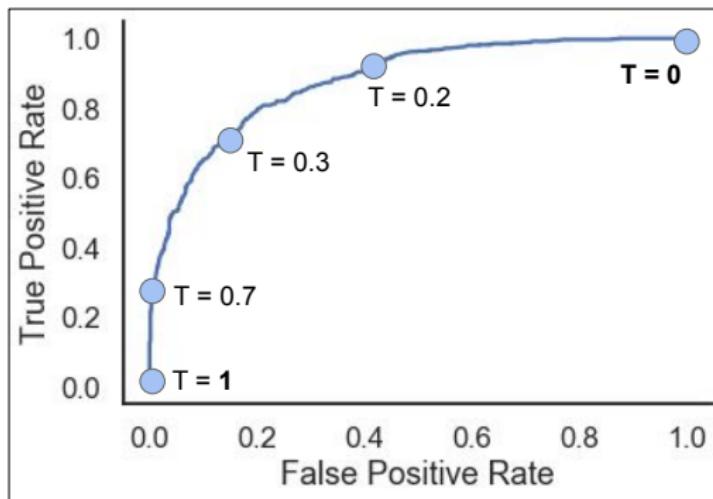
Decreasing the decision threshold:

- Increases TPR (good).
- Increases FPR (bad).

A “ROC curve” shows this tradeoff.

- X-axis: False positive rate.
- Y-axis: True positive rate.

What are the Ts in the bottom left and top right?



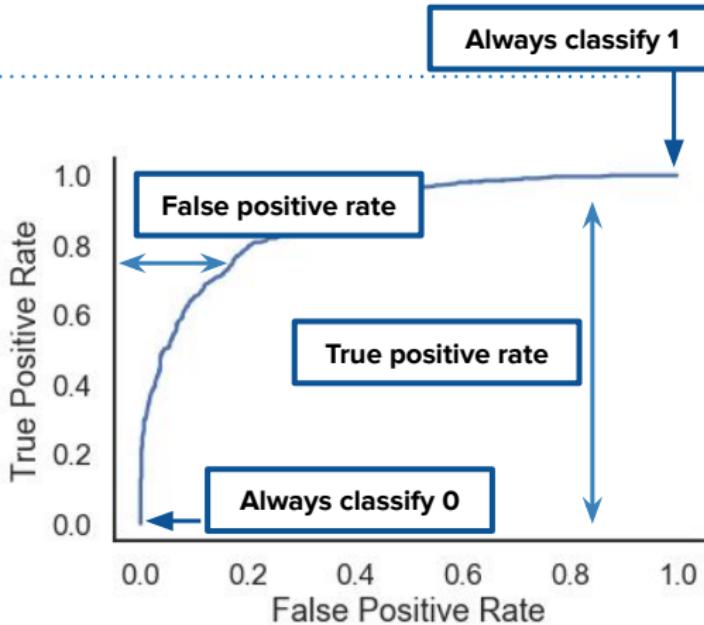
ROC Curves

Decreasing the decision threshold:

- Increases TPR (good).
- Increases FPR (bad).

A “ROC curve” shows this tradeoff.

- X-axis: False positive rate.
- Y-axis: True positive rate.

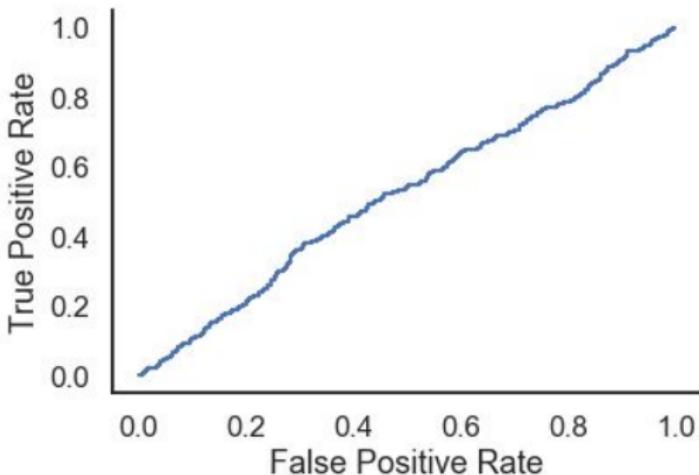


What Good Are ROC Curves?

ROC Curves provide a visual picture of the underlying quality of the regression model.

Example: [Random Predictor](#).

- Assigns random probability between 0 and 1 to any prediction.
- Resulting ROC curve is a diagonal line.

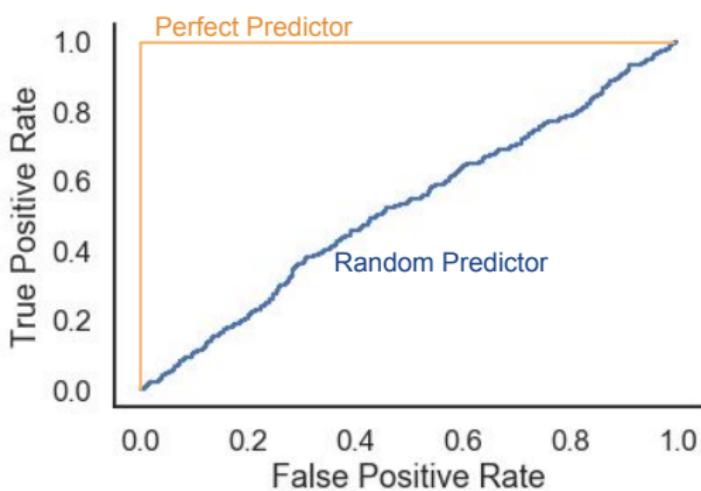


What Good Are ROC Curves?

ROC Curves provide a visual picture of the underlying quality of the regression model.

Example: Perfect Predictor.

- Gives probability 1 to class 1 and probability 0 to class 0.
- Resulting ROC curve is a sharp curve.



The better a model, the more it will resemble the perfect predictor.

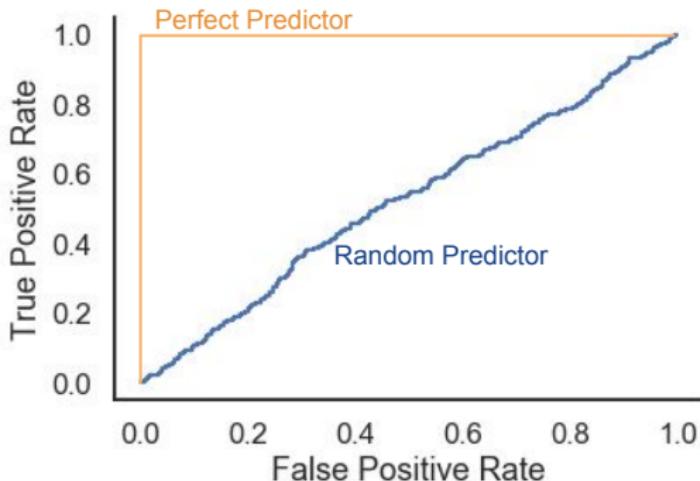
Area under ROC Curve

Can also compute the area under the ROC curve (a.k.a. **AUC**).

- **Perfect Predictor**: 1
- **Random Predictor**: 0.5
- Your predictor: Somewhere in $[0.5, 1]$.

The **AUC** provides a dimensionless quantity that characterizes our underlying regression model.

- Dimensionless means it has no units (unlike MSE, mean cross entropy loss, etc.)



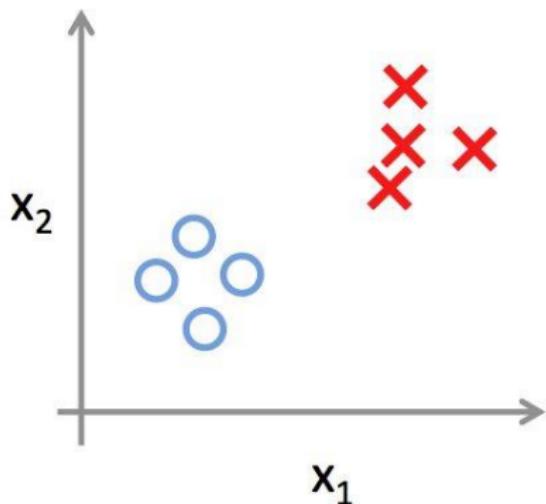
It is possible to build a predictor with $\text{AUC} < 0.5$, but that means it does worse than just randomly guessing.

Lecture 05-4c: Additional Notes on Classification

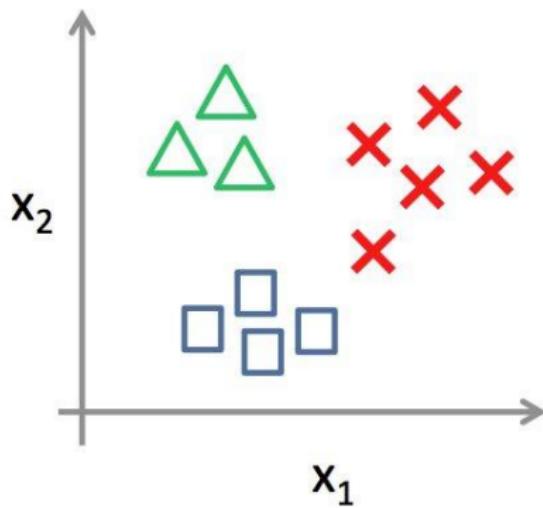
Multi-Class Classification

So far we have seen classification when the output variable is binary.
What about if y is discrete with **more than two** categories (classes)?

Binary classification:



Multi-class classification:



One Versus Rest Classifiers

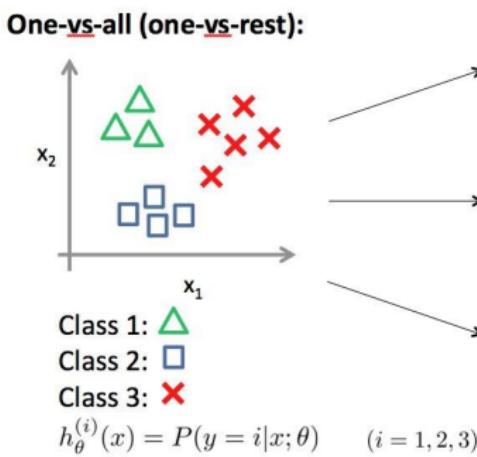
For multi-class classification with K classes, we can build K binary classifiers, one for each class:

$$h_{\mathbf{w}^{(1)}}(\mathbf{x}) = P(Y = 1 | \mathbf{X} = \mathbf{x})$$

$$h_{\mathbf{w}^{(2)}}(\mathbf{x}) = P(Y = 2 | \mathbf{X} = \mathbf{x})$$

⋮

$$h_{\mathbf{w}^{(K)}}(\mathbf{x}) = P(Y = K | \mathbf{X} = \mathbf{x})$$



Classify points according to the highest-probability class:

$$\hat{y}_i = \arg \max_k \{h_{\mathbf{w}^{(k)}}(\mathbf{x}_i)\}$$

Evaluating Multi-Class Classifiers

Confusion matrix for binary classification:

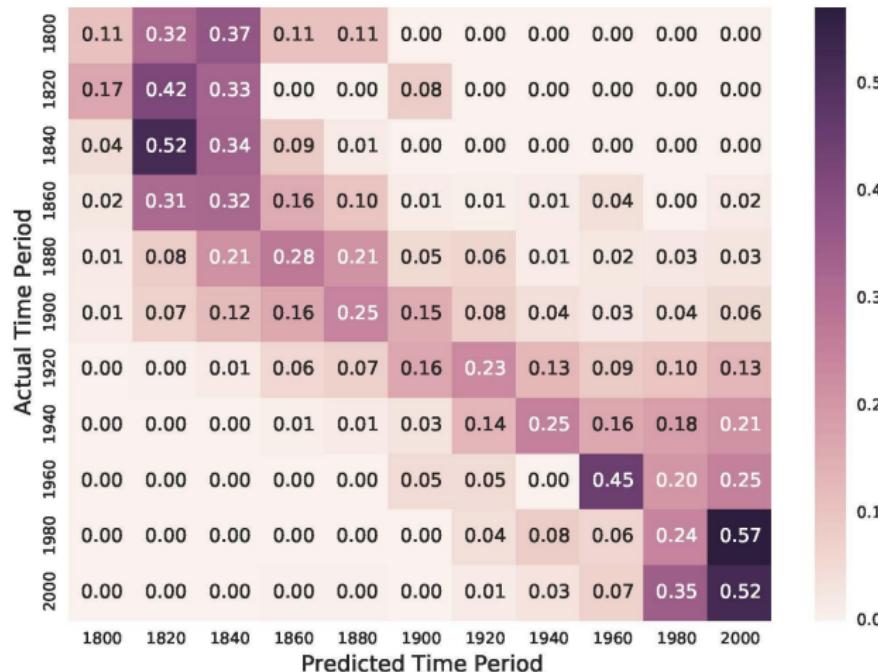
		Truth (y_i)	
		1	0
Prediction (\hat{y}_i)	1	TP	FP
	0	FN	TN

Confusion matrix for multi-class classification:

		Truth (y_i)				
		1	2	3	...	K
Prediction (\hat{y}_i)	1	✓	X	X	...	X
	2	X	✓	X	...	X
	3	X	X	✓	...	X
	⋮	⋮	⋮	⋮	⋮	⋮
	K	X	X	X	...	✓

Example:

Classifying Documents by Time Period

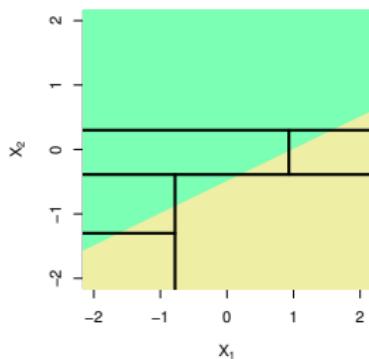
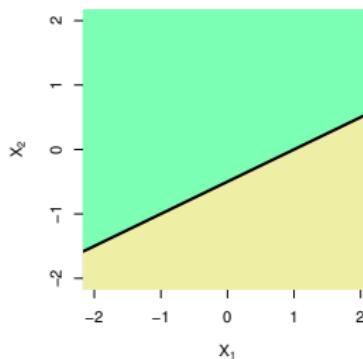


- More classes leads to a harder problem!
- Scoring with hit/miss only is excessively harsh
- A top- k success rate gives credit if the correct label would have been one of the model's first k guesses

Decision Trees vs. Linear Models

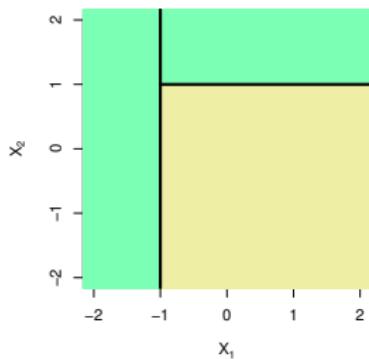
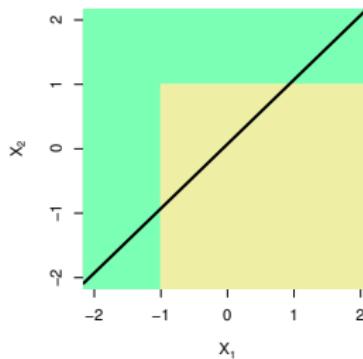
Scenario 1 (top row):

- Ideal for logistic regression
- Problematic for decision trees with axis-aligned hyperplanes



Scenario 2 (bottom row):

- Problematic for logistic regression
- Easy for decision trees



Taken from "An Introduction to Statistical Learning, with applications in R" (Springer, 2013) with permission from the authors:
G. James, D. Witten, T. Hastie and R. Tibshirani.

CS 457/557: Machine Learning

Lecture 06-1: k -Nearest Neighbors

Lecture 06-1a: Nearest Neighbor Classification

Types of Learning Methods: Parametric Methods

Definition

A learning method that summarizes the data with a set of parameters of *fixed size* (independent of the size of the training set) is called a **parametric method**.

Both linear and logistic regression are **parametric methods**:

- The choice of hypothesis space determines the number of model parameters (weights)
- Once we learn the optimal weights w^* , we can throw out the training data!

Parametric methods often make strong assumptions about the form of the function f that maps inputs to outputs.

Types of Learning Methods: Nonparametric Methods

Definition

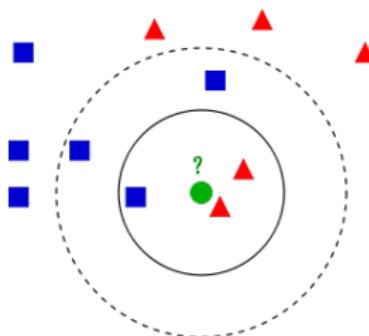
A **nonparametric method** is a method that cannot be characterized by a bounded set of parameters.

Decision trees are one type of **nonparametric method**:

- The hypothesis space is large (trees can represent any binary-valued function defined on the input space)
- The size of the learned tree depends on the training data itself

Nonparametric methods generally don't make assumptions about the form of the function f , and instead try to "let the data speak for itself".

A Nonparametric Classification Technique



Intuition: A new point should “look like” the point that is closest to it!

Definition

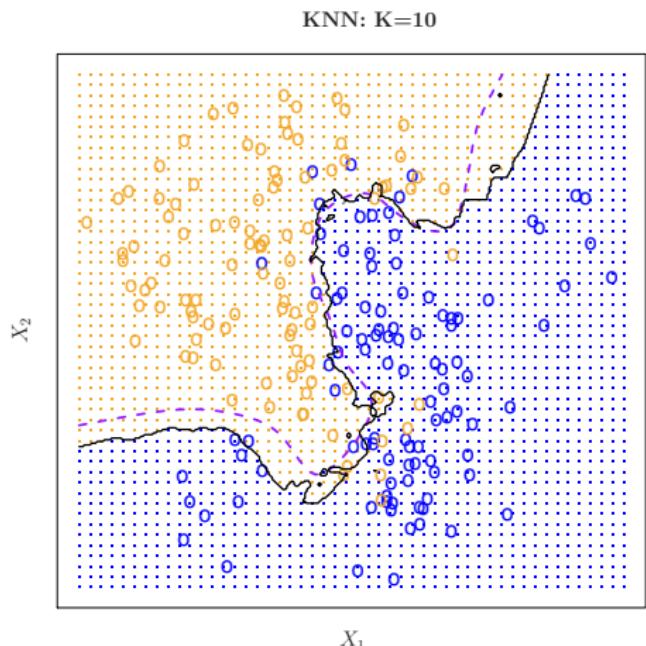
Given a training set $\{(\mathbf{x}_i, y_i)\}_{i=1}^N$, **nearest neighbor (NN) classification** makes predictions for a query point \mathbf{x}_q by:

- ① Find $i^* = \arg \min_{i=1}^N d(\mathbf{x}_q, \mathbf{x}_i)$, where $d(\mathbf{u}, \mathbf{v})$ is a distance function
- ② Predict $\hat{y}_q = y_{i^*}$

Nearest Neighbor Classification

Benefits of NN classification:

- Simplicity: all we need to do is specify a **distance metric!**
- Interpretability: easy to know why predictions are made
- Non-linearity: can produce complex decision boundaries



Taken from "An Introduction to Statistical Learning, with applications in R" (Springer, 2013)
with permission from the authors: G. James, D. Witten, T. Hastie and R. Tibshirani.

Lecture 06-1b: Distance Metrics

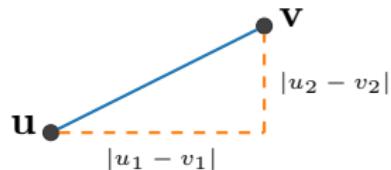
Distance Metrics

Any distance metric $d : \mathbb{R}^p \times \mathbb{R}^p \rightarrow \mathbb{R}$ is expected to satisfy:

- Nonnegativity: $d(\mathbf{u}, \mathbf{v}) \geq 0$ for all \mathbf{u}, \mathbf{v}
- Identity: $d(\mathbf{u}, \mathbf{v}) = 0$ if and only if $\mathbf{u} = \mathbf{v}$
- Symmetry: $d(\mathbf{u}, \mathbf{v}) = d(\mathbf{v}, \mathbf{u})$
- Triangle Inequality: $d(\mathbf{u}, \mathbf{v}) \leq d(\mathbf{u}, \mathbf{z}) + d(\mathbf{z}, \mathbf{v})$

Euclidean distance:

$$L^2(\mathbf{u}, \mathbf{v}) = \sqrt{\sum_{j=1}^p |u_j - v_j|^2}$$



Manhattan distance:

$$L^1(\mathbf{u}, \mathbf{v}) = \sum_{j=1}^p |u_j - v_j|$$



L^k -Norms

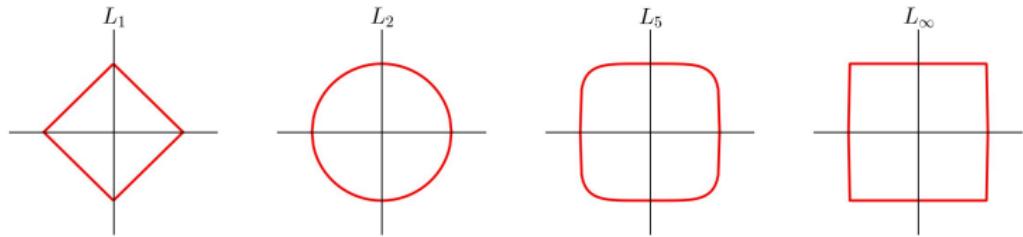
Some common options for $d(\mathbf{u}, \mathbf{v})$:

$$L^1 \text{ norm: } \sum_{j=1}^p |u_j - v_j|$$

$$L^2 \text{ norm: } \sqrt{\sum_{j=1}^p |u_j - v_j|^2}$$

$$L^k \text{ norm: } \left(\sum_{j=1}^p |u_j - v_j|^k \right)^{1/k}$$

$$L^\infty \text{ norm: } \max_{j=1}^p \{|u_j - v_j|\}$$



Moves from total dimensional difference to largest dimensional difference

Important Considerations for Distances

Which distance metric should be used?

- Euclidean distance is commonly used when attributes all measure similar properties (e.g., height, width, depth).
- Manhattan distance is often better for heterogeneous attributes (e.g., age, weight, height, education level)

Values should be comparable **across** attributes, too!

- E.g., using age measured in years versus age measured in months can change the “distance” between two points!

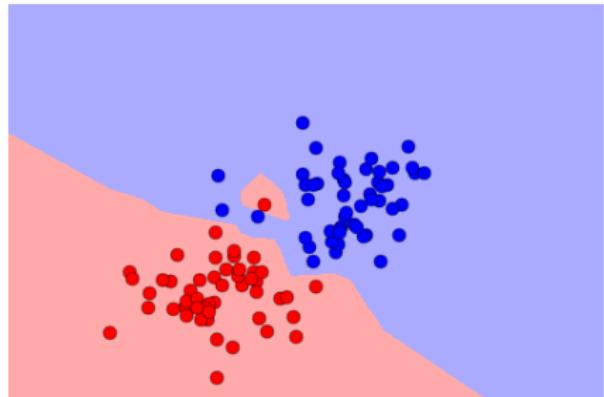
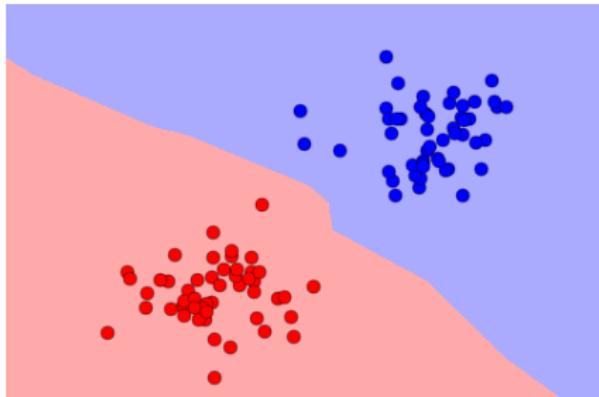
It is **important** to standardize (e.g., using Z -scores) or apply some other type of normalization across dimensions!

(Note that **irrelevant features** are actually harmful here!)

Lots of other distance metrics exist, too; e.g.: Cosine similarity
Mahalanobis distance

Lecture 06-1c: k -Nearest Neighbors

NN in Action



- NN classifier produces a nonlinear decision boundary in feature space
- May have “islands” depending on training data

To fix this, we can include more neighbors!

k -Nearest Neighbors for Classification

Definition

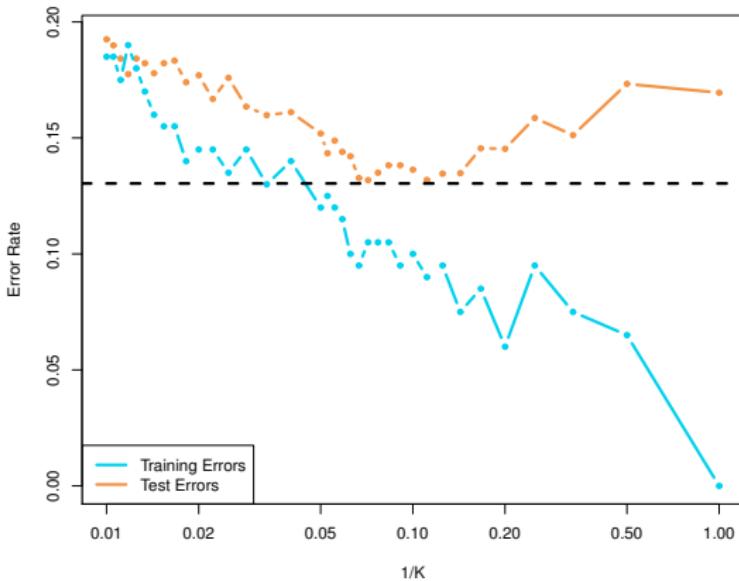
The **k -nearest neighbors** (k -NN, kNN, KNN) classifier predicts the class label for a query point \mathbf{x}_q by finding the k closest training points and returning the majority label.

Here k represents a model hyperparameter that we need to specify.
Typically we use odd k for binary classification.

- With small k : overfitting (small bias, high variance)
- With large k : underfitting (high bias, small variance)
($k = N$ gives majority classifier)

We can try to find the **right** k using cross validation!

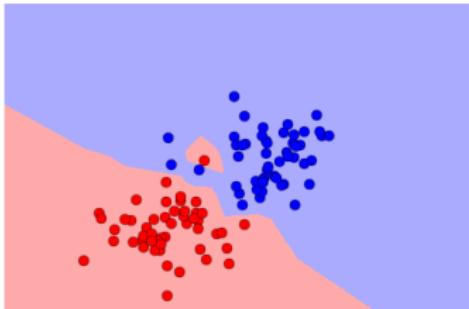
k -NN Training Error and Test Error



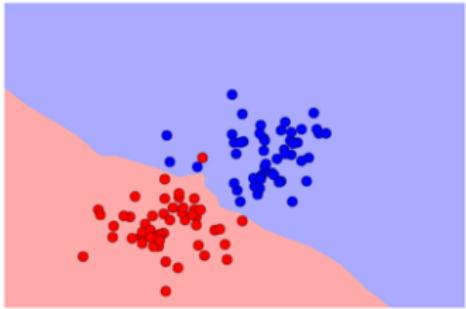
Taken from "An Introduction to Statistical Learning, with applications in R" (Springer, 2013)
with permission from the authors: G. James, D. Witten, T. Hastie and R. Tibshirani.

The More the Merrier

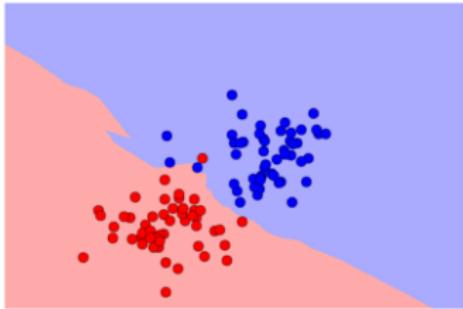
One neighbor:



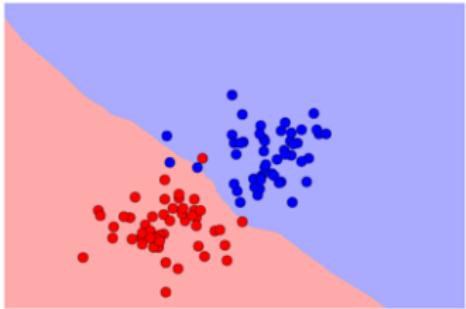
Three neighbors:



Five neighbors:



Nine neighbors:



k-Nearest Neighbors for Regression

We can also use *k*-NN for regression problems!

For query point \mathbf{x}_q , *k*-NN regression predicts output \hat{y}_q as

$$\hat{y}_q = \frac{1}{k} \sum_{i \in \mathcal{N}_k(\mathbf{x}_q)} y_i$$

where $\mathcal{N}_k(\mathbf{x}_q)$ is the ***k*-neighborhood** of \mathbf{x}_q that contains (indices for) the closest points in the training set.

- Could also weight the *k* neighbors by proximity so that closer neighbors have more influence over predicted label (e.g., weighting using inverse of distance)

CS 457/557: Machine Learning

Lecture 06-2: Practical Considerations with k -NN

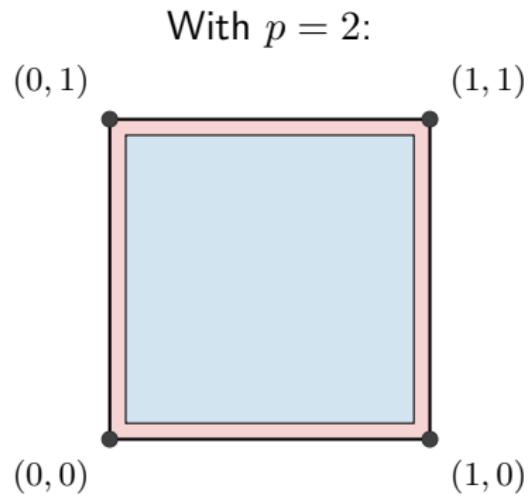
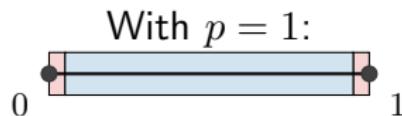
Lecture 06-2a: The Curse of Dimensionality

Space is Big

What happens in higher-dimensional spaces?

Let's look at points generated **uniformly** in the unit hypercube, $[0, 1]^p$.

We will classify a point as an **outlier** if its value for **any** attribute occurs within the range $[0, 0.05]$ or $[0.95, 1]$.



How Many Outliers Are There?

We can determine the proportion of outliers by finding the probability that any given point is an **inlier**!

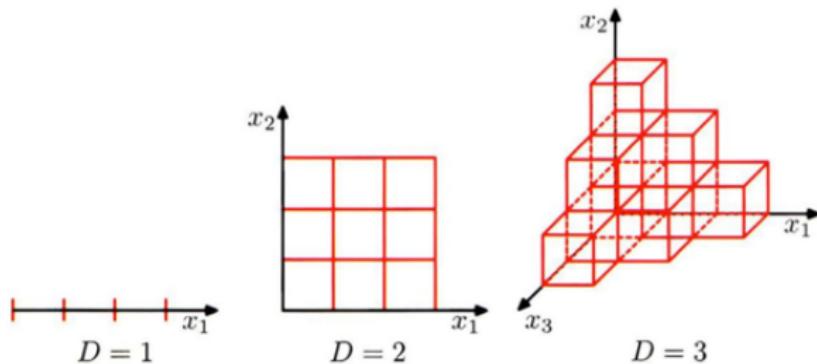
p	$P(\text{point is an inlier})$	% of points that are outliers
1	0.9	10%
2	$0.9 \cdot 0.9 = 0.81$	19%
3	$0.9^3 = 0.729$	$\approx 27\%$
4	$0.9^4 \approx 0.656$	$\approx 34\%$
10	$0.9^{10} \approx 0.349$	$\approx 65\%$
20	$0.9^{20} \approx 0.122$	$\approx 88\%$

In higher dimensions, **most points** live near the exterior of the hypercube!

The Curse of Dimensionality

Definition

The **curse of dimensionality** refers to odd behavior that occurs in higher dimensions that is not observed in lower dimensions.



Implications

The curse of dimensionality has implications for using k -NN with a large number of attributes p :

- Volume grows exponentially with p ; space becomes very sparse
- **All** points in the training set are **far away**
- Distances between points become more and more similar

Because of this, k -NN tends to do worse than linear/logistic regression in higher dimensions. However, things aren't quite as bleak for k -NN as the preceding analysis suggests:

- In practice, data is **not** uniformly distributed, and so the previous analysis is a bit extreme.
- More data helps: with p binary attributes, you need $N \geq 2^p$ to have training data containing each possible combination of attribute values. This makes computation **more costly**, though.

Lecture 06-2b: k -NN Efficiency and k -d Trees

Efficiency Comparison

For training set $\{(\mathbf{x}_i, y_i)\}_{i=1}^N$ with p attributes:

Logistic regression for classification:

- One-time cost to fit the model
- Prediction for \mathbf{x}_q : $O(p)$ (just need to compute $h_{\mathbf{w}}(\mathbf{x}_q) = \sigma(\mathbf{w} \cdot \mathbf{x}_q)$)
- Size of model: $O(p)$ (just the weight vector \mathbf{w})

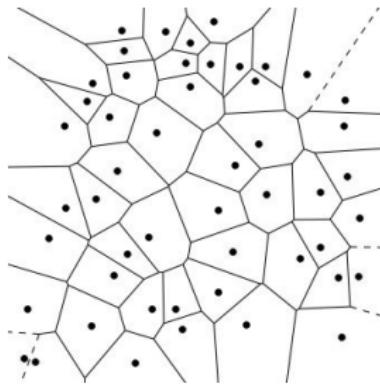
NN classification:

- **No** up-front costs for model fitting
- Prediction for \mathbf{x}_q : $O(Np)$
 - Computing the distance between two vectors in \mathbb{R}^p requires $O(p)$ work
 - To find the nearest neighbor, a straightforward linear search requires $O(Np)$ work
 - To find the k nearest neighbors, $O(Np)$ for a linear search
- Size of model: $O(Np)$

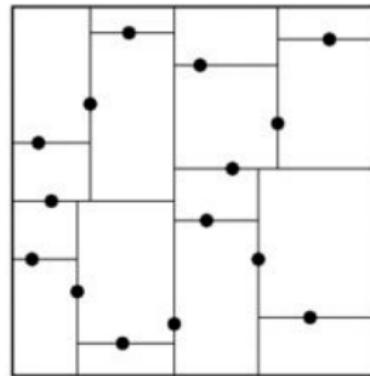
Finding Nearest Neighbors Efficiently

NN classification seems reasonable when N isn't too big, but can be problematic for large N . We can make the search **more efficient** with data structures such as

Voronoi Diagrams



k -d trees



and algorithmic techniques like **locality-sensitive hashing**.

k-d Trees: Finding Neighbors Efficiently

We use a data structure called a *k*-d tree (for *k*-dimensional tree) to store the training set for efficient querying. A *k*-d tree is a binary tree where every leaf represents a region in the space.

Building a *k*-d Tree

```
struct KDTree
    root : NODE

struct NODE
    examples : set of indices                                ▷ provided at construction
    left : NODE
    right : NODE
    median : median value

procedure BUILDTREE( $\{(\mathbf{x}_i, y_i)\}_{i=1}^N$  with  $p$  attributes)
     $T \leftarrow \text{new KDTree}()$ 
     $T.\text{root} \leftarrow \text{new NODE}(\{1, 2, \dots, N\})$ 
    SPLITNODE( $T.\text{root}$ , 0)
    return  $T$ 
```

Splitting Nodes

Building a k -d Tree: Splitting a Node

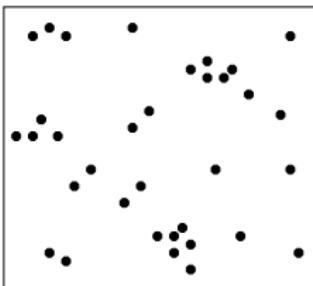
```
procedure SPLITNODE( $N$ ,  $d$ )
  if  $|N.\text{examples}| \leq \text{size limit}$  then return
   $j \leftarrow (d \bmod p) + 1$ 
   $N.\text{median} \leftarrow \text{MEDIAN}(\{x_{ij} \mid i \in N.\text{examples}\})$ 
   $N.\text{left} \leftarrow \text{new NODE}(\{i \in N.\text{examples} \mid x_{ij} \leq N.\text{median}\})$ 
   $N.\text{right} \leftarrow \text{new NODE}(\{i \in N.\text{examples} \mid x_{ij} > N.\text{median}\})$ 
  SPLITNODE( $N.\text{left}$ ,  $d + 1$ )
  SPLITNODE( $N.\text{right}$ ,  $d + 1$ )
```

- We recursively subdivide the set of examples by splitting on the median value of one attribute
- The attribute for splitting is chosen sequentially based on depth in the tree: $X_1, X_2, \dots, X_p, X_1, X_2, \dots$
- Recursion stops when nodes contain few examples

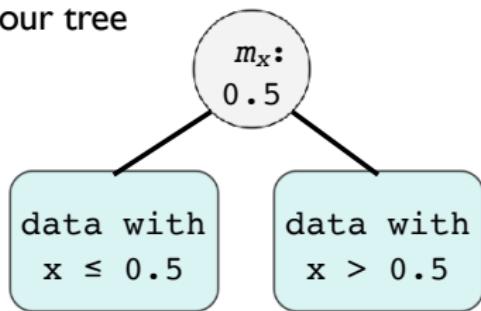
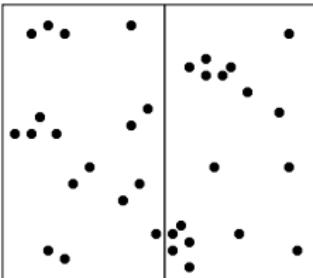
Lecture 06-2c: k -d Trees Example

A 2-Dimensional Example

- We start with a set of 2-dimensional data-points, $p_i = (x_i, y_i)$

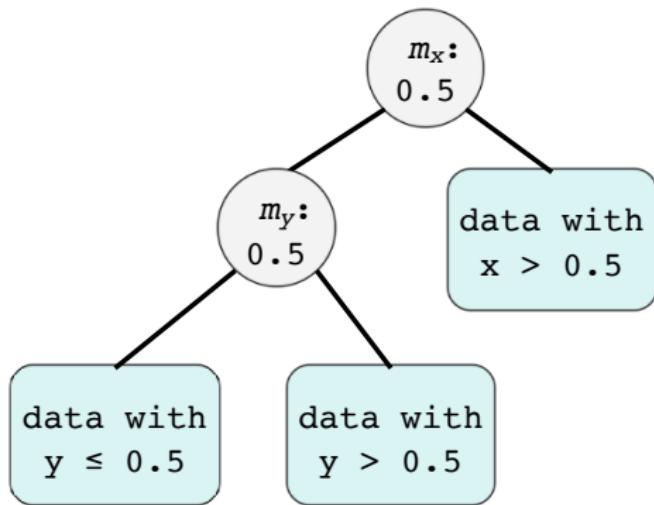
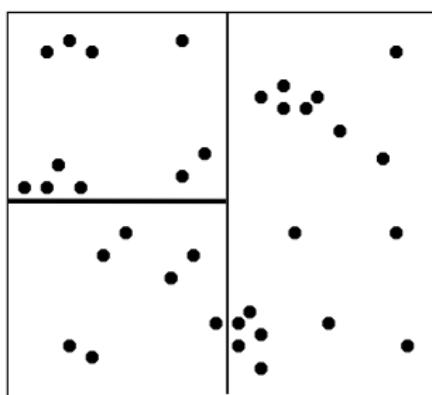


- Split along x -dimension median to start building our tree



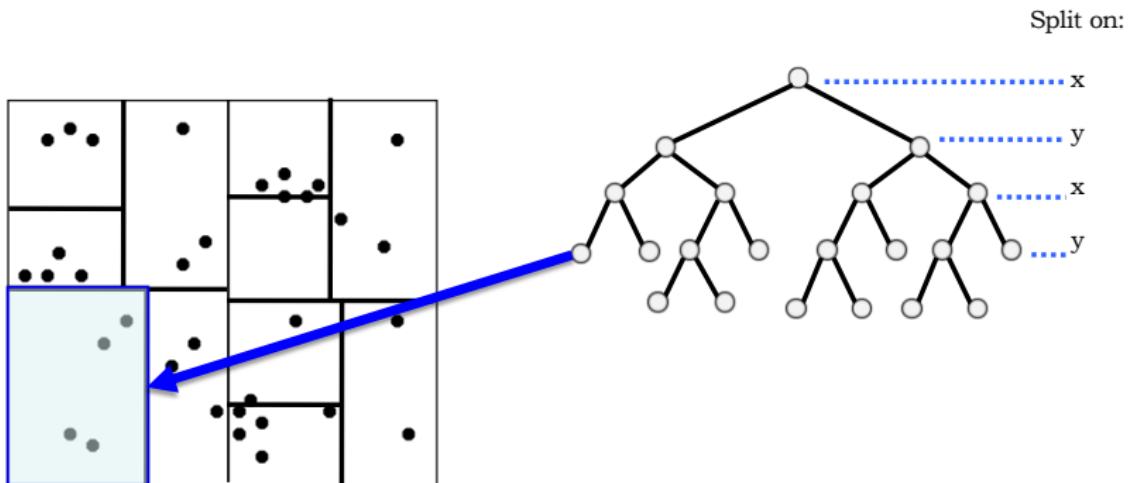
A 2-Dimensional Example

- We then split each group along y -dimension median separately



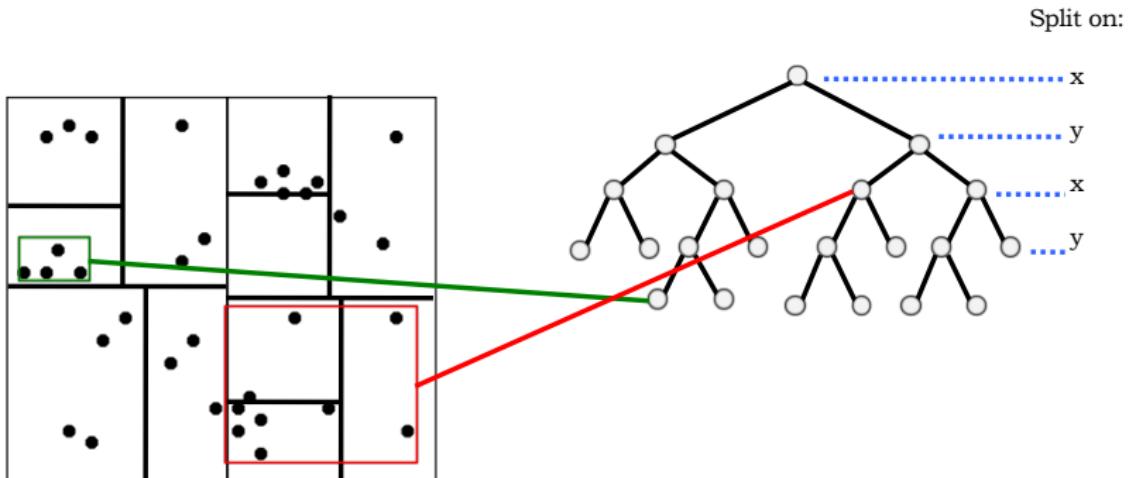
- We repeat on the other branch, and continue in each case, **splitting again** on dimensions x, y, x, y, x, y, \dots

A 2-Dimensional Example



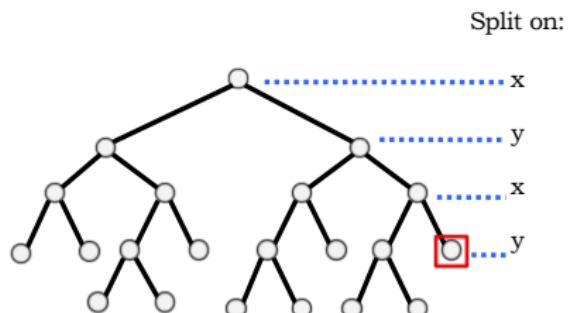
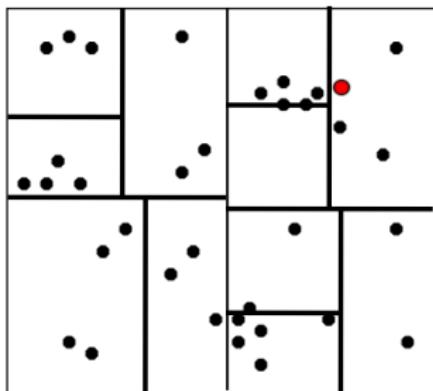
- ▶ We stop when we have small enough subsets, each of which is stored in and represented by a leaf-node of our tree
- ▶ Interior nodes store median values

A 2-Dimensional Example



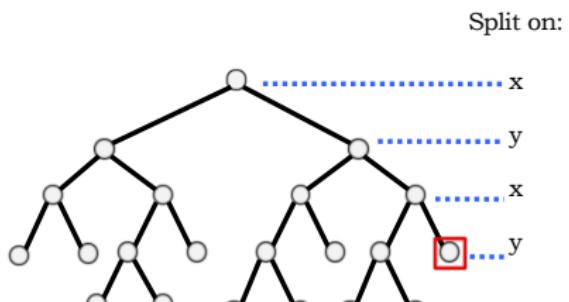
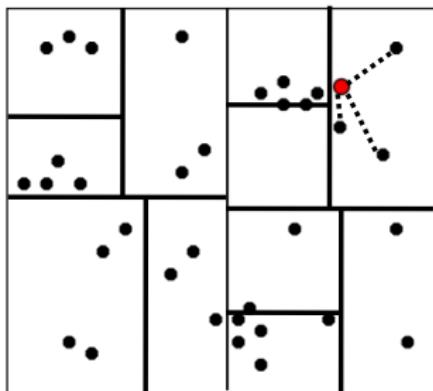
- ▶ Interior nodes store median values
- ▶ Each node (leaf **or** interior) also stores information about the least (tightest) **bounding box** of all points below it in its sub-tree

Querying the Tree for the Nearest Neighbor



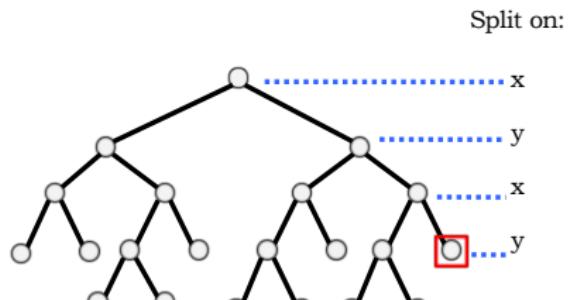
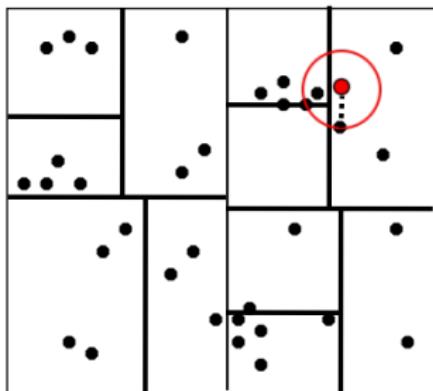
- ▶ Suppose we want to find the nearest neighbor of a new data-point (red)
- ▶ We start by isolating what sub-set it belongs to, following branches according to the median values (like a binary search tree)

Querying the Tree for the Nearest Neighbor



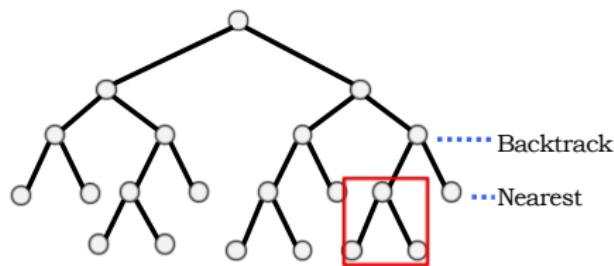
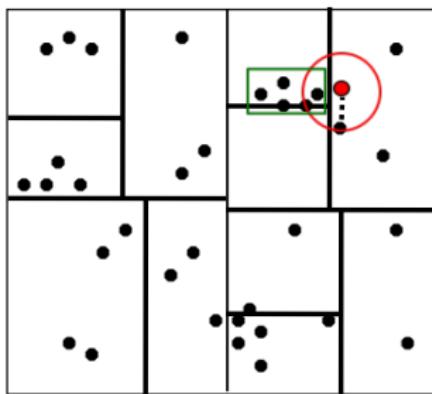
- Once we have found the proper subset, we measure all distances within it
- The closest neighbor **may be** in this set, but it **may not**

Querying the Tree for the Nearest Neighbor



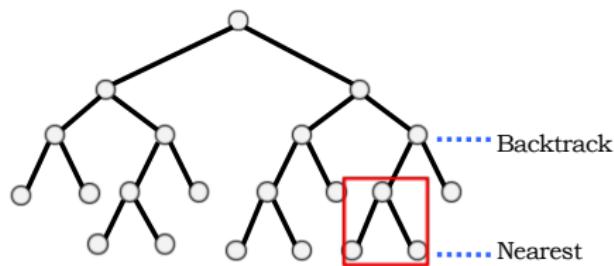
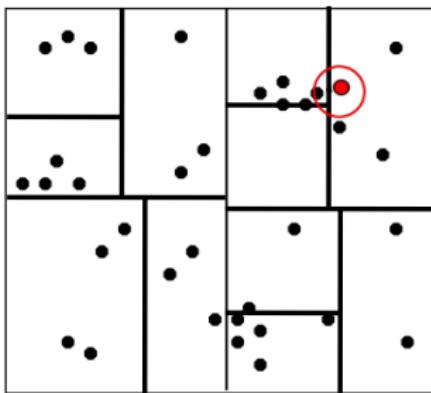
- ▶ We need to check any data-point that **could be closer** to our new point
- ▶ The tree helps us here, as we can do some **pruning** as we go backwards up the tree towards the root

Querying the Tree for the Nearest Neighbor



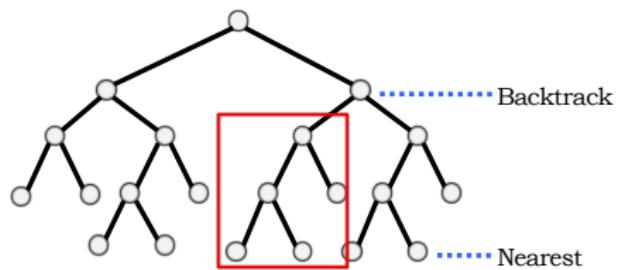
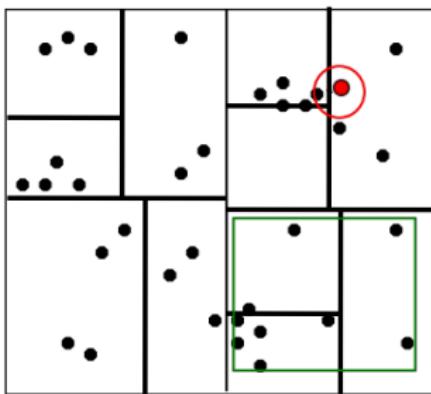
- As we back-track up the tree, we check any branch where the stored bounding box intersects our current bounds

Querying the Tree for the Nearest Neighbor



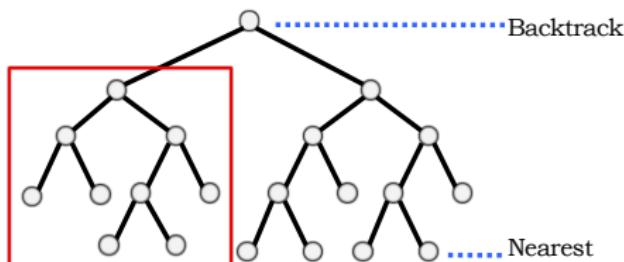
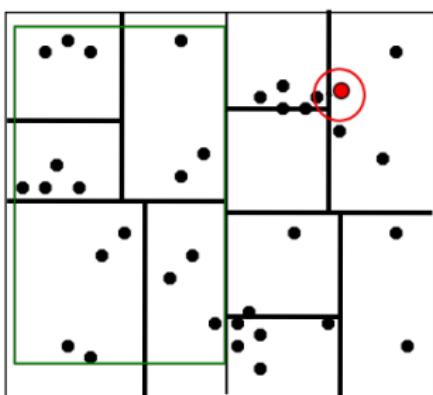
- When this happens, we compute distances to all required nodes, and update distance measure if necessary

Querying the Tree for the Nearest Neighbor



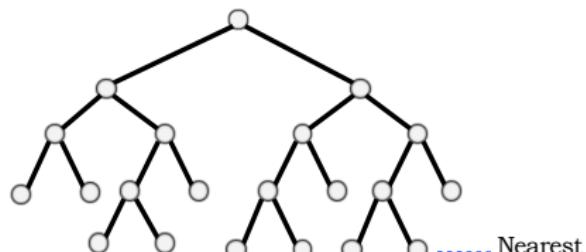
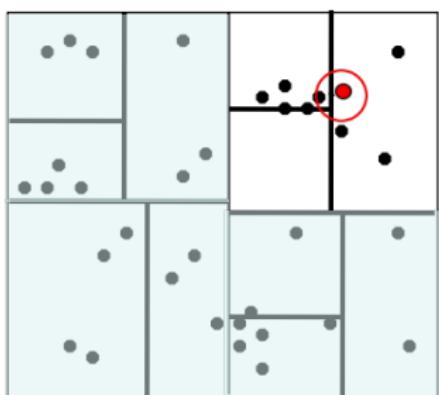
- When we see a sub-tree with a bounding box that **does not** intersect our current bound, we can *ignore it*, saving time overall

Querying the Tree for the Nearest Neighbor



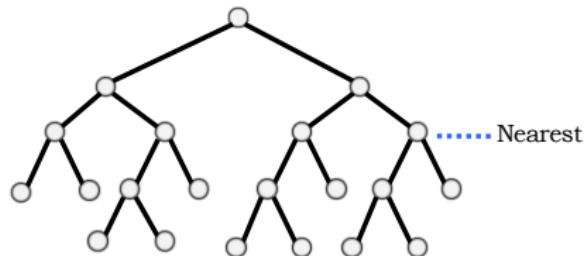
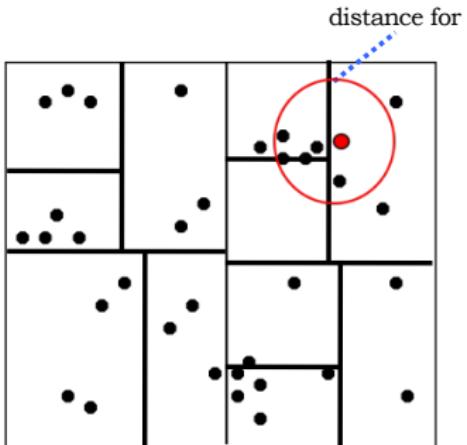
- When we see a sub-tree with a bounding box that **does not** intersect our current bound, we can **ignore it**, saving time overall
- Once we are at the root, we have found the overall nearest neighbor

Querying the Tree for the Nearest Neighbor



- ▶ The data-structure may allow us to prune off large numbers of nodes, restricting those that we need to measure distance from new point
- ▶ Although it is possible that we still have to do $O(N)$ comparisons, under many distributions of data-points, we get $O(\log N)$, significantly speeding up our algorithm for classification

K-D Trees for k -Nearest Neighbors



- ▶ If what we want is not the single nearest neighbor point, but some set of k such points (for better classification), the **exact same approach** can be used
- ▶ Works the same way, but the distance measure is set to use the full set of neighbors (i.e., distance to **farthest one** of the k nearest)

Lecture 06-2d: Locality-Sensitive Hashing

Finding a Faster Search Process

What's **faster** than looking up an object in a tree?

Looking up an object in a **hash table**!

- Hash tables are designed to map each item to a **unique** table entry
- Equal items should map to the same entry, while distinct items should map to different entries

Difficulty: Query point is not likely to **exactly** match any training point.

Resolution: Settle for **approximate nearest neighbor matching**: with high probability, we find a match that is “near” the query point.

Key idea: Points that are **nearby** in high-dimensional space will tend to **remain nearby** when **projected** down to lower dimensions.

- (Though sometimes points that are far away in high dimensions get closer together in the lower-dimensional projection)

Hashing for Collisions

For simplicity, we'll assume all attributes are Boolean-valued.

Training Phase

- ① Pick a subset of attribute indices $I \subseteq \{1, 2, \dots, p\}$.
- ② Create a hash table mapping each training point to a bin based on its attribute values for I .

Lookup Phase

- ③ For query point x_q , identify its attribute values for I and look up the corresponding entry in the hash table.
- ④ Compute distances between x_q and each training point in the hash table entry and return the closest match.

Except instead of using a **single** hash table, we use **many** different ones!

Picture of Locality-Sensitive Hashing

$$\mathbf{x}_1 = (1, 1, 0, 1, 0, 0, 1, 0)$$

$$\mathbf{x}_2 = (0, 1, 1, 0, 1, 0, 1, 0)$$

$$\mathbf{x}_3 = (1, 1, 0, 0, 1, 1, 0, 0)$$

$$\mathbf{x}_4 = (0, 1, 0, 1, 1, 1, 0, 1)$$

Picture of Locality-Sensitive Hashing

$$\mathbf{x}_1 = (1, 1, 0, 1, \boxed{0}, 0, 1, 0)$$

$$\mathbf{x}_2 = (0, 1, 1, 0, \boxed{1}, 0, 1, 0)$$

$$\mathbf{x}_3 = (1, 1, 0, 0, \boxed{1}, 1, 0, 0)$$

$$\mathbf{x}_4 = (0, 1, 0, 1, \boxed{1}, 1, 0, 1)$$

Picture of Locality-Sensitive Hashing

$$\mathbf{x}_1 = (1, 1, 0, 1, \boxed{0}, 0, 1, 0)$$

$$\mathbf{x}_2 = (0, 1, 1, 0, 1, 0, 1, 0)$$

$$\mathbf{x}_3 = (1, 1, 0, 0, 1, 1, 0, 0)$$

$$\mathbf{x}_4 = (0, 1, 0, 1, 1, 1, 0, 1)$$

$$g_1 : 1, 4, 5$$

000:
001: \mathbf{x}_2
010:
011: \mathbf{x}_4
100:
101: \mathbf{x}_3
110: \mathbf{x}_1
111:

Picture of Locality-Sensitive Hashing

$$\mathbf{x}_1 = (1, 1, 0, 1, 0, 0, 1, 0)$$

$$\mathbf{x}_2 = (0, 1, 1, 0, 1, 0, 1, 0)$$

$$\mathbf{x}_3 = (1, 1, 0, 0, 1, 1, 0, 0)$$

$$\mathbf{x}_4 = (0, 1, 0, 1, 1, 1, 0, 1)$$

$$g_1 : 1, 4, 5$$

000:
001: \mathbf{x}_2
010:
011: \mathbf{x}_4
100:
101: \mathbf{x}_3
110: \mathbf{x}_1
111:

Picture of Locality-Sensitive Hashing

$$\mathbf{x}_1 = (1, 1, 0, 1, 0, 0, 1, 0)$$

$$\mathbf{x}_2 = (0, 1, 1, 0, 1, 0, 1, 0)$$

$$\mathbf{x}_3 = (1, 1, 0, 0, 1, 1, 0, 0)$$

$$\mathbf{x}_4 = (0, 1, 0, 1, 1, 1, 0, 1)$$

$$g_1 : 1, 4, 5$$

$$g_2 : 2, 4, 7$$

000:
001: \mathbf{x}_2
010:
011: \mathbf{x}_4
100:
101: \mathbf{x}_3
110: \mathbf{x}_1
111:

000:
001:
010:
011:
100: \mathbf{x}_3
101: \mathbf{x}_2
110: \mathbf{x}_4
111: \mathbf{x}_1

Picture of Locality-Sensitive Hashing

$$\mathbf{x}_1 = (1, 1, 0, 1, 0, 0, 1, 0)$$
$$\mathbf{x}_2 = (0, 1, 1, 0, 1, 0, 1, 0)$$
$$\mathbf{x}_3 = (1, 1, 0, 0, 1, 1, 0, 0)$$
$$\mathbf{x}_4 = (0, 1, 0, 1, 1, 1, 0, 1)$$

$$g_1 : 1, 4, 5$$

000:
001: \mathbf{x}_2
010:
011: \mathbf{x}_4
100:
101: \mathbf{x}_3
110: \mathbf{x}_1
111:

$$g_2 : 2, 4, 7$$

000:
001:
010:
011:
100: \mathbf{x}_3
101: \mathbf{x}_2
110: \mathbf{x}_4
111: \mathbf{x}_1

Picture of Locality-Sensitive Hashing

$\mathbf{x}_1 = (1, 1, 0, 1, 0, 0, 1, 0)$
 $\mathbf{x}_2 = (0, 1, 1, 0, 1, 0, 1, 0)$
 $\mathbf{x}_3 = (1, 1, 0, 0, 1, 1, 0, 0)$
 $\mathbf{x}_4 = (0, 1, 0, 1, 1, 1, 0, 1)$

$g_1 : 1, 4, 5$

000:
001: \mathbf{x}_2
010:
011: \mathbf{x}_4
100:
101: \mathbf{x}_3
110: \mathbf{x}_1
111:

$g_2 : 2, 4, 7$

000:
001:
010:
011:
100: \mathbf{x}_3
101: \mathbf{x}_2
110: \mathbf{x}_4
111: \mathbf{x}_1

$g_3 : 1, 3, 8$

000:
001: \mathbf{x}_4
010: \mathbf{x}_2
011:
100: $\mathbf{x}_1, \mathbf{x}_3$
101:
110:
111: \mathbf{x}_1

Picture of Locality-Sensitive Hashing

$$\mathbf{x}_1 = (1, 1, 0, 1, 0, 0, 1, 0)$$

$$\mathbf{x}_2 = (0, 1, 1, 0, 1, 0, 1, 0)$$

$$\mathbf{x}_3 = (1, 1, 0, 0, 1, 1, 0, 0)$$

$$\mathbf{x}_4 = (0, 1, 0, 1, 1, 1, 0, 1)$$

$$g_1 : 1, 4, 5$$

000:
001: \mathbf{x}_2
010:
011: \mathbf{x}_4
100:
101: \mathbf{x}_3
110: \mathbf{x}_1
111:

$$g_2 : 2, 4, 7$$

000:
001:
010:
011:
100: \mathbf{x}_3
101: \mathbf{x}_2
110: \mathbf{x}_4
111: \mathbf{x}_1

$$g_3 : 1, 3, 8$$

000:
001: \mathbf{x}_4
010: \mathbf{x}_2
011:
100: $\mathbf{x}_1, \mathbf{x}_3$
101:
110:
111: \mathbf{x}_1

Picture of Locality-Sensitive Hashing

$$\mathbf{x}_1 = (1, 1, \boxed{0}, 1, 0, 0, 1, 0)$$

$$\mathbf{x}_2 = (0, 1, 1, \boxed{0}, 1, 0, 1, 0)$$

$$\mathbf{x}_3 = (1, 1, 0, \boxed{0}, 1, 1, 0, 0)$$

$$\mathbf{x}_4 = (0, 1, \boxed{0}, 1, 1, 1, 0, 1)$$

$$g_1 : 1, 4, 5$$

000:
001: \mathbf{x}_2
010:
011: \mathbf{x}_4
100:
101: \mathbf{x}_3
110: \mathbf{x}_1
111:

$$g_2 : 2, 4, 7$$

000:
001:
010:
011:
100: \mathbf{x}_3
101: \mathbf{x}_2
110: \mathbf{x}_4
111: \mathbf{x}_1

$$g_3 : 1, 3, 8$$

000:
001: \mathbf{x}_4
010: \mathbf{x}_2
011:
100: $\mathbf{x}_1, \mathbf{x}_3$
101:
110:
111:

$$g_4 : 3, 4, 6$$

000:
001: \mathbf{x}_3
010: \mathbf{x}_1
011: \mathbf{x}_4
100: \mathbf{x}_2
101:
110:
111:

Picture of Locality-Sensitive Hashing

$$\mathbf{x}_1 = (1, 1, 0, 1, 0, 0, 1, 0)$$

$$\mathbf{x}_2 = (0, 1, 1, 0, 1, 0, 1, 0)$$

$$\mathbf{x}_3 = (1, 1, 0, 0, 1, 1, 0, 0)$$

$$\mathbf{x}_4 = (0, 1, 0, 1, 1, 1, 0, 1)$$

$$\mathbf{x}_q = (0, 1, 1, 0, 0, 0, 0, 1)$$

$$g_1 : 1, 4, 5$$

000:
001: \mathbf{x}_2
010:
011: \mathbf{x}_4
100:
101: \mathbf{x}_3
110: \mathbf{x}_1
111:

$$g_2 : 2, 4, 7$$

000:
001:
010:
011:
100: \mathbf{x}_3
101: \mathbf{x}_2
110: \mathbf{x}_4
111: \mathbf{x}_1

$$g_3 : 1, 3, 8$$

000:
001: \mathbf{x}_4
010: \mathbf{x}_2
011:
100: $\mathbf{x}_1, \mathbf{x}_3$
101:
110:
111:

$$g_4 : 3, 4, 6$$

000:
001: \mathbf{x}_3
010: \mathbf{x}_1
011: \mathbf{x}_4
100: \mathbf{x}_2
101:
110:
111:

Picture of Locality-Sensitive Hashing

$$\mathbf{x}_1 = (1, 1, 0, 1, 0, 0, 1, 0)$$

$$\mathbf{x}_2 = (0, 1, 1, 0, 1, 0, 1, 0)$$

$$\mathbf{x}_3 = (1, 1, 0, 0, 1, 1, 0, 0)$$

$$\mathbf{x}_4 = (0, 1, 0, 1, 1, 1, 0, 1)$$

$$\mathbf{x}_q = (0, 1, 1, 0, 0, 0, 0, 1)$$

$$g_1 : 1, 4, 5$$

000:
001: \mathbf{x}_2
010:
011: \mathbf{x}_4
100:
101: \mathbf{x}_3
110: \mathbf{x}_1
111:

$$g_2 : 2, 4, 7$$

000:
001:
010:
011:
100: \mathbf{x}_3
101: \mathbf{x}_2
110: \mathbf{x}_4
111: \mathbf{x}_1

$$g_3 : 1, 3, 8$$

000:
001: \mathbf{x}_4
010: \mathbf{x}_2
011:
100: $\mathbf{x}_1, \mathbf{x}_3$
101:
110:
111:

$$g_4 : 3, 4, 6$$

000:
001: \mathbf{x}_3
010: \mathbf{x}_1
011: \mathbf{x}_4
100: \mathbf{x}_2
101:
110:
111:

$$\text{NeighborSet}(\mathbf{x}_q) = g_1(\mathbf{x}_q) \cup g_2(\mathbf{x}_q) \cup g_3(\mathbf{x}_q) \cup g_4(\mathbf{x}_q)$$

Picture of Locality-Sensitive Hashing

$$\mathbf{x}_1 = (1, 1, 0, 1, 0, 0, 1, 0)$$

$$\mathbf{x}_2 = (0, 1, 1, 0, 1, 0, 1, 0)$$

$$\mathbf{x}_3 = (1, 1, 0, 0, 1, 1, 0, 0)$$

$$\mathbf{x}_4 = (0, 1, 0, 1, 1, 1, 0, 1)$$

$$g_1 : 1, 4, 5$$

000:
001: \mathbf{x}_2
010:
011: \mathbf{x}_4
100:
101: \mathbf{x}_3
110: \mathbf{x}_1
111:

$$g_2 : 2, 4, 7$$

000:
001:
010:
011:
100: \mathbf{x}_3
101: \mathbf{x}_2
110: \mathbf{x}_4
111: \mathbf{x}_1

$$g_3 : 1, 3, 8$$

000:
001: \mathbf{x}_4
010: \mathbf{x}_2
011:
100: $\mathbf{x}_1, \mathbf{x}_3$
101:
110:
111:

$$g_4 : 3, 4, 6$$

000:
001: \mathbf{x}_3
010: \mathbf{x}_1
011: \mathbf{x}_4
100: \mathbf{x}_2
101:
110:
111:

$$\mathbf{x}_q = (\underline{0}, 1, 1, \underline{0}, \underline{0}, 0, 0, 1)$$

$$\text{NeighborSet}(\mathbf{x}_q) = g_1(\mathbf{x}_q) \cup g_2(\mathbf{x}_q) \cup g_3(\mathbf{x}_q) \cup g_4(\mathbf{x}_q)$$

Picture of Locality-Sensitive Hashing

$$\mathbf{x}_1 = (1, 1, 0, 1, 0, 0, 1, 0)$$

$$\mathbf{x}_2 = (0, 1, 1, 0, 1, 0, 1, 0)$$

$$\mathbf{x}_3 = (1, 1, 0, 0, 1, 1, 0, 0)$$

$$\mathbf{x}_4 = (0, 1, 0, 1, 1, 1, 0, 1)$$

$$g_1 : 1, 4, 5$$

000:
001: \mathbf{x}_2
010:
011: \mathbf{x}_4
100:
101: \mathbf{x}_3
110: \mathbf{x}_1
111:

$$g_2 : 2, 4, 7$$

000:
001:
010:
011:
100: \mathbf{x}_3
101: \mathbf{x}_2
110: \mathbf{x}_4
111: \mathbf{x}_1

$$g_3 : 1, 3, 8$$

000:
001: \mathbf{x}_4
010: \mathbf{x}_2
011:
100: $\mathbf{x}_1, \mathbf{x}_3$
101:
110:
111:

$$g_4 : 3, 4, 6$$

000:
001: \mathbf{x}_3
010: \mathbf{x}_1
011: \mathbf{x}_4
100: \mathbf{x}_2
101:
110:
111:

$$\mathbf{x}_q = (\underline{0}, 1, 1, \underline{0}, \underline{0}, 0, 0, 1)$$

$$\begin{aligned}\text{NeighborSet}(\mathbf{x}_q) &= g_1(\mathbf{x}_q) \cup g_2(\mathbf{x}_q) \cup g_3(\mathbf{x}_q) \cup g_4(\mathbf{x}_q) \\ &= \{\}\end{aligned}$$

Picture of Locality-Sensitive Hashing

$$\mathbf{x}_1 = (1, 1, 0, 1, 0, 0, 1, 0)$$

$$\mathbf{x}_2 = (0, 1, 1, 0, 1, 0, 1, 0)$$

$$\mathbf{x}_3 = (1, 1, 0, 0, 1, 1, 0, 0)$$

$$\mathbf{x}_4 = (0, 1, 0, 1, 1, 1, 0, 1)$$

$$g_1 : 1, 4, 5$$

000:
001: \mathbf{x}_2
010:
011: \mathbf{x}_4
100:
101: \mathbf{x}_3
110: \mathbf{x}_1
111:

$$g_2 : 2, 4, 7$$

000:
001:
010:
011:
100: \mathbf{x}_3
101: \mathbf{x}_2
110: \mathbf{x}_4
111:

$$g_3 : 1, 3, 8$$

000:
001: \mathbf{x}_4
010: \mathbf{x}_2
011:
100: $\mathbf{x}_1, \mathbf{x}_3$
101:
110:
111:

$$g_4 : 3, 4, 6$$

000:
001: \mathbf{x}_3
010: \mathbf{x}_1
011: \mathbf{x}_4
100: \mathbf{x}_2
101:
110:
111:

$$\mathbf{x}_q = (0, \boxed{1}, 1, \boxed{0}, 0, 0, \boxed{0}, 1)$$

$$\begin{aligned}\text{NeighborSet}(\mathbf{x}_q) &= g_1(\mathbf{x}_q) \cup g_2(\mathbf{x}_q) \cup g_3(\mathbf{x}_q) \cup g_4(\mathbf{x}_q) \\ &= \{\}\end{aligned}$$

Picture of Locality-Sensitive Hashing

$$\mathbf{x}_1 = (1, 1, 0, 1, 0, 0, 1, 0)$$

$$\mathbf{x}_2 = (0, 1, 1, 0, 1, 0, 1, 0)$$

$$\mathbf{x}_3 = (1, 1, 0, 0, 1, 1, 0, 0)$$

$$\mathbf{x}_4 = (0, 1, 0, 1, 1, 1, 0, 1)$$

$$g_1 : 1, 4, 5$$

000:
001: \mathbf{x}_2
010:
011: \mathbf{x}_4
100:
101: \mathbf{x}_3
110: \mathbf{x}_1
111:

$$g_2 : 2, 4, 7$$

000:
001:
010:
011:
100: \mathbf{x}_3
101: \mathbf{x}_2
110: \mathbf{x}_4
111:

$$g_3 : 1, 3, 8$$

000:
001: \mathbf{x}_4
010: \mathbf{x}_2
011:
100: $\mathbf{x}_1, \mathbf{x}_3$
101:
110:
111:

$$g_4 : 3, 4, 6$$

000:
001: \mathbf{x}_3
010: \mathbf{x}_1
011: \mathbf{x}_4
100: \mathbf{x}_2
101:
110:
111:

$$\mathbf{x}_q = (0, \boxed{1}, 1, \boxed{0}, 0, 0, \boxed{0}, 1)$$

$$\begin{aligned}\text{NeighborSet}(\mathbf{x}_q) &= g_1(\mathbf{x}_q) \cup g_2(\mathbf{x}_q) \cup g_3(\mathbf{x}_q) \cup g_4(\mathbf{x}_q) \\ &= \{\} \cup \{\mathbf{x}_3\}\end{aligned}$$

Picture of Locality-Sensitive Hashing

$$\mathbf{x}_1 = (1, 1, 0, 1, 0, 0, 1, 0)$$

$$\mathbf{x}_2 = (0, 1, 1, 0, 1, 0, 1, 0)$$

$$\mathbf{x}_3 = (1, 1, 0, 0, 1, 1, 0, 0)$$

$$\mathbf{x}_4 = (0, 1, 0, 1, 1, 1, 0, 1)$$

$$g_1 : 1, 4, 5$$

000:
001: \mathbf{x}_2
010:
011: \mathbf{x}_4
100:
101: \mathbf{x}_3
110: \mathbf{x}_1
111:

$$g_2 : 2, 4, 7$$

000:
001:
010:
011:
100: \mathbf{x}_3
101: \mathbf{x}_2
110: \mathbf{x}_4
111:

$$g_3 : 1, 3, 8$$

000:
001: \mathbf{x}_4
010: \mathbf{x}_2
011:
100: $\mathbf{x}_1, \mathbf{x}_3$
101:
110:
111:

$$g_4 : 3, 4, 6$$

000:
001: \mathbf{x}_3
010: \mathbf{x}_1
011: \mathbf{x}_4
100: \mathbf{x}_2
101:
110:
111:

$$\mathbf{x}_q = (\underline{0}, 1, \underline{1}, 0, 0, 0, 0, \underline{1})$$

$$\begin{aligned}\text{NeighborSet}(\mathbf{x}_q) &= g_1(\mathbf{x}_q) \cup g_2(\mathbf{x}_q) \cup g_3(\mathbf{x}_q) \cup g_4(\mathbf{x}_q) \\ &= \{\} \cup \{\mathbf{x}_3\}\end{aligned}$$

Picture of Locality-Sensitive Hashing

$$\mathbf{x}_1 = (1, 1, 0, 1, 0, 0, 1, 0)$$

$$\mathbf{x}_2 = (0, 1, 1, 0, 1, 0, 1, 0)$$

$$\mathbf{x}_3 = (1, 1, 0, 0, 1, 1, 0, 0)$$

$$\mathbf{x}_4 = (0, 1, 0, 1, 1, 1, 0, 1)$$

$$g_1 : 1, 4, 5$$

000:
001: \mathbf{x}_2
010:
011: \mathbf{x}_4
100:
101: \mathbf{x}_3
110: \mathbf{x}_1
111:

$$g_2 : 2, 4, 7$$

000:
001:
010:
011:
100: \mathbf{x}_3
101: \mathbf{x}_2
110: \mathbf{x}_4
111:

$$g_3 : 1, 3, 8$$

000:
001: \mathbf{x}_4
010: \mathbf{x}_2
011:
100: $\mathbf{x}_1, \mathbf{x}_3$
101:
110:
111:

$$g_4 : 3, 4, 6$$

000:
001: \mathbf{x}_3
010: \mathbf{x}_1
011: \mathbf{x}_4
100: \mathbf{x}_2
101:
110:
111:

$$\mathbf{x}_q = (\underline{0}, 1, \underline{1}, 0, 0, 0, 0, \underline{1})$$

$$\begin{aligned}\text{NeighborSet}(\mathbf{x}_q) &= g_1(\mathbf{x}_q) \cup g_2(\mathbf{x}_q) \cup g_3(\mathbf{x}_q) \cup g_4(\mathbf{x}_q) \\ &= \{\} \cup \{\mathbf{x}_3\} \cup \{\}\end{aligned}$$

Picture of Locality-Sensitive Hashing

$$\mathbf{x}_1 = (1, 1, 0, 1, 0, 0, 1, 0)$$

$$\mathbf{x}_2 = (0, 1, 1, 0, 1, 0, 1, 0)$$

$$\mathbf{x}_3 = (1, 1, 0, 0, 1, 1, 0, 0)$$

$$\mathbf{x}_4 = (0, 1, 0, 1, 1, 1, 0, 1)$$

$$g_1 : 1, 4, 5$$

000:
001: \mathbf{x}_2
010:
011: \mathbf{x}_4
100:
101: \mathbf{x}_3
110: \mathbf{x}_1
111:

$$g_2 : 2, 4, 7$$

000:
001:
010:
011:
100: \mathbf{x}_3
101: \mathbf{x}_2
110: \mathbf{x}_4
111:

$$g_3 : 1, 3, 8$$

000:
001: \mathbf{x}_4
010: \mathbf{x}_2
011:
100: $\mathbf{x}_1, \mathbf{x}_3$
101:
110:
111:

$$g_4 : 3, 4, 6$$

000:
001: \mathbf{x}_3
010: \mathbf{x}_1
011: \mathbf{x}_4
100: \mathbf{x}_2
101:
110:
111:

$$\mathbf{x}_q = (0, 1, \boxed{1}, \boxed{0}, 0, \boxed{0}, 0, 1)$$

$$\begin{aligned}\text{NeighborSet}(\mathbf{x}_q) &= g_1(\mathbf{x}_q) \cup g_2(\mathbf{x}_q) \cup g_3(\mathbf{x}_q) \cup g_4(\mathbf{x}_q) \\ &= \{\} \cup \{\mathbf{x}_3\} \cup \{\}\end{aligned}$$

Picture of Locality-Sensitive Hashing

$$\mathbf{x}_1 = (1, 1, 0, 1, 0, 0, 1, 0)$$

$$\mathbf{x}_2 = (0, 1, 1, 0, 1, 0, 1, 0)$$

$$\mathbf{x}_3 = (1, 1, 0, 0, 1, 1, 0, 0)$$

$$\mathbf{x}_4 = (0, 1, 0, 1, 1, 1, 0, 1)$$

$$g_1 : 1, 4, 5$$

000:
001: \mathbf{x}_2
010:
011: \mathbf{x}_4
100:
101: \mathbf{x}_3
110: \mathbf{x}_1
111:

$$g_2 : 2, 4, 7$$

000:
001:
010:
011:
100: \mathbf{x}_3
101: \mathbf{x}_2
110: \mathbf{x}_4
111:

$$g_3 : 1, 3, 8$$

000:
001: \mathbf{x}_4
010: \mathbf{x}_2
011:
100: $\mathbf{x}_1, \mathbf{x}_3$
101:
110:
111:

$$g_4 : 3, 4, 6$$

000:
001: \mathbf{x}_3
010: \mathbf{x}_1
011: \mathbf{x}_4
100: \mathbf{x}_2
101:
110:
111:

$$\mathbf{x}_q = (0, 1, \boxed{1}, \boxed{0}, 0, \boxed{0}, 0, 1)$$

$$\begin{aligned}\text{NeighborSet}(\mathbf{x}_q) &= g_1(\mathbf{x}_q) \cup g_2(\mathbf{x}_q) \cup g_3(\mathbf{x}_q) \cup g_4(\mathbf{x}_q) \\ &= \{\} \cup \{\mathbf{x}_3\} \cup \{\} \cup \{\mathbf{x}_2\}\end{aligned}$$

Picture of Locality-Sensitive Hashing

$$\mathbf{x}_1 = (1, 1, 0, 1, 0, 0, 1, 0)$$

$$\mathbf{x}_2 = (0, 1, 1, 0, 1, 0, 1, 0)$$

$$\mathbf{x}_3 = (1, 1, 0, 0, 1, 1, 0, 0)$$

$$\mathbf{x}_4 = (0, 1, 0, 1, 1, 1, 0, 1)$$

$$\mathbf{x}_q = (0, 1, 1, 0, 0, 0, 0, 1)$$

$$g_1 : 1, 4, 5$$

000:
001: \mathbf{x}_2
010:
011: \mathbf{x}_4
100:
101: \mathbf{x}_3
110: \mathbf{x}_1
111:

$$g_2 : 2, 4, 7$$

000:
001:
010:
011:
100: \mathbf{x}_3
101: \mathbf{x}_2
110: \mathbf{x}_4
111:

$$g_3 : 1, 3, 8$$

000:
001: \mathbf{x}_4
010: \mathbf{x}_2
011:
100: $\mathbf{x}_1, \mathbf{x}_3$
101:
110:
111:

$$g_4 : 3, 4, 6$$

000:
001: \mathbf{x}_3
010: \mathbf{x}_1
011: \mathbf{x}_4
100: \mathbf{x}_2
101:
110:
111:

$$\begin{aligned}\text{NeighborSet}(\mathbf{x}_q) &= g_1(\mathbf{x}_q) \cup g_2(\mathbf{x}_q) \cup g_3(\mathbf{x}_q) \cup g_4(\mathbf{x}_q) \\ &= \{\} \cup \{\mathbf{x}_3\} \cup \{\} \cup \{\mathbf{x}_2\} \\ &= \{\mathbf{x}_2, \mathbf{x}_3\}\end{aligned}$$

CS 457/557: Machine Learning

Lecture 06-3: Maximum Margin and Support Vector Classifiers

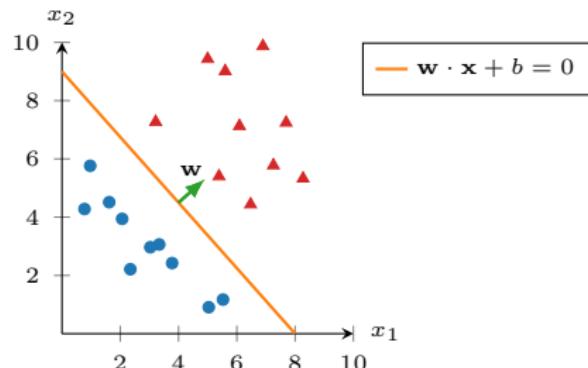
Lecture 06-3a: Revisiting Linear Classification

Quick Recap on Perceptron

For a data set $\{(\mathbf{x}_i, y_i)\}_{i=1}^N$ with p input attributes, our goal in **perceptron learning** is to identify a **separating hyperplane** in \mathbb{R}^p .

We **modify notation** slightly so that the hyperplane parameters (weights) are given by the vector $\mathbf{w} = (w_1, w_2, \dots, w_p)$ **and** the scalar intercept (bias) b (which corresponds to w_0 in earlier formulation).

Then a hyperplane is defined as $\{\mathbf{x} \in \mathbb{R}^p \mid \mathbf{w} \cdot \mathbf{x} + b = 0\}$.



Classification with Perceptron

We also **modify notation** so that class labels are $y_i \in \{-1, 1\}$ instead of $y_i \in \{0, 1\}$. Then our hypothesis function is

$$h_{\mathbf{w}, b}(\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{w} \cdot \mathbf{x} + b \geq 0 \\ -1 & \text{if } \mathbf{w} \cdot \mathbf{x} + b < 0 \end{cases},$$

which is written more compactly as

$$h_{\mathbf{w}, b}(\mathbf{x}) = \text{sgn}(\mathbf{w} \cdot \mathbf{x} + b)$$

where **sgn** is the **sign function** defined as $\text{sgn}(t) = 1$ if $t \geq 0$ and -1 if $t < 0$ for all $t \in \mathbb{R}$.

Perceptron Learning

- ① Start with $t = 0$ and initial weights $\mathbf{w}^{(0)}$ and $b^{(0)}$
- ② Choose an input \mathbf{x}_i that is **incorrectly classified**
- ③ Update components in weight vector for iteration $t + 1$ using

$$b^{(t+1)} \leftarrow b^{(t)} + \alpha y_i$$

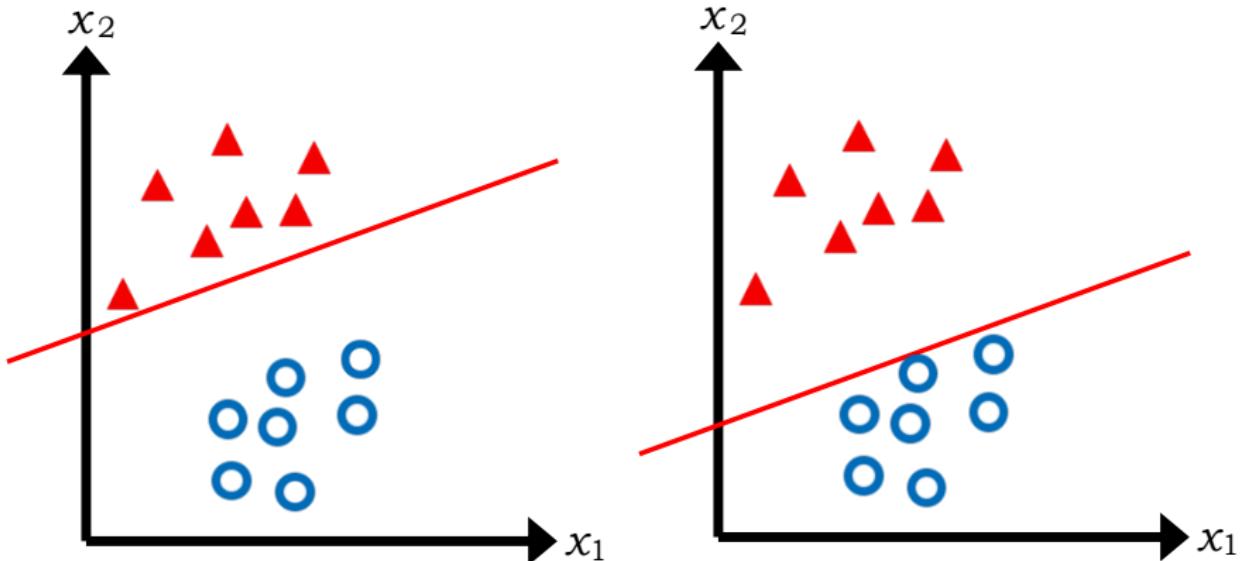
$$w_j^{(t+1)} \leftarrow w_j^{(t)} + \alpha y_i x_{ij} \quad \forall j \in \{1, 2, \dots, p\}$$

- ④ Increment t and repeat Steps 2 and 3 until **no classification errors remain**

If the data $\{(\mathbf{x}_i, y_i)\}_{i=1}^N$ is linearly separable, then the perceptron learning algorithm will terminate with a separating hyperplane!

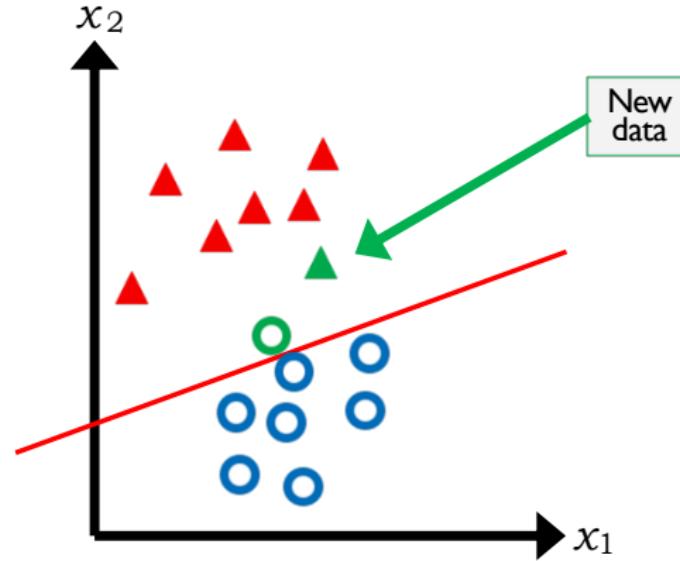
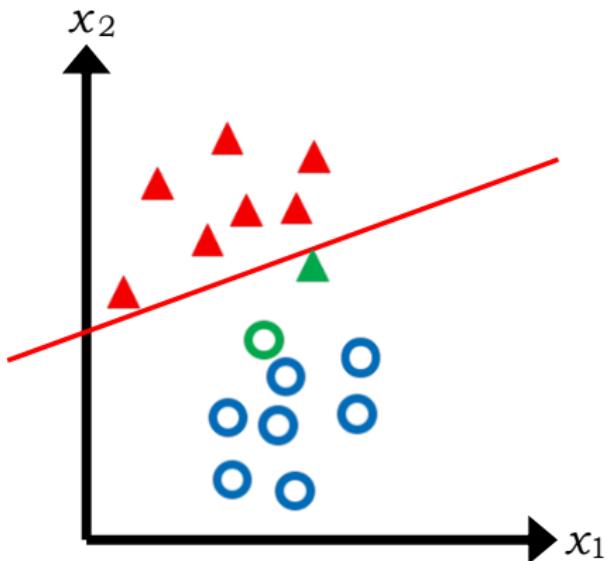
- Separating hyperplane (\mathbf{w}, b) satisfies $h_{\mathbf{w}, b}(\mathbf{x}_i) = y_i$ for all i
- A separating hyperplane also satisfies $y_i(\mathbf{w} \cdot \mathbf{x} + b) \geq 0$ for all i

Data Separation



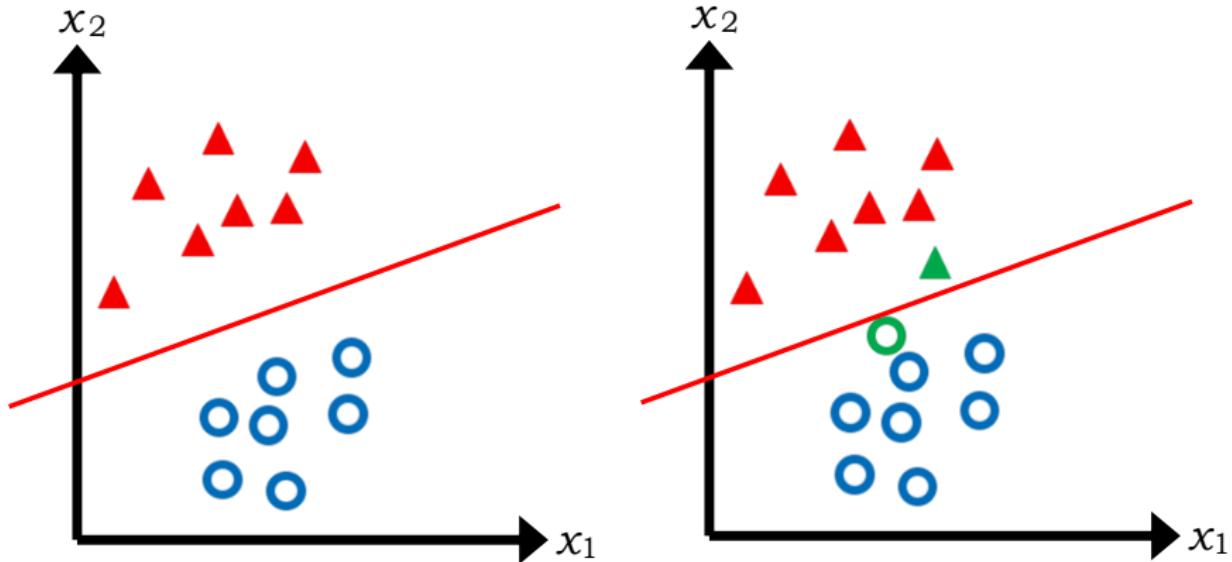
- ▶ Linear classification with a perceptron or logistic function look for a dividing line in the data (or a plane, or other linearly defined structure)
 - ▶ Often **multiple** lines are possible
 - ▶ Essentially, the algorithms are **indifferent**: they don't care which line we pick
 - ▶ In the example seen here, either classification line separates data perfectly well

“Fragile” Separation



- ▶ As more data comes in, these classifiers may start to fail
 - ▶ A separator that is too close to one cluster or the other now makes mistakes
 - ▶ May happen even if new data follows same distribution seen in the training set

“Robust” Separation



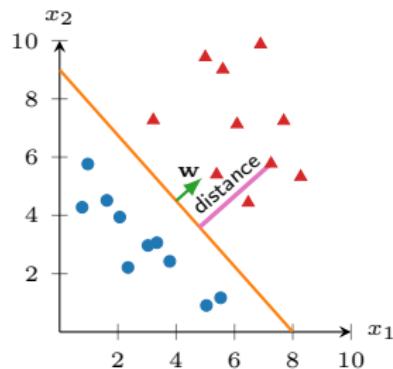
- ▶ What we want is a **large margin** separator: a separation that has the largest distance possible from each part of our data-set
- ▶ This will often give much better performance when used on new data

Lecture 06-3b: Finding Separating Hyperplanes

Evaluating Separating Hyperplanes

We want to find a separating hyperplane that is **farthest** from the training data.

For a separating hyperplane (\mathbf{w}, b) and a training point \mathbf{x}_i , we measure the perpendicular distance between them:



With some work, we can derive the distance as

$$d((\mathbf{w}, b), \mathbf{x}_i) = \frac{|\mathbf{w} \cdot \mathbf{x}_i + b|}{\sqrt{\mathbf{w} \cdot \mathbf{w}}} = \frac{|\mathbf{w} \cdot \mathbf{x}_i + b|}{\|\mathbf{w}\|_2}$$

$$\text{where } \|\mathbf{w}\|_2 = \sqrt{\sum_{j=1}^p w_j^2}.$$

The smallest such distance is called the **margin** of the hyperplane:

$$\begin{aligned}\gamma(\mathbf{w}, b) &= \min_{i=1}^N d((\mathbf{w}, b), \mathbf{x}_i) \\ &= \min_{i=1}^N \frac{|\mathbf{w} \cdot \mathbf{x}_i + b|}{\|\mathbf{w}\|_2}.\end{aligned}$$

The Maximum Margin Separating Hyperplane

We want to find a separating hyperplane that is **farthest** from the training data: we want a **maximum margin separating hyperplane**.

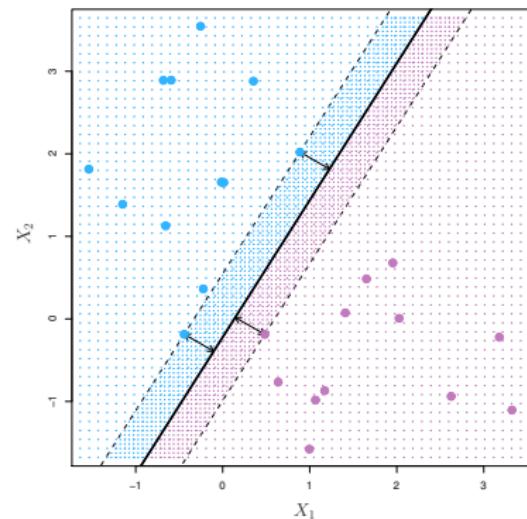
$$\max_{\mathbf{w}, b} \gamma(\mathbf{w}, b)$$

$$\text{s.t. } y_i(\mathbf{w} \cdot \mathbf{x}_i + b) \geq 0 \quad \forall i$$

We want to find the “widest road” that can fit between the two classes. The points on the edges of this road (the **margin**) are called **support vectors**.

- If support vectors are moved, the hyperplane **changes**.
- If other points are moved slightly, the hyperplane **does not change!**

The hyperplane depends on only a **small** number of data points!



Taken from “An Introduction to Statistical Learning, with applications in R” (Springer, 2013) with permission from the authors: G. James, D. Witten, T. Hastie and R. Tibshirani.

Finding the Maximum Margin Separating Hyperplane

The specification of a hyperplane (\mathbf{w}, b) is **scale-invariant** because

$$\mathbf{w} \cdot \mathbf{x} + b = 0 \Leftrightarrow (a\mathbf{w}) \cdot \mathbf{x} + ab = 0$$

for any constant $a \in \mathbb{R}$. Therefore, we can add some requirements on \mathbf{w} to aid in interpretation of distances.

Option 1: If \mathbf{w} is constrained to be a unit vector (i.e., $\|\mathbf{w}\|_2 = 1$), then the margin formula simplifies:

$$\gamma(\mathbf{w}, b) = \min_{i=1}^N \frac{|\mathbf{w} \cdot \mathbf{x}_i + b|}{\|\mathbf{w}\|_2} = \min_{i=1}^N |\mathbf{w} \cdot \mathbf{x}_i + b|.$$

Optimization Problem Formulations

We can formulate the problem of finding a **maximum margin separating hyperplane** as an optimization problem:

Straightforward formulation:

$$\begin{aligned} \max_{\mathbf{w}, b} \quad & M \\ \text{s.t.} \quad & y_i (\mathbf{w} \cdot \mathbf{x}_i + b) \geq 0 \quad \forall i \\ & M = \min_{i=1}^N |\mathbf{w} \cdot \mathbf{x}_i + b| \\ & \sum_{j=1}^p w_j^2 = 1 \\ & M \in \mathbb{R} \end{aligned}$$

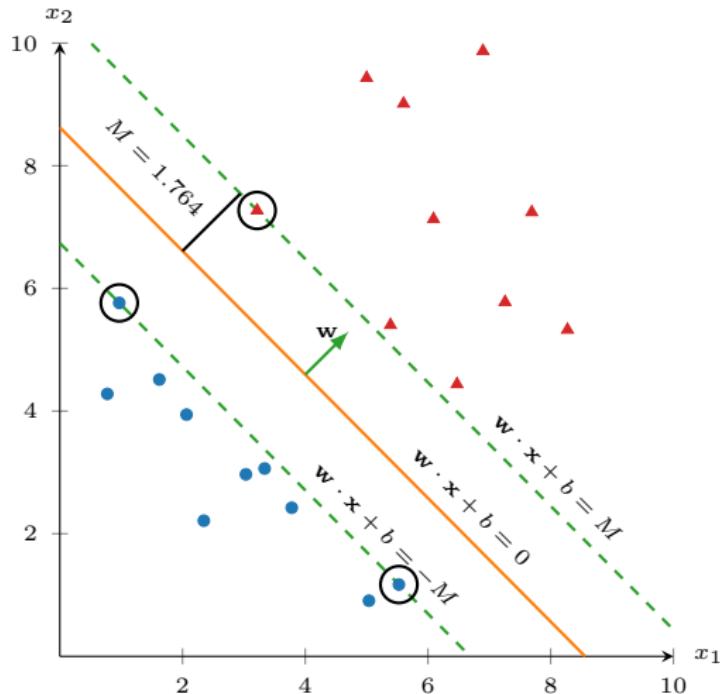
Improved formulation:

$$\begin{aligned} \max_{\mathbf{w}, b} \quad & M \\ \text{s.t.} \quad & y_i (\mathbf{w} \cdot \mathbf{x}_i + b) \geq M \quad \forall i \\ & \sum_{j=1}^p w_j^2 = 1 \\ & M \in \mathbb{R} \end{aligned}$$

For an optimal solution (\mathbf{w}, b) with $M > 0$:

- First set of constraints ensures that (\mathbf{w}, b) is a separating hyperplane
- Second set of constraints ensures that $\|\mathbf{w}\|_2 = 1$
- Maximizing M ensures that the hyperplane's margin is equal to M

Picture



$$w_1 = 0.7096, w_2 = 0.7045, b = -6.0754$$

- Classification is done using $h_{\mathbf{w},b}(\mathbf{x}) = \text{sgn}(\mathbf{w} \cdot \mathbf{x} + b)$
- The separating hyperplane is the solid yellow line
- The margins are the dashed green lines on either side of the separating hyperplane
- The support vectors are the points on the margin (circled)

Lecture 06-3c: Reformulating the Optimization Problem

Alternate Constraints on the Hyperplane

Recall that the hyperplane margin is given by:

$$\gamma(\mathbf{w}, b) = \min_{i=1}^N \frac{|\mathbf{w} \cdot \mathbf{x}_i + b|}{\|\mathbf{w}\|_2} = \frac{1}{\|\mathbf{w}\|_2} \min_{i=1}^N |\mathbf{w} \cdot \mathbf{x}_i + b|$$

Option 1: If we constrain \mathbf{w} to be a unit vector with $\|\mathbf{w}\|_2 = 1$, then the margin formula simplifies to:

$$\gamma(\mathbf{w}, b) = \min_{i=1}^N |\mathbf{w} \cdot \mathbf{x}_i + b|.$$

The resulting optimization problem is:

$$\begin{aligned} & \max_{\mathbf{w}, b} M \\ \text{s.t. } & y_i (\mathbf{w} \cdot \mathbf{x}_i + b) \geq M \quad \forall i \\ & \sum_{j=1}^p w_j^2 = 1 \\ & M \in \mathbb{R} \end{aligned}$$

Option 2: We could instead constrain \mathbf{w} by requiring that $\min_{i=1}^N |\mathbf{w} \cdot \mathbf{x}_i + b| = 1$, which simplifies the margin formula to:

$$\gamma(\mathbf{w}, b) = \frac{1}{\|\mathbf{w}\|_2}$$

The resulting optimization problem is:

$$\begin{aligned} & \max_{\mathbf{w}, b} \frac{1}{\|\mathbf{w}\|_2} \\ \text{s.t. } & y_i (\mathbf{w} \cdot \mathbf{x}_i + b) \geq 0 \quad \forall i \\ & \min_{i=1}^N |\mathbf{w} \cdot \mathbf{x}_i + b| = 1 \end{aligned}$$

Reformulating the Optimization Problem

In formulation 2, maximizing $\frac{1}{\|\mathbf{w}\|_2}$ is equivalent to minimizing $\|\mathbf{w}\|_2$.

As $g(t) = t^2$ is monotonically increasing for $t \geq 0$, the \mathbf{w} that minimizes $\|\mathbf{w}\|_2^2$ also minimizes $\|\mathbf{w}\|_2$, and $\|\mathbf{w}\|_2^2 = \sqrt{\mathbf{w} \cdot \mathbf{w}}^2 = \mathbf{w} \cdot \mathbf{w}$.

With some additional work, we can rewrite the optimization problem

$$\begin{array}{lll} \max_{\mathbf{w}, b} & \frac{1}{\|\mathbf{w}\|_2} & \min_{\mathbf{w}, b} \mathbf{w} \cdot \mathbf{w} \\ \text{s.t. } & y_i (\mathbf{w} \cdot \mathbf{x}_i + b) \geq 0 \quad \forall i & \text{s.t. } y_i (\mathbf{w} \cdot \mathbf{x}_i + b) \geq 1 \quad \forall i \\ & \min_{i=1}^N |\mathbf{w} \cdot \mathbf{x}_i + b| = 1 & \end{array}$$

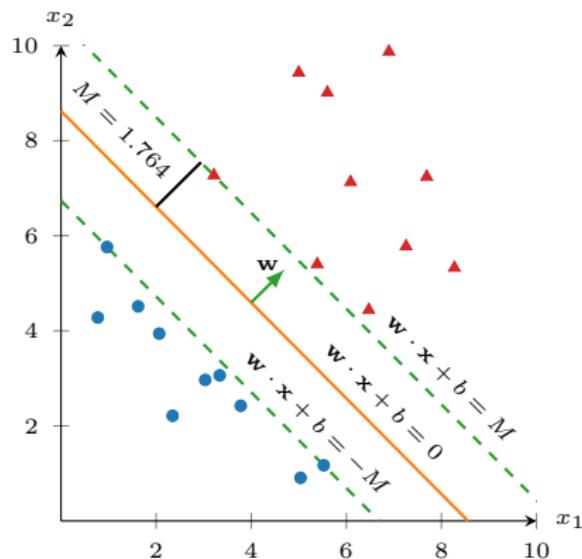
This leads to a **simpler** quadratic optimization problem!

- Convex objective function, linear constraints

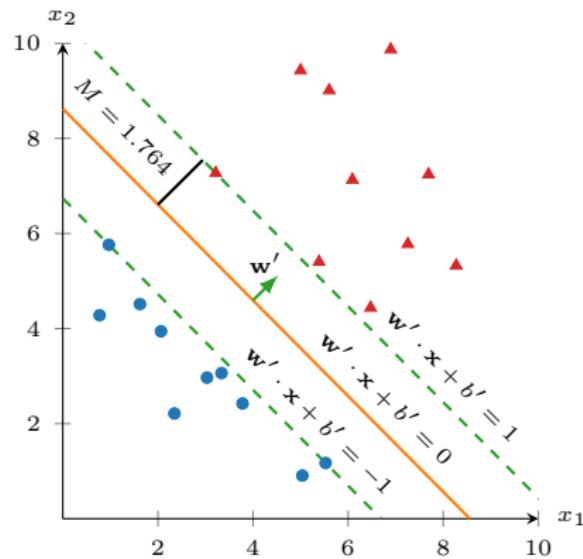
We'll revisit this a bit later...

Pictures

Option 1: Constraining \mathbf{w} to be a unit vector (i.e., $\|\mathbf{w}\|_2 = 1$):



Option 2: Constraining margin lines to satisfy $\mathbf{w}' \cdot \mathbf{x} + b' = \pm 1$:



$$w_1 = 0.7096, w_2 = 0.7045, b = -6.0754$$

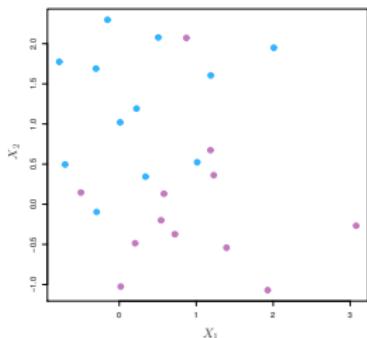
$$w'_1 = 0.5343, w'_2 = 0.5303, b' = -4.5735$$

Lecture 06-3d: Using Soft Margins

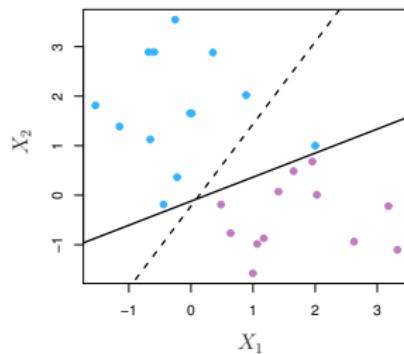
Issues with Hard Margins

Two issues with using **maximum margin separating hyperplanes**:

Data may not be linearly separable:



Data may require a very **narrow** margin:



Relaxing the requirement of exact classification leads to:

- Greater robustness to individual observations
- Better classification on *most* training observations
(where “better” is measured by distance from the hyperplane)

Pictures taken from “An Introduction to Statistical Learning, with applications in R” (Springer, 2013) with permission from the authors: G. James, D. Witten, T. Hastie and R. Tibshirani.

Soft Margins and the Support Vector Classifier

We relax the exact classification requirement by adding nonnegative **slack variables** ϵ_i for each data point that allow for violating the margin, along with a total violation budget B (nonnegative hyperparameter):

$$\min_{\mathbf{w}, b} \mathbf{w} \cdot \mathbf{w}$$

$$\text{s.t. } y_i (\mathbf{w} \cdot \mathbf{x}_i + b) \geq 1 - \epsilon_i \quad \forall i$$

$$\sum_{i=1}^N \epsilon_i \leq B$$

$$\epsilon_i \geq 0 \quad \forall i$$

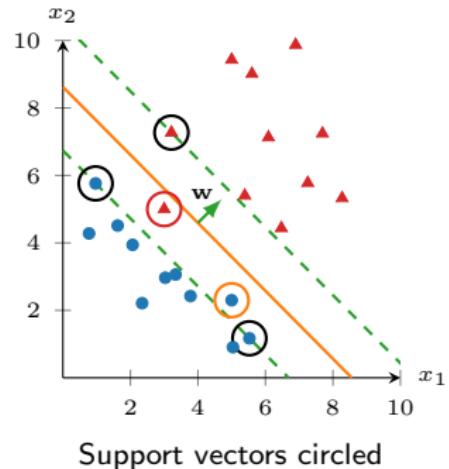
This results in the **support vector classifier (soft margin classifier)**.

Interpreting the Variables

Solving the optimization problem yields values for w , b and ϵ_i .

Interpretation of ϵ_i :

- If $\epsilon_i = 0$:
 x_i is on the **right** side of the margin
- If $\epsilon_i > 0$:
 x_i is on the **wrong** side of the margin
(x_i **violates** the margin)
 - If $\epsilon_i < 1$:
 x_i is on the **right** side of the hyperplane (classified **correctly**)
 - If $\epsilon_i > 1$:
 x_i is on the **wrong** side of the hyperplane (classified **incorrectly**)



Points that are on the margin or that violate the margin are the **support vectors**.

Understanding the Budget Hyperparameter

The optimization problem uses the budget hyperparameter B :

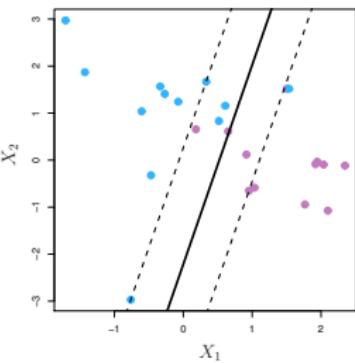
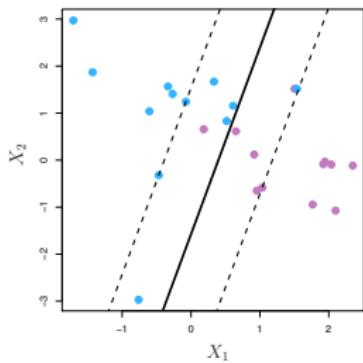
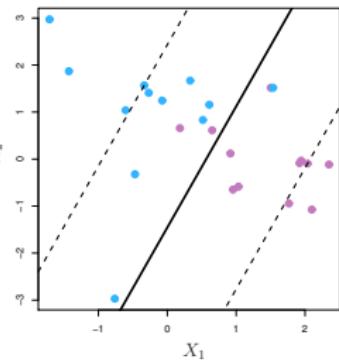
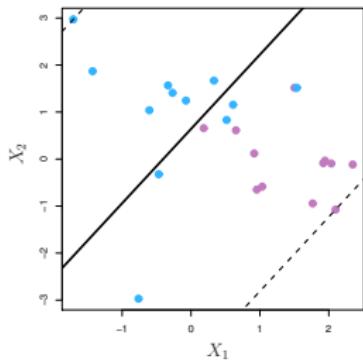
$$\sum_{i=1}^N \epsilon_i \leq B$$

B limits the number and severity of margin violations.

- $B = 0$ yields hard margin case
- $B > 0$: No more than B points can be **incorrectly classified**
 - With large B : wider margin, more support vectors
(high bias, low variance)
 - With small B : narrower margin, fewer support vectors
(low bias, high variance)

We can pick the **right** value for B using cross-validation.

Pictures



Support vector classifiers with different budgets:

- Top left: very high budget
- Top right: high budget
- Bottom left: moderate budget
- Bottom right: small budget

Taken from "An Introduction to Statistical Learning, with applications in R" (Springer, 2013) with permission from the authors:
G. James, D. Witten, T. Hastie and
R. Tibshirani.

Lecture 06-3e: More Reformulations of the Optimization Problem

Another Reformulation: Penalizing Violations

Instead of imposing a **violation limit** with a budget hyperparameter B

$$\sum_{i=1}^N \epsilon_i \leq B$$

as a constraint, we can instead **penalize violations** in the objective function using a cost hyperparameter C :

$$\begin{aligned} & \min_{\mathbf{w}, b} \mathbf{w} \cdot \mathbf{w} + C \sum_{i=1}^N \epsilon_i \\ \text{s.t. } & y_i (\mathbf{w} \cdot \mathbf{x}_i + b) \geq 1 - \epsilon_i \quad \forall i \\ & \epsilon_i \geq 0 \quad \forall i \end{aligned}$$

In general, having fewer constraints is better!

Solving for the Slack Variables ϵ_i

Rearranging the first constraint in the optimization problem yields:

$$\begin{aligned} \min_{\mathbf{w}, b} \quad & \mathbf{w} \cdot \mathbf{w} + C \sum_{i=1}^N \epsilon_i \\ \text{s.t.} \quad & \epsilon_i \geq 1 - y_i (\mathbf{w} \cdot \mathbf{x}_i + b) \quad \forall i \\ & \epsilon_i \geq 0 \quad \forall i \end{aligned}$$

With $C \neq 0$, we want to **minimize** each ϵ_i .

If we set

$$\epsilon_i = \max\{1 - y_i(\mathbf{w} \cdot \mathbf{x}_i + b), 0\},$$

then we **satisfy the constraints** while **minimizing** the violation cost.

The above choice for slack variables is **optimal** for any given \mathbf{w} and b !

An Unconstrained Optimization Problem

With the optimal choice of $\epsilon_i = \max\{1 - y_i(\mathbf{w} \cdot \mathbf{x}_i + b), 0\}$, we can reformulate the previous optimization problem as an **unconstrained** problem:

$$\min_{\mathbf{w}, b} \mathbf{w} \cdot \mathbf{w} + C \sum_{i=1}^N \max\{1 - y_i(\mathbf{w} \cdot \mathbf{x}_i + b), 0\}.$$

Rearranging slightly yields:

$$\min_{\mathbf{w}, b} C \sum_{i=1}^N \max\{1 - y_i(\mathbf{w} \cdot \mathbf{x}_i + b), 0\} + \sum_{j=1}^p w_j^2.$$

That is, our objective function is a cost function combining the total hinge loss on the training set with L_2 regularization of the weights!

We can **solve** this problem using gradient descent!

CS 457/557: Machine Learning

Lecture 06-4: Maximum Margin Classifier: The Dual Formulation

Lecture 06-4a: The Primal Problem and Lagrangian Relaxation

Finding the Maximum Margin Classifier

Recall the optimization problem for finding the maximum margin separating hyperplane:

$$\begin{aligned} \min_{\mathbf{w}, b} \quad & \frac{1}{2} \mathbf{w} \cdot \mathbf{w} \\ \text{s.t.} \quad & y_i (\mathbf{w} \cdot \mathbf{x}_i + b) \geq 1 \quad \forall i \end{aligned}$$

- The constraints ensure that the hyperplane (\mathbf{w}, b) actually separates the data by class labels.
- The objective is to find the “simplest” vector \mathbf{w} that does this (with the $\frac{1}{2}$ just included for convenience).

For now, let's rewrite the constraints as

$$y_i (\mathbf{w} \cdot \mathbf{x}_i + b) - 1 \geq 0.$$

Lagrangian Relaxation

Definition

Lagrangian relaxation attempts to solve a relaxed version of a constrained optimization problem by removing the constraints and instead penalizing constraint violations in the objective.

So the prior problem becomes

$$\min_{\mathbf{w}, b} \frac{1}{2} \mathbf{w} \cdot \mathbf{w} - \sum_{i=1}^N \alpha_i [y_i(\mathbf{w} \cdot \mathbf{x}_i + b) - 1]$$

where α_i represents a non-negative **penalty** associated with constraint i .
The objective function of this optimization problem is called the
Lagrangian function, denoted

$$L(\mathbf{w}, b, \boldsymbol{\alpha}) = \frac{1}{2} \mathbf{w} \cdot \mathbf{w} - \sum_{i=1}^N \alpha_i [y_i(\mathbf{w} \cdot \mathbf{x}_i + b) - 1].$$

Setting the Penalties

The penalties α get set via a **two-stage process**:

- ① We pick w, b to try to **minimize** $L(w, b, \alpha)$
 - There are **no constraints** on w, b for us
 - But we **don't know the penalties yet!**
- ② Then the **adversary** picks the penalties α to **maximize** $L(w, b, \alpha)$ given our choice of w, b

What would the adversary do?

- If $[y_i(w \cdot x_i + b) - 1] \geq 0$:
 Constraint i is **satisfied** (even though we didn't have to satisfy it)
 Then $L(w, b, \alpha)$ **decreases** with $\alpha_i > 0$, so **adversary** picks $\alpha_i = 0$
- If $[y_i(w \cdot x_i + b) - 1] < 0$:
 Constraint i is **violated** (so we are trying to "cheat" here)
 Then $L(w, b, \alpha)$ **increases** with $\alpha_i > 0$, so **adversary** picks $\alpha_i = +\infty$

The Two-Stage Optimization

For any choice of (\mathbf{w}, b) , the **adversary's** strategy of maximizing $L(\mathbf{w}, b, \alpha)$ yields:

$$\max_{\alpha \geq 0} L(\mathbf{w}, b, \alpha) = \begin{cases} +\infty & \text{if } (\mathbf{w}, b) \text{ violates any constraint} \\ \frac{1}{2}\mathbf{w} \cdot \mathbf{w} & \text{if } (\mathbf{w}, b) \text{ satisfies all constraints} \end{cases}$$

That is, the **adversary** “keeps us honest” when picking (\mathbf{w}, b) , because if we choose to violate a constraint, then we pay a huge penalty.

So the **unconstrained** optimization problem

$$\min_{\mathbf{w}, b} \max_{\alpha \geq 0} L(\mathbf{w}, b, \alpha)$$

captures the requirements of the earlier **constrained** optimization problem.

Changing the Order of Play

It turns out that $\min_{\mathbf{w}, b} \max_{\boldsymbol{\alpha} \geq \mathbf{0}} L(\mathbf{w}, b, \boldsymbol{\alpha})$ is **still a hard problem!**

But what if the **adversary** had to **go first**?

$$\max_{\boldsymbol{\alpha} \geq \mathbf{0}} \min_{\mathbf{w}, b} L(\mathbf{w}, b, \boldsymbol{\alpha})$$

- ① The **adversary** picks the penalties $\boldsymbol{\alpha}$ to try to **maximize** $L(\mathbf{w}, b, \boldsymbol{\alpha})$, without knowing our choice of \mathbf{w}, b !
- ② Then **we** pick (\mathbf{w}, b) to **minimize** $L(\mathbf{w}, b, \boldsymbol{\alpha})$

For our particular objective, Slater's condition ensures that these problems are **equivalent**!

Using Penalties to Make Our Move

Let's think about our move *after* the **adversary** picks α . We want to pick (\mathbf{w}, b) that **minimize** $L(\mathbf{w}, b, \alpha)$, given by

$$\frac{1}{2}\mathbf{w} \cdot \mathbf{w} - \sum_{i=1}^N \alpha_i [y_i(\mathbf{w} \cdot \mathbf{x}_i + b) - 1]$$

For the constraint $y_i(\mathbf{w} \cdot \mathbf{x}_i + b) - 1 \geq 0$:

- If $\alpha_i = 0$:
There is **no penalty** to us for picking (\mathbf{w}, b) that **violate** this constraint!
- If $\alpha_i > 0$:
We pay a **penalty** for **violating** this constraint.
But we also get a **benefit** for satisfying the constraint **with slack!**

Given this, how should the **adversary** pick the penalties?

Intuition for the Adversary's Move

The **adversary** needs to “price” the constraints appropriately to ensure that:

- we can't get any benefit by **violating a constraint**
- we can't get any benefit by **satisfying a constraint** with slack

Intuition:

- Constraints that we wouldn't ordinarily have violated have a price of 0 (to avoid giving us a benefit by **satisfying those constraints with slack**)
- Constraints that we would *like to violate* must have a sufficiently large penalty to ensure that we **don't** violate them
 - But the penalty can't be **so high** that we would benefit by **satisfying the constraint with slack** (so the **adversary** can't pick $+\infty$ for penalties now!)

Lecture 06-4b: Towards a Solution

Figuring Out Our Move

By making the adversary go first, we get the problem

$$\max_{\alpha \geq 0} \min_{w,b} L(w, b, \alpha).$$

Let's think about how we can determine **our move** for any given choice of α by the adversary. Our problem simplifies to

$$\min_{w,b} L(w, b, \alpha)$$

with α fixed. How do we find the **optimal solution** (w^*, b^*) to this problem? Analytically!

- Compute partial derivatives of $L(w, b, \alpha)$ with respect to the variables w and b
- Set the partial derivatives equal to zero and solve for w^* and b^*

Expanding the Lagrangian

First let's expand the Lagrangian function to make it easier to see the partial derivatives:

$$\begin{aligned} L(\mathbf{w}, b, \boldsymbol{\alpha}) &= \frac{1}{2} \mathbf{w} \cdot \mathbf{w} - \sum_{i=1}^N \alpha_i [y_i (\mathbf{w} \cdot \mathbf{x}_i + b) - 1] \\ &= \frac{1}{2} \sum_{j=1}^p w_j^2 - \sum_{i=1}^N \alpha_i \left[y_i \left(\sum_{j=1}^p w_j x_{ij} + b \right) - 1 \right] \\ &= \frac{1}{2} \sum_{j=1}^p w_j^2 - \sum_{i=1}^N \left[\alpha_i y_i \left(\sum_{j=1}^p w_j x_{ij} + b \right) - \alpha_i \right] \\ &= \frac{1}{2} \sum_{j=1}^p w_j^2 - \sum_{i=1}^N \left[\sum_{j=1}^p \alpha_i y_i w_j x_{ij} + \alpha_i y_i b - \alpha_i \right] \\ &= \frac{1}{2} \sum_{j=1}^p w_j^2 - \sum_{i=1}^N \sum_{j=1}^p \alpha_i y_i w_j x_{ij} - \sum_{i=1}^N \alpha_i y_i b + \sum_{i=1}^N \alpha_i \end{aligned}$$

Partial Derivatives

The partial derivatives of $L(\mathbf{w}, b, \boldsymbol{\alpha})$ are

$$\begin{aligned}\frac{\partial}{\partial w_k} L(\mathbf{w}, b, \boldsymbol{\alpha}) &= \frac{\partial}{\partial w_k} \left[\frac{1}{2} \sum_{j=1}^p w_j^2 - \sum_{i=1}^N \sum_{j=1}^p \alpha_i y_i w_j x_{ij} - \sum_{i=1}^N \alpha_i y_i b + \sum_{i=1}^N \alpha_i \right] \\ &= w_k - \sum_{i=1}^N \alpha_i y_i x_{ik}\end{aligned}$$

and

$$\begin{aligned}\frac{\partial}{\partial b} L(\mathbf{w}, b, \boldsymbol{\alpha}) &= \frac{\partial}{\partial b} \left[\frac{1}{2} \sum_{j=1}^p w_j^2 - \sum_{i=1}^N \sum_{j=1}^p \alpha_i y_i w_j x_{ij} - \sum_{i=1}^N \alpha_i y_i b + \sum_{i=1}^N \alpha_i \right] \\ &= - \sum_{i=1}^N \alpha_i y_i.\end{aligned}$$

Solution

Setting the partial derivatives of $L(\mathbf{w}, b, \alpha) = 0$ and simplifying yields

$$\frac{\partial}{\partial w_k} L(\mathbf{w}, b, \alpha) = 0 \Leftrightarrow w_k - \sum_{i=1}^N \alpha_i y_i x_{ik} = 0 \Leftrightarrow w_k = \sum_{i=1}^N \alpha_i y_i x_{ik}$$

$$\frac{\partial}{\partial w_b} L(\mathbf{w}, b, \alpha) = 0 \Leftrightarrow - \sum_{i=1}^N \alpha_i y_i = 0 \Leftrightarrow \sum_{i=1}^N \alpha_i y_i = 0.$$

So for any given α , we see that the optimal solution to $\min_{\mathbf{w}, b} L(\mathbf{w}, b, \alpha)$ is given partially by $\mathbf{w}^* = \sum_{i=1}^N \alpha_i y_i \mathbf{x}_i$. We **don't** have an explicit solution for b^* , though, just an equation that says $\sum_{i=1}^N \alpha_i y_i = 0$. What does this mean?

- The α_i values were already picked ahead of time by the **adversary**, so by the time **we** choose \mathbf{w} and b , there's no way to satisfy $\sum_{i=1}^N \alpha_i y_i = 0$ if it isn't already satisfied.
- By recognizing that $y_i \in \{-1, 1\}$ for all i , we can rewrite this constraint as $\sum_{i:y_i=1} \alpha_i = \sum_{i:y_i=-1} \alpha_i$. I.e., the sums of the penalties for positive and negative examples must be equal.
- Remember, we **pay a penalty** proportional to α_i for any violation of the constraint, but we also **get a benefit** proportional to α_i for any slack in the constraint.
- Intuitively, if the **adversary** didn't choose α to make this happen, then we could keep decreasing $L(\mathbf{w}, b, \alpha)$ by moving the hyperplane further and further away from the class which provides greater benefit for slack!

Simplifying the Lagrangian

If we substitute our optimal choice of $\mathbf{w}^* = \sum_{i=1}^N \alpha_i y_i \mathbf{x}_i$ into $L(\mathbf{w}, b, \boldsymbol{\alpha})$, we get:

$$\begin{aligned} L(\mathbf{w}^*, b, \boldsymbol{\alpha}) &= \frac{1}{2} \mathbf{w}^* \cdot \mathbf{w}^* - \sum_{i=1}^N \alpha_i [y_i (\mathbf{w}^* \cdot \mathbf{x}_i + b) - 1] \\ &= \frac{1}{2} \left(\sum_{i=1}^N \alpha_i y_i \mathbf{x}_i \right) \cdot \left(\sum_{i=1}^N \alpha_i y_i \mathbf{x}_i \right) - \sum_{i=1}^N \alpha_i \left[y_i \left(\left(\sum_{i'=1}^N \alpha_{i'} y_{i'} \mathbf{x}_{i'} \right) \cdot \mathbf{x}_i + b \right) - 1 \right] \\ &= \frac{1}{2} \left(\sum_{i=1}^N \sum_{i'=1}^N \alpha_i \alpha_{i'} y_i y_{i'} \mathbf{x}_i \cdot \mathbf{x}_{i'} \right) - \sum_{i=1}^N \left[\alpha_i y_i \left(\sum_{i'=1}^N \alpha_{i'} y_{i'} \mathbf{x}_{i'} \cdot \mathbf{x}_i + b \right) - \alpha_i \right] \\ &= \frac{1}{2} \left(\sum_{i=1}^N \sum_{i'=1}^N \alpha_i \alpha_{i'} y_i y_{i'} \mathbf{x}_i \cdot \mathbf{x}_{i'} \right) - \sum_{i=1}^N \left[\sum_{i'=1}^N \alpha_i y_i \alpha_{i'} y_{i'} \mathbf{x}_{i'} \cdot \mathbf{x}_i + \alpha_i y_i b - \alpha_i \right] \\ &= \frac{1}{2} \left(\sum_{i=1}^N \sum_{i'=1}^N \alpha_i \alpha_{i'} y_i y_{i'} \mathbf{x}_i \cdot \mathbf{x}_{i'} \right) - \left(\sum_{i=1}^N \sum_{i'=1}^N \alpha_i y_i \alpha_{i'} y_{i'} \mathbf{x}_i \cdot \mathbf{x}_{i'} \right) - \sum_{i=1}^N \alpha_i y_i b + \sum_{i=1}^N \alpha_i \\ &= \sum_{i=1}^N \alpha_i - \frac{1}{2} \left(\sum_{i=1}^N \sum_{i'=1}^N \alpha_i \alpha_{i'} y_i y_{i'} \mathbf{x}_i \cdot \mathbf{x}_{i'} \right) - b \sum_{i=1}^N \alpha_i y_i. \end{aligned}$$

The last sum drops out assuming that $\sum_{i=1}^N \alpha_i y_i = 0$.

Putting It All Together

With the “adversary goes first” problem of

$$\max_{\alpha \geq 0} \min_{w,b} L(w, b, \alpha),$$

we see that if the adversary picks α to satisfy $\sum_{i=1}^N \alpha_i y_i = 0$ and we always pick w optimally given the adversary’s choice of α , then the **adversary’s** problem becomes the following:

$$\begin{aligned} & \max \sum_{i=1}^N \alpha_i - \frac{1}{2} \sum_{i=1}^N \sum_{i'=1}^N \alpha_i \alpha_{i'} y_i y_{i'} \mathbf{x}_i \cdot \mathbf{x}_{i'} \\ \text{s.t. } & \sum_{i=1}^N \alpha_i y_i = 0 \\ & \alpha_i \geq 0 \quad \forall i \end{aligned}$$

Lecture 06-4c: The Dual Problem

The Dual Problem

The adversary's problem is called the **dual problem**:

$$\begin{aligned} \max \quad & \sum_{i=1}^N \alpha_i - \frac{1}{2} \sum_{i=1}^N \sum_{i'=1}^N \alpha_i \alpha_{i'} y_i y_{i'} \mathbf{x}_i \cdot \mathbf{x}_{i'} \\ \text{s.t.} \quad & \sum_{i=1}^N \alpha_i y_i = 0 \\ & \alpha_i \geq 0 \quad \forall i \end{aligned}$$

This problem gives us an **alternate** way to find the maximum margin separating hyperplane, specified in terms of its **support vectors**!

- $\alpha_i > 0$ implies that \mathbf{x}_i is a support vector
- $\alpha_i = 0$ implies that \mathbf{x}_i is **not** a support vector

Also, we **only need** dot products of pairs of feature vectors to solve!

Recovering the Hyperplane Parameters

Suppose we solved the **dual problem** and got the penalties α . How can we determine the hyperplane (\mathbf{w}, b) itself?

Earlier we had seen that:

$$\mathbf{w} = \sum_{i=1}^N \alpha_i y_i \mathbf{x}_i$$

In most cases, the number of support vectors is small, so many terms in the sum will have $\alpha_i = 0$. The remaining terms then give \mathbf{w} as a **linear combination** of the support vectors!

Determining b takes some more work. Let's look at an index i with $\alpha_i > 0$, which implies that the constraint $y_i(\mathbf{w} \cdot \mathbf{x}_i + b) \geq 1$ is tight. Then:

$$\begin{aligned} y_i(\mathbf{w} \cdot \mathbf{x}_i + b) = 1 &\Leftrightarrow y_i y_i (\mathbf{w} \cdot \mathbf{x}_i + b) = y_i \\ &\Leftrightarrow \mathbf{w} \cdot \mathbf{x}_i + b = y_i \\ &\Leftrightarrow b = y_i - \mathbf{w} \cdot \mathbf{x}_i. \end{aligned}$$

(In practice we would want to compute b by averaging of $y_i - \mathbf{w} \cdot \mathbf{x}_i$ from several support vectors to mitigate numerical issues.)

Classification with the Dual Problem

When solving for (\mathbf{w}, b) directly, we classified new data using

$$h_{\mathbf{w}, b}(\mathbf{x}) = \text{sgn}(\mathbf{w} \cdot \mathbf{x} + b).$$

If we solve for α and compute b , we can classify new data using

$$\begin{aligned} h_{\alpha, b}(\mathbf{x}) &= \text{sgn} \left(\left(\sum_{i=1}^N \alpha_i y_i \mathbf{x}_i \right) \cdot \mathbf{x} + b \right) \\ &= \text{sgn} \left(\sum_{i=1}^N \alpha_i y_i \mathbf{x}_i \cdot \mathbf{x} + b \right) \\ &= \text{sgn} \left(\sum_{i \in S} \alpha_i y_i \mathbf{x}_i \cdot \mathbf{x} + b \right) \end{aligned}$$

where S is the set of support vectors.

The Maximum Margin Separating Hyperplane: Primal and Dual Problems

The Primal Problem:

$$\begin{aligned} \min_{\mathbf{w}, b} \quad & \frac{1}{2} \mathbf{w} \cdot \mathbf{w} \\ \text{s.t.} \quad & y_i (\mathbf{w} \cdot \mathbf{x}_i + b) \geq 1 \quad \forall i \end{aligned}$$

The Dual Problem:

$$\begin{aligned} \max \quad & \sum_{i=1}^N \alpha_i - \frac{1}{2} \sum_{i=1}^N \sum_{i'=1}^N \alpha_i \alpha_{i'} y_i y_{i'} \mathbf{x}_i \cdot \mathbf{x}_{i'} \\ \text{s.t.} \quad & \sum_{i=1}^N \alpha_i y_i = 0 \\ & \alpha_i \geq 0 \quad \forall i \end{aligned}$$

- $p+1$ variables, N constraints
- Primal classifier:

$$h_{\mathbf{w}, b}(\mathbf{x}) = \operatorname{sgn}(\mathbf{w} \cdot \mathbf{x} + b)$$

- Parametric: stores $p+1$ weights

- N variables, $N+1$ constraints
- Dual classifier:

$$h_{\boldsymbol{\alpha}, b}(\mathbf{x}) = \operatorname{sgn}\left(\sum_{i \in \mathcal{S}} \alpha_i y_i \mathbf{x}_i \cdot \mathbf{x} + b\right)$$

- Nonparametric: stores support vectors

Usefulness of the Dual Problem

Some useful properties associated with the dual:

- Solutions tend to be sparse (low number of support vectors), making the classifier representation compact
- Only uses dot products of feature vectors in calculations
 - Remember: a dot product gives a scalar value, regardless of how many features are in the feature vectors!
- Assuming the dot products of feature vectors are pre-computed, the dual itself depends only on the number of training examples, **not** on the number of features!

We'll see why this is so helpful shortly!

CS 457/557: Machine Learning

Lecture 06-5: Support Vector Machines

Lecture 06-5a: Primal and Dual Problems

Quick Recap: Maximum Margin Primal and Dual Problems

The Primal Problem:

$$\begin{aligned} \min_{\mathbf{w}, b} \quad & \frac{1}{2} \mathbf{w} \cdot \mathbf{w} \\ \text{s.t.} \quad & y_i (\mathbf{w} \cdot \mathbf{x}_i + b) \geq 1 \quad \forall i \end{aligned}$$

The Dual Problem:

$$\begin{aligned} \max \quad & \sum_{i=1}^N \alpha_i - \frac{1}{2} \sum_{i=1}^N \sum_{i'=1}^N \alpha_i \alpha_{i'} y_i y_{i'} \mathbf{x}_i \cdot \mathbf{x}_{i'} \\ \text{s.t.} \quad & \sum_{i=1}^N \alpha_i y_i = 0 \\ & \alpha_i \geq 0 \quad \forall i \end{aligned}$$

- $p+1$ variables, N constraints
- Primal classifier:

$$h_{\mathbf{w}, b}(\mathbf{x}) = \operatorname{sgn}(\mathbf{w} \cdot \mathbf{x} + b)$$

- Parametric: stores $p+1$ weights

- N variables, $N+1$ constraints
- Dual classifier:

$$h_{\boldsymbol{\alpha}, b}(\mathbf{x}) = \operatorname{sgn}\left(\sum_{i=1}^N \alpha_i y_i \mathbf{x}_i \cdot \mathbf{x} + b\right)$$

- Nonparametric: stores support vectors

Either problem can be solved with quadratic optimization techniques.

The Support Vector Classifier: Adding Soft Margins

The Primal Problem:

$$\begin{aligned} \min_{\mathbf{w}, b} \quad & \frac{1}{2} \mathbf{w} \cdot \mathbf{w} + C \sum_{i=1}^N \epsilon_i \\ \text{s.t.} \quad & y_i (\mathbf{w} \cdot \mathbf{x}_i + b) \geq 1 - \epsilon_i \quad \forall i \\ & \epsilon_i \geq 0 \quad \forall i \end{aligned}$$

The Dual Problem:

$$\begin{aligned} \max \quad & \sum_{i=1}^N \alpha_i - \frac{1}{2} \sum_{i=1}^N \sum_{i'=1}^N \alpha_i \alpha_{i'} y_i y_{i'} \mathbf{x}_i \cdot \mathbf{x}_{i'} \\ \text{s.t.} \quad & \sum_{i=1}^N \alpha_i y_i = 0 \\ & 0 \leq \alpha_i \leq C \quad \forall i \end{aligned}$$

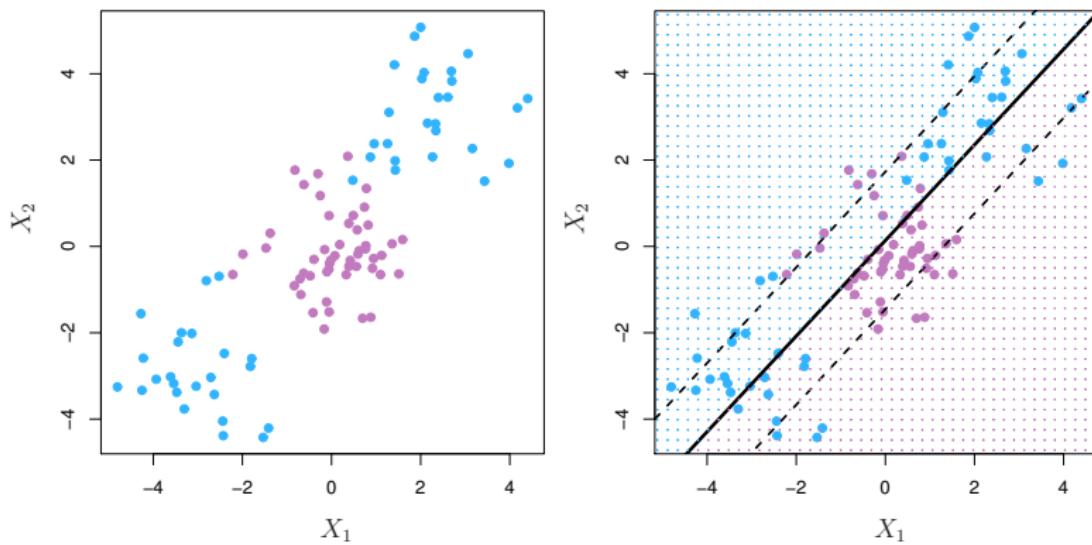
We'll skip the derivation of the dual here; the only difference in the formulation is the upper bound of C on each penalty term α_i .

- Recall: In the hard margin setting with us going first, the adversary could drive $L(\mathbf{w}, b, \alpha)$ to $+\infty$ if there was a violated constraint
- In the soft margin setting, the maximum penalty for violating the margin is limited to C

(In soft margin setting, when calculating b from α , pick i with $0 < \alpha_i < C$)

Lecture 06-5b: Dealing with Nonlinear Data

Issues with Nonlinear Data



Taken from "An Introduction to Statistical Learning, with applications in R" (Springer, 2013)
with permission from the authors: G. James, D. Witten, T. Hastie and R. Tibshirani.

Dealing with Nonlinear Data

One general way to deal with **nonlinear data** is to create **new derived features** representing higher-order and interaction terms, e.g.:

- Transform raw feature X_1 into X_1^2
- Create interaction $X_1 X_2$

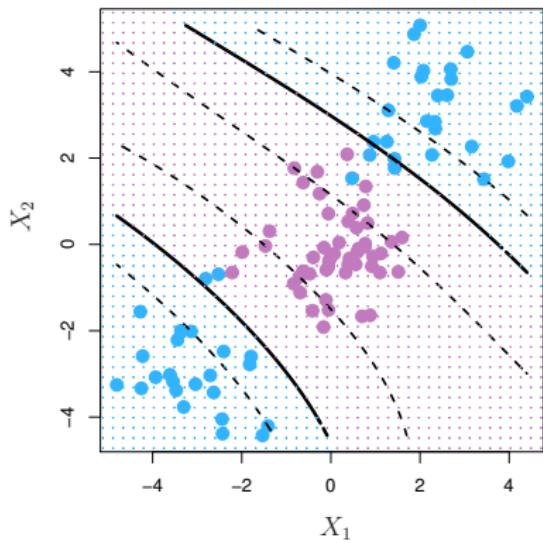
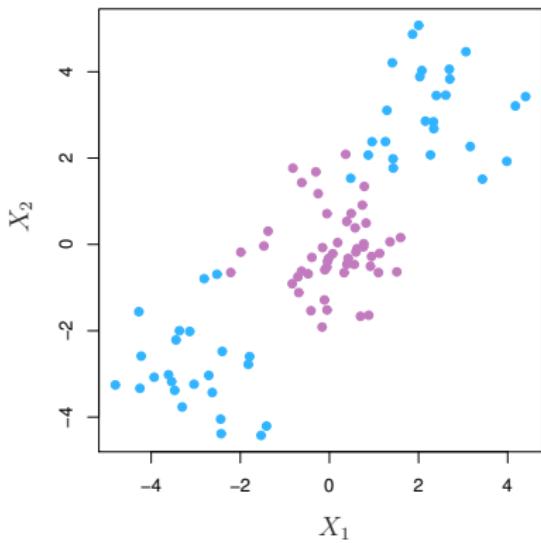
We can represent this transformation using a **feature map** function $\phi : \mathbb{R}^p \rightarrow \mathbb{R}^P$, where p is the number of raw attributes and P is the number of derived attributes. E.g.,

$$\mathbf{x} = (x_1, x_2, \dots, x_p), \quad \phi(\mathbf{x}) = (x_1, x_2, \dots, x_p, x_1^2, x_2^2, \dots, x_p^2)$$

Then we look for a **separating hyperplane** in the enlarged feature space!

- The separator is linear in the **enlarged feature space**
- The separator looks nonlinear in the **original feature space**

A Separating Hyperplane with Higher-Order Terms



Taken from "An Introduction to Statistical Learning, with applications in R" (Springer, 2013)
with permission from the authors: G. James, D. Witten, T. Hastie and R. Tibshirani.

Optimization with Derived Features

With $\phi : \mathbb{R}^p \rightarrow \mathbb{R}^P$, the soft margin optimization problems are:

The Primal Problem:

$$\begin{aligned} \min_{\mathbf{w}, b} \quad & \frac{1}{2} \mathbf{w} \cdot \mathbf{w} + C \sum_{i=1}^N \epsilon_i \\ \text{s.t.} \quad & y_i (\mathbf{w} \cdot \phi(\mathbf{x}_i) + b) \geq 1 - \epsilon_i \quad \forall i \\ & \epsilon_i \geq 0 \quad \forall i \end{aligned}$$

The Dual Problem:

$$\begin{aligned} \max \quad & \sum_{i=1}^N \alpha_i - \frac{1}{2} \sum_{i=1}^N \sum_{i'=1}^N \alpha_i \alpha_{i'} y_i y_{i'} \phi(\mathbf{x}_i) \cdot \phi(\mathbf{x}_{i'}) \\ \text{s.t.} \quad & \sum_{i=1}^N \alpha_i y_i = 0 \\ & 0 \leq \alpha_i \leq C \quad \forall i \end{aligned}$$

- $P + 1$ variables, $2N$ constraints
- N variables, $N + 1$ constraints

Despite **increasing** the number of features (assuming $P > p$), the size of the dual problem **doesn't change!**

- We still need $\phi(\mathbf{x}_i) \cdot \phi(\mathbf{x}_{i'})$ for all pairs i, i' , though
- A **naïve implementation** would require $O(P)$ work to compute this
- For an appropriate choice of ϕ , we can **actually do this** in $O(p)$ though!

Lecture 06-5c: Kernel Functions and Support Vector Machines

A Motivating Example

Suppose $p = 2$ and

$$\phi(\mathbf{x}) = \left(1, \sqrt{2}x_1, \sqrt{2}x_2, x_1^2, x_2^2, \sqrt{2}x_1x_2\right)$$

(i.e., we transform from $p = 2$ raw features to $P = 5$ derived features, plus a constant).

Then

$$\begin{aligned}\phi(\mathbf{x}_i) \cdot \phi(\mathbf{x}_{i'}) &= \left(1, \sqrt{2}x_{i1}, \sqrt{2}x_{i2}, x_{i1}^2, x_{i2}^2, \sqrt{2}x_{i1}x_{i2}\right) \\ &\quad \cdot \left(1, \sqrt{2}x_{i'1}, \sqrt{2}x_{i'2}, x_{i'1}^2, x_{i'2}^2, \sqrt{2}x_{i'1}x_{i'2}\right) \\ &= 1 + 2x_{i1}x_{i'1} + 2x_{i2}x_{i'2} + x_{i1}^2x_{i'1}^2 + x_{i2}^2x_{i'2}^2 + 2x_{i1}x_{i2}x_{i'1}x_{i'2} \\ &= (x_{i1}x_{i'1} + x_{i2}x_{i'2} + 1)(x_{i1}x_{i'1} + x_{i2}x_{i'2} + 1) \\ &= (x_{i1}x_{i'1} + x_{i2}x_{i'2} + 1)^2 \\ &= (\mathbf{x}_i \cdot \mathbf{x}_{i'} + 1)^2\end{aligned}$$

That is, we can compute the dot product of $\phi(\mathbf{x}_i)$ and $\phi(\mathbf{x}_{i'})$, both vectors with 6 components, by computing the dot product of \mathbf{x}_i and $\mathbf{x}_{i'}$, adding 1, and squaring it.

The Kernel Function

For an appropriate choice of ϕ , we represent $\phi(\mathbf{x}_i) \cdot \phi(\mathbf{x}_{i'})$ using a **kernel function** $K : \mathbb{R}^p \times \mathbb{R}^p \rightarrow \mathbb{R}$. With $p = 2$ and

$$\phi(\mathbf{x}) = \left(1, \sqrt{2}x_1, \sqrt{2}x_2, x_1^2, x_2^2, \sqrt{2}x_1x_2\right)$$

as used previously, we saw that

$$K(\mathbf{x}_i, \mathbf{x}_{i'}) = \phi(\mathbf{x}_i) \cdot \phi(\mathbf{x}_{i'}) = (\mathbf{x}_i \cdot \mathbf{x}_{i'} + 1)^2.$$

- The final expression (right side) is what actually gets used to compute dot products in our optimization problem
- Often we pre-compute these values and store them in a kernel matrix \mathbf{K}

Ideally, we want kernels that represent expressive feature maps ϕ and are also easy to calculate!

The Support Vector Machine

The **support vector machine** (SVM) is an extension of the support vector classifier that uses kernels to enlarge the feature space without needing to explicitly compute the derived feature vectors.

Optimization problem (dual):

$$\begin{aligned} \max \quad & \sum_{i=1}^N \alpha_i - \frac{1}{2} \sum_{i=1}^N \sum_{i'=1}^N \alpha_i \alpha_{i'} y_i y_{i'} K(\mathbf{x}_i, \mathbf{x}_{i'}) \\ \text{s.t.} \quad & \sum_{i=1}^N \alpha_i y_i = 0 \\ & 0 \leq \alpha_i \leq C \quad \forall i \end{aligned}$$

Classification process:

$$h_{\alpha,b}(\mathbf{x}) = \operatorname{sgn} \left(\sum_{i=1}^N \alpha_i y_i K(\mathbf{x}_i, \mathbf{x}) + b \right)$$

Common Kernels

- **Linear kernel:**

$$K(\mathbf{x}_i, \mathbf{x}_{i'}) = \mathbf{x}_i \cdot \mathbf{x}_{i'}$$

(just uses original features)

- **Polynomial kernel** of degree exactly d :

$$K(\mathbf{x}_i, \mathbf{x}_{i'}) = (\mathbf{x}_i \cdot \mathbf{x}_{i'})^d$$

associated with feature map using features of degree exactly d ;
e.g., with $p = 2$ and $d = 2$, we have $\phi(\mathbf{x}) = (x_1^2, x_2^2, \sqrt{2}x_1x_2)$

- **Polynomial kernel** of degree up to d :

$$K(\mathbf{x}_i, \mathbf{x}_{i'}) = (\mathbf{x}_i \cdot \mathbf{x}_{i'} + 1)^d$$

associated with feature map using features of degree at most d ;
e.g., with $p = 2$ and $d = 2$, we have $\phi(\mathbf{x}) = (1, \sqrt{2}x_1, \sqrt{2}x_2, x_1^2, x_2^2, \sqrt{2}x_1x_2)$

Mercer's Theorem: Any *reasonable* kernel function corresponds to *some* feature space (where reasonable means that the kernel matrix K is positive definite).

Appreciating the Power of Kernels

Let's think about the polynomial kernel $K(\mathbf{x}_i, \mathbf{x}_{i'}) = (\mathbf{x}_i \cdot \mathbf{x}_{i'} + 1)^d$.

- With p raw features, there are $O(p^d)$ derived features with degree at most d
- **Explicitly generating** these derived feature vectors would take $O(p^d)$ time
- However, we can compute $K(\mathbf{x}_i, \mathbf{x}_{i'})$ in $O(p)$ time!

The ability to operate on an enlarged feature space without needing to **explicitly construct** enlarged feature vectors is one of the **primary advantages** of the dual formulation, and is referred to as the **kernel trick**.

Lecture 06-5d: Gaussian Kernels

Some More Kernels

- **Gaussian kernel** (radial basis function):

$$K(\mathbf{x}_i, \mathbf{x}_{i'}) = \exp\left(-\frac{\|\mathbf{x}_i - \mathbf{x}_{i'}\|_2^2}{2\sigma^2}\right)$$

What does the underlying feature map look like?

With $p = 1$, we have

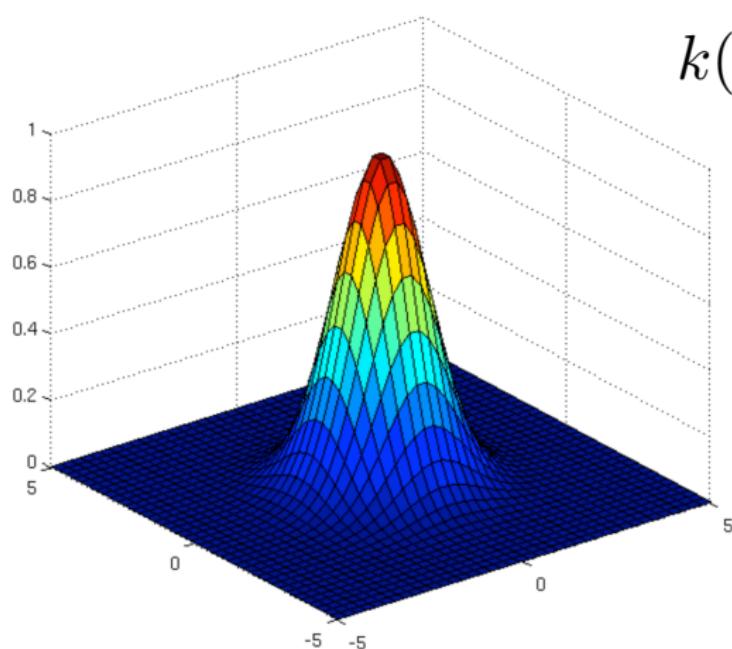
$$\phi(x) = \exp\left(-\frac{x^2}{2\sigma^2}\right) \left[1, \frac{x}{\sigma\sqrt{1!}}, \frac{x^2}{\sigma^2\sqrt{2!}}, \frac{x^3}{\sigma^3\sqrt{3!}}, \dots\right].$$

This is an **infinite-dimensional** feature vector!

So it is probably more helpful to think of $K(\mathbf{x}_i, \mathbf{x}_{i'})$ as returning a **similarity score** instead of doing implicit feature mapping.

Gaussian Radial Basis Function (RBF)

$$k(\mathbf{x}, \mathbf{z}) = e^{-\frac{\|\mathbf{x}-\mathbf{z}\|^2}{2\sigma^2}}$$

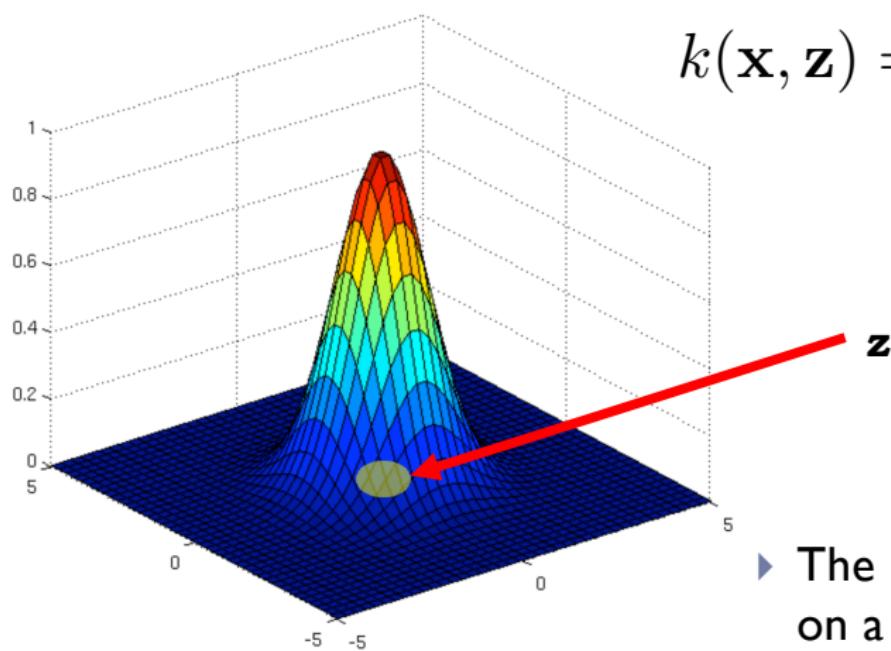


- ▶ A popular kernel with many uses is the **Gaussian RBF**

Image source: <https://www.cs.toronto.edu/~duvenaud/cookbook/>

Gaussian Radial Basis Function

$$k(\mathbf{x}, \mathbf{z}) = e^{-\frac{\|\mathbf{x}-\mathbf{z}\|^2}{2\sigma^2}}$$

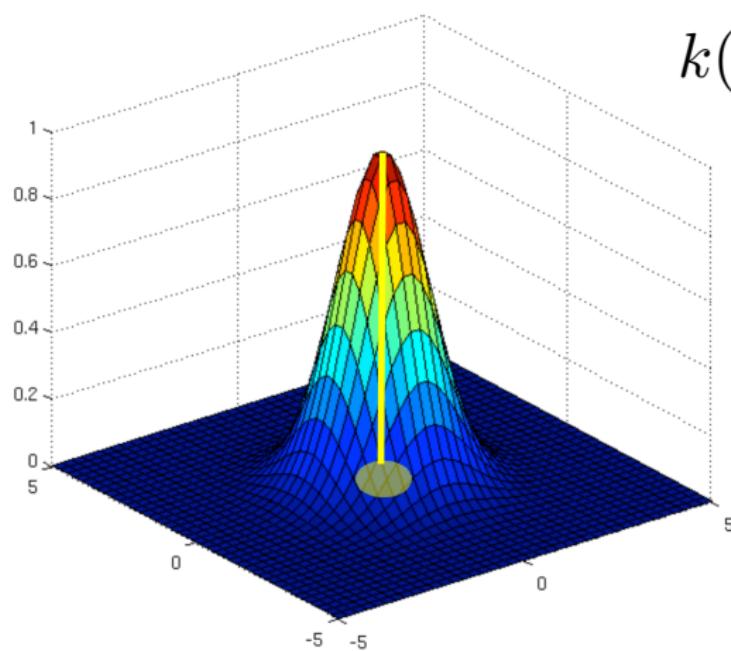


- ▶ The RBF is based on a **distance** from a central focus, \mathbf{z}

Image source: <https://www.cs.toronto.edu/~duvenaud/cookbook/>

Gaussian Radial Basis Function

$$k(\mathbf{x}, \mathbf{z}) = e^{-\frac{\|\mathbf{x}-\mathbf{z}\|^2}{2\sigma^2}}$$



$$\|\mathbf{x} - \mathbf{z}\| = 0$$

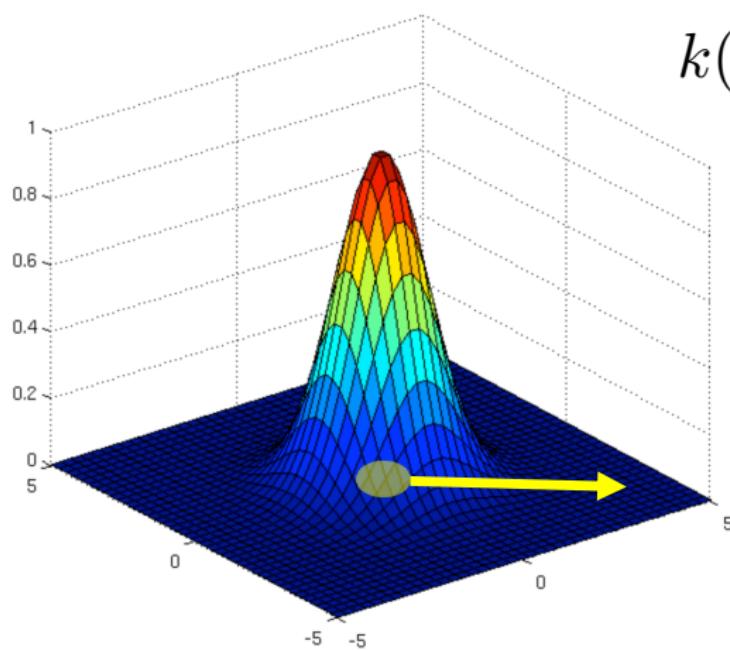
$$k(\mathbf{x}, \mathbf{z}) = e^0 = 1$$

- ▶ The value of the function is **highest** at point \mathbf{z} itself

Image source: <https://www.cs.toronto.edu/~duvenaud/cookbook/>

Gaussian Radial Basis Function

$$k(\mathbf{x}, \mathbf{z}) = e^{-\frac{\|\mathbf{x}-\mathbf{z}\|^2}{2\sigma^2}}$$



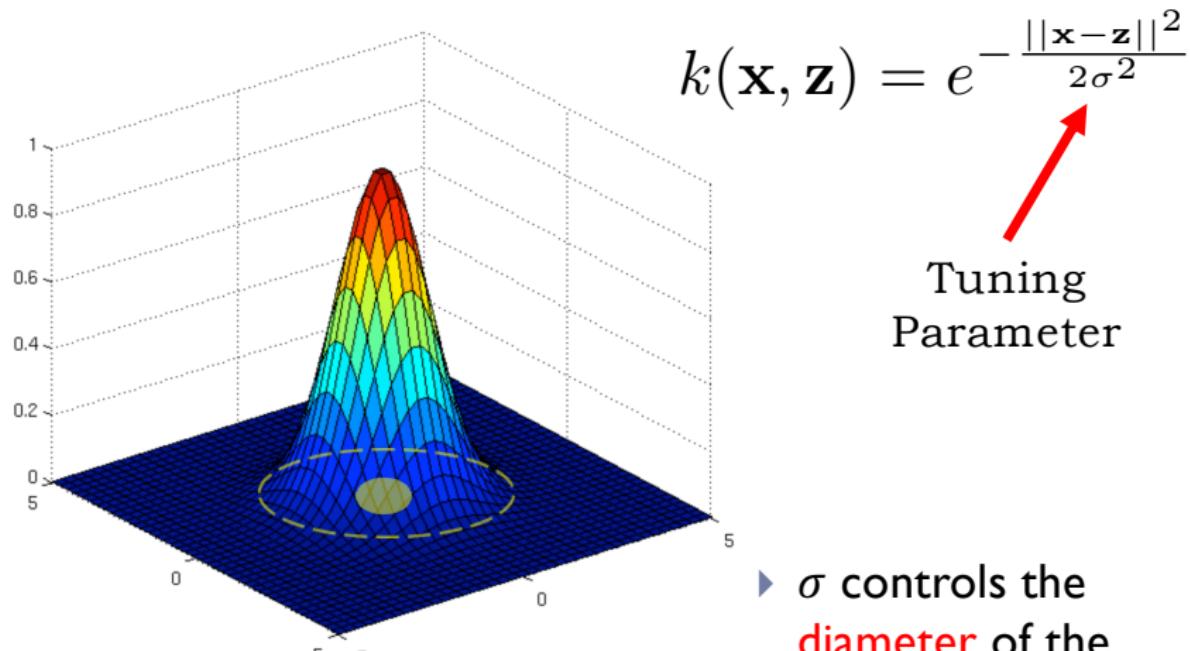
$$\|\mathbf{x} - \mathbf{z}\| \rightarrow \infty$$

$$k(\mathbf{x}, \mathbf{z}) \rightarrow e^{-\infty} = 0$$

- ▶ The value drops to 0 as we get further from \mathbf{z}

Image source: <https://www.cs.toronto.edu/~duvenaud/cookbook/>

Gaussian Radial Basis Function

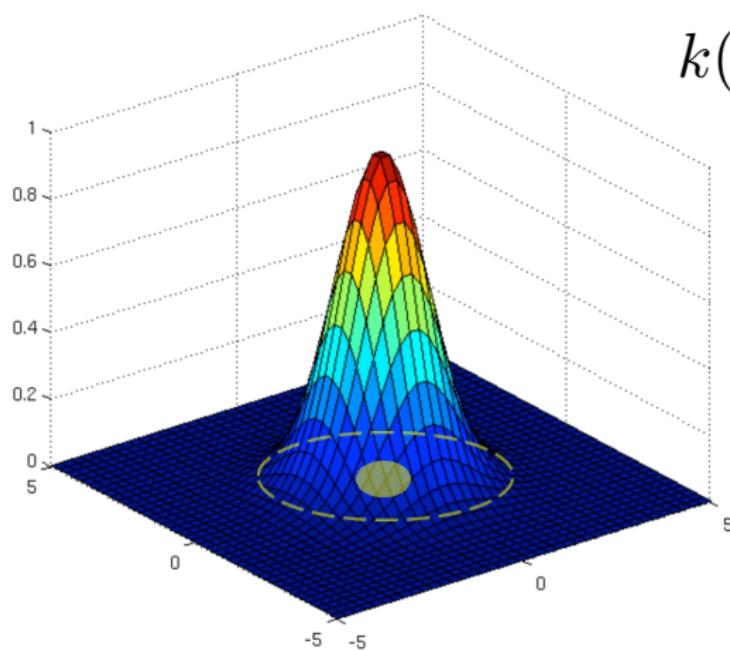


► σ controls the
diameter of the
non-zero area

Image source: <https://www.cs.toronto.edu/~duvenaud/cookbook/>

Gaussian Radial Basis Function

$$k(\mathbf{x}, \mathbf{z}) = e^{-\frac{\|\mathbf{x}-\mathbf{z}\|^2}{2\sigma^2}}$$



$$\sigma \rightarrow \infty$$

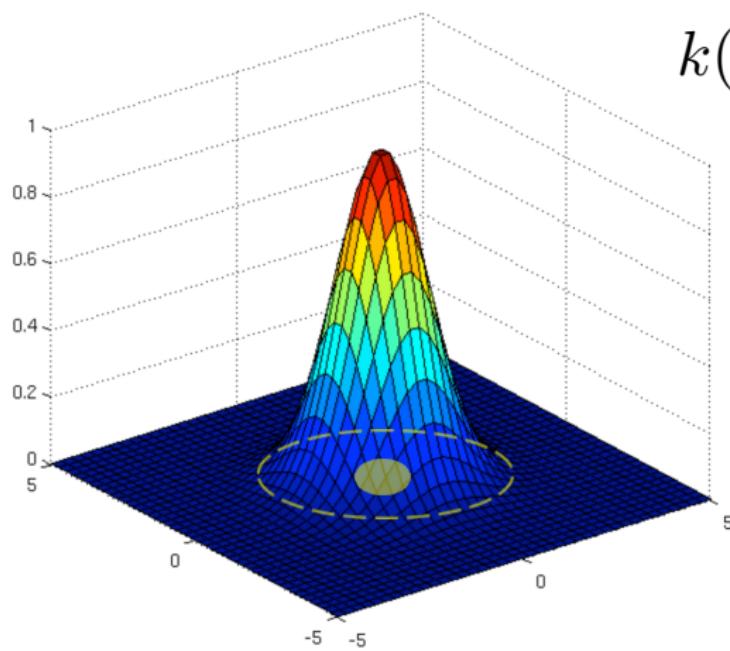
$$k(\mathbf{x}, \mathbf{z}) \rightarrow e^0 = 1$$

- ▶ If σ gets **larger**, the non-0 area will become **wider**

Image source: <https://www.cs.toronto.edu/~duvenaud/cookbook/>

Gaussian Radial Basis Function

$$k(\mathbf{x}, \mathbf{z}) = e^{-\frac{\|\mathbf{x}-\mathbf{z}\|^2}{2\sigma^2}}$$



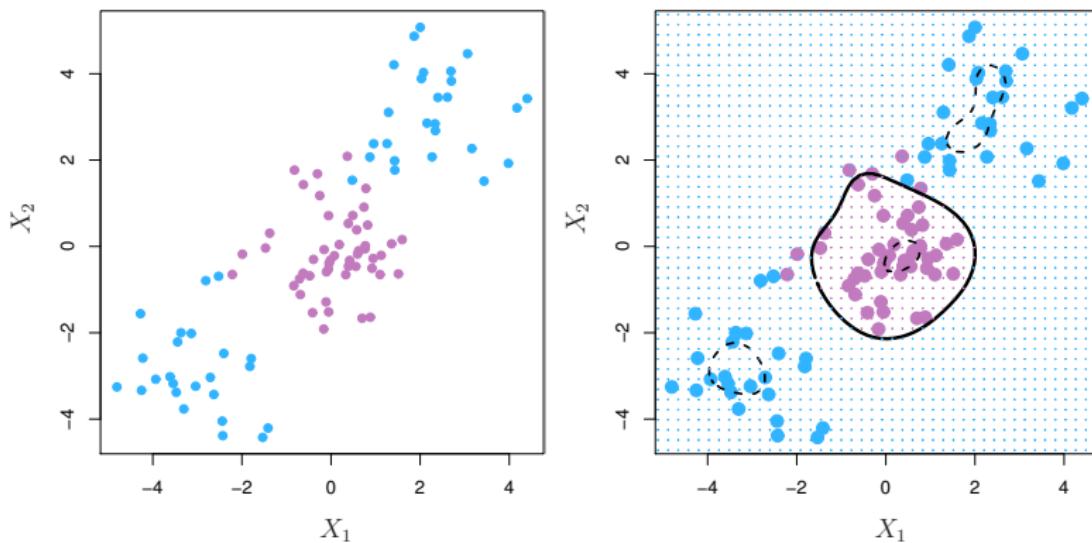
$$\sigma \rightarrow 0$$

$$k(\mathbf{x}, \mathbf{z}) \rightarrow e^{-\infty} = 0$$

- ▶ If σ gets **smaller**,
non-0 area will
become **narrower**

Image source: <https://www.cs.toronto.edu/~duvenaud/cookbook/>

SVM with Gaussian Kernel



Taken from "An Introduction to Statistical Learning, with applications in R" (Springer, 2013)
with permission from the authors: G. James, D. Witten, T. Hastie and R. Tibshirani.

CS 457/557: Machine Learning

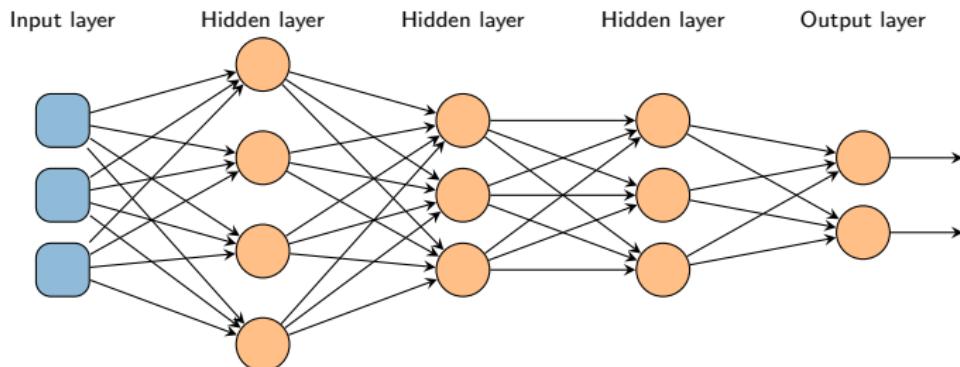
Lecture 07-1: Neural Networks

Lecture 07-1a: Neural Networks

Neural Learning Methods

Definition

A **neural network** is a computational model inspired by the structure of the human brain which consists of a network of simple information processing units called **neurons** (or **units**).

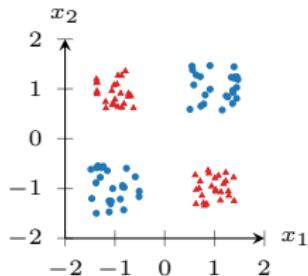


Can be used for classification and regression, plus other uses as well!

History of Neural Networks

A brief history:

- 1943: Early work by McCulloch and Pitts on modeling of neurons and network of connections that allow animals to learn
- 1957: Rosenblatt's perceptron algorithm was a single-layer neural network using the threshold activation function
- 1969: Minsky and Papert's book "Perceptrons" pointed out limitations (e.g., XOR) and led to the "AI Winter"



- Data is not linearly separable, so the perceptron algorithm cannot handle this!
- This problem can be solved using **multiple neurons**, but at the time it wasn't known how such networks could be trained

History of Neural Networks (continued)

1980s: The Connectionism Era: Connectionism is the idea that intelligent behavior can emerge from interactions between simple units.

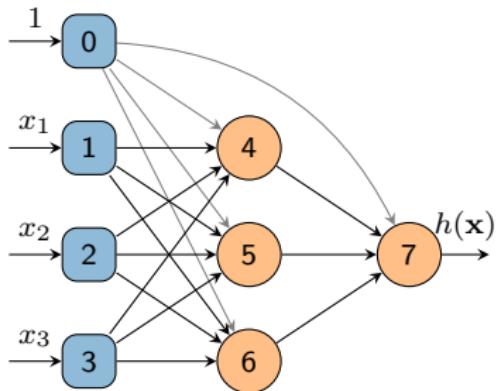
- 1986: The Parallel Distributed Processing (PDP) research group popularized the backpropagation algorithm for training a multi-layer neural network with logistic activation functions
- 1991: Vanishing gradient problem noted for deeper networks
- 1990s: Support vector machines gained in popularity due to ease of training compared to neural networks, leading to decreased interest in neural networks

2000s: Deep Learning Era

- Hinton et al. introduce greedy layer-wise pre-training process
- Use of rectified linear activation function (ReLU)
- Virtuous cycle: Better algorithms, faster hardware, bigger data

Lecture 07-1b: Anatomy of a Neuron

Network Topology and Notation



A **network** $\mathcal{N} = (V, A)$ consists of:

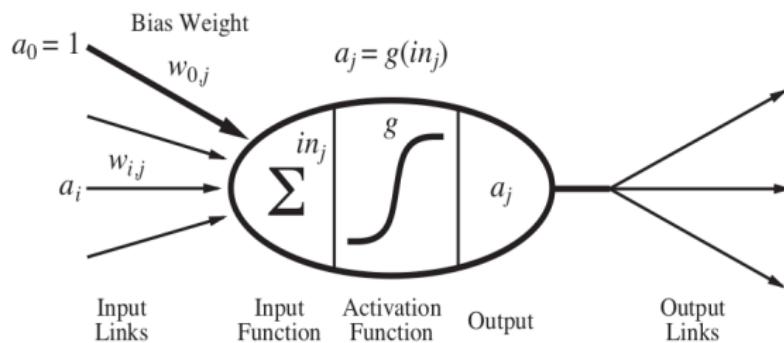
- a set of **vertices (nodes)** V , and
- a set of **directed edges (arcs)**
 $A \subseteq V \times V$

Neurons are labeled from 0 to n .

(Sometimes neurons are also indexed by layer, typically using a superscript, but we'll avoid this here to keep notation simple.)

- **Input neurons** are in **blue**, **processing neurons** are in **yellow**.
- The number of input neurons matches the dimensionality of a feature vector \mathbf{x} , plus the dummy neuron 0 (e.g., here $\mathbf{x} \in \mathbb{R}^3$)
- Neuron j produces an output a_j
- Input neuron j receives input x_j and transmits this as its output a_j (input neuron 0 receives and transmits $x_0 = 1$)
- Processing neuron j receives input from its incoming edges and produces some function of this input as its output a_j , which it transmits on its outgoing edges
- An edge (i, j) indicates a connection from neuron i to neuron j
- Input neuron 0 connects to all processing neurons (usually these edges are omitted)

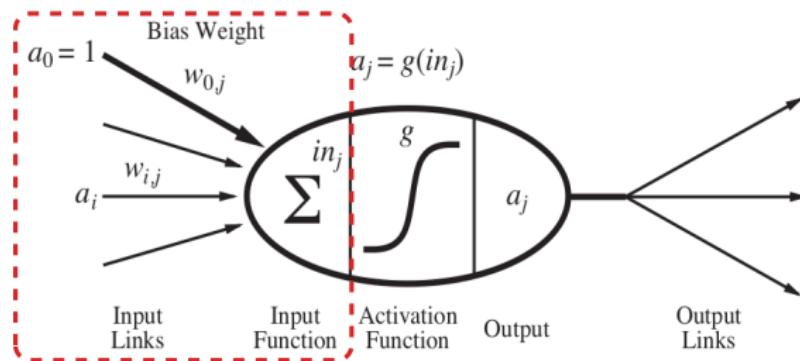
The Basic Neuron Model



For a processing neuron j :

- Input comes in from other neurons i with $(i, j) \in A$
- Some internal processing is done using the inputs
- Output transmitted to other neurons or used as network output

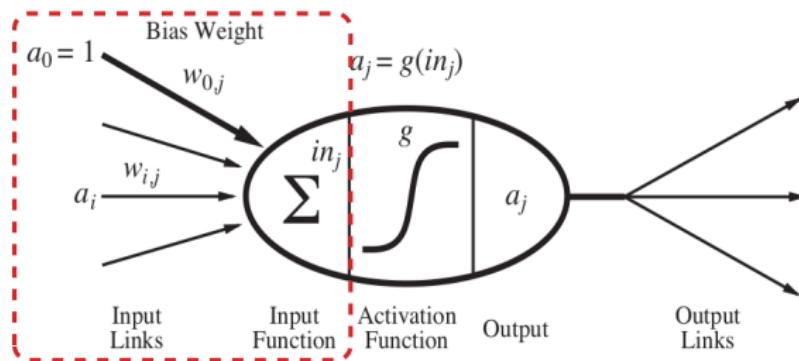
Analyzing A Neuron's Inputs



- Each input a_i from neuron i to neuron j weighted by w_{ij}
- Input $a_0 = 1$ from dummy neuron 0 is weighted by w_{0j}
- The input function is then a weighted linear sum:

$$in_j = \sum_{i:(i,j) \in A} w_{ij} a_i$$

Linear Transformations of Inputs

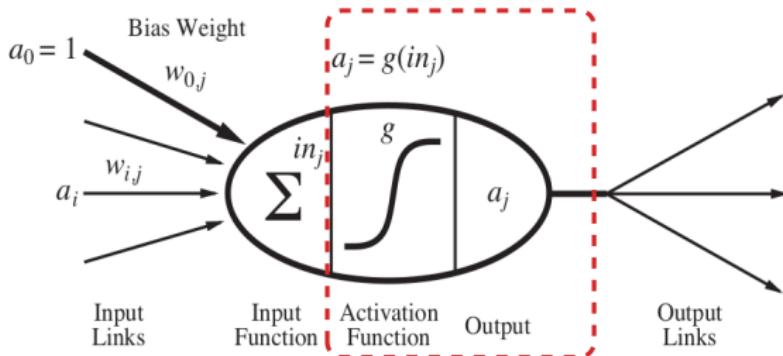


The weighted linear sum of inputs combined with $a_0 = 1$ is just like what gets done in other classifiers such as the perceptron!

$$in_j = \sum_{i:(i,j) \in A} w_{ij} a_i = \sum_{i=0}^n w_{ij} a_i = \mathbf{w}_j \cdot \mathbf{a}$$

where this last part abuses notation slightly using $w_{ij} = 0$ for $(i, j) \notin A$.

Neuron Output Functions



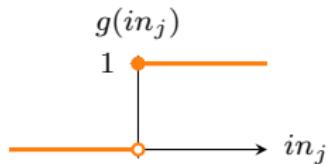
- After computing in_j , neuron j computes its output as $a_j = g(in_j)$, where g is an **activation function** that is generally **nonlinear**.
- The introduction of nonlinearity through g is crucial to allowing neural networks to **learn nonlinear functions!**

Different Activation Functions

Over the years, many different activation functions have been used:

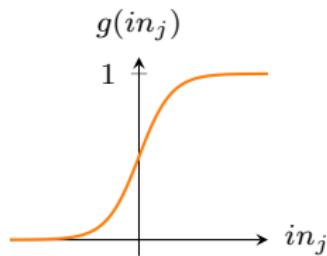
The **binary threshold** function:

$$g(in_j) = \begin{cases} 1 & \text{if } in_j \geq 0 \\ 0 & \text{otherwise} \end{cases}$$



The **logistic function**:

$$g(in_j) = \sigma(in_j) = \frac{1}{1 + e^{-in_j}}$$



- Provides a hard activation threshold
- Used by perceptron algorithm
- Not differentiable at 0; derivatives elsewhere are 0

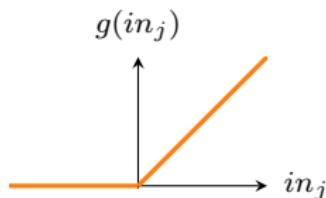
- Provides a soft activation threshold
- Differentiable everywhere, with $g'(in_j) = g(in_j)(1 - g(in_j))$
- Limits output to $[0, 1]$

Different Activation Functions (continued)

Over the years, many different activation functions have been used:

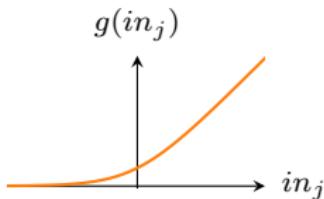
The **ReLU** function:

$$\begin{aligned}g(in_j) &= \text{ReLU}(in_j) \\&= \max \{0, in_j\}\end{aligned}$$



The **softplus** function:

$$\begin{aligned}g(in_j) &= \text{softplus}(in_j) \\&= \log(1 + e^{in_j})\end{aligned}$$



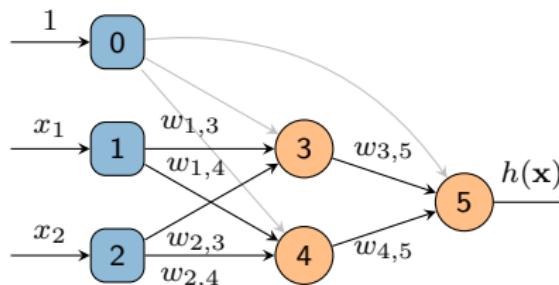
- ReLU is **rectified linear unit**
- Also known as a “ramp” function
- Popular in many modern applications
- Not differentiable at 0; derivative of 1 for positives 0 for negatives

- Smoothed version of ReLU
- Differentiable everywhere, with $g'(in_j) = \sigma(in_j)$

Lecture 07-1c: Forward Propagation

Example: Making Predictions Using Forward Propagation

Let's make a prediction for the network below using input $\mathbf{x} = (3, 4)$, the weights given at the right, and the logistic activation function.



Hidden layer:

$$\begin{aligned}in_3 &= w_{0,3}a_0 + w_{1,3}a_1 + w_{2,3}a_2 \\&= 5 \cdot 1 + 1 \cdot 3 - 2 \cdot 4 = 0\end{aligned}$$

$$a_3 = g(in_3) = 0.5$$

$$\begin{aligned}in_4 &= w_{0,4}a_0 + w_{1,4}a_1 + w_{2,4}a_2 \\&= 3 \cdot 1 - 4 \cdot 3 + 2 \cdot 4 = -1\end{aligned}$$

$$a_4 = g(in_4) \approx 0.27$$

Parameter values:

$$\begin{array}{lll}w_{0,3} = 5 & w_{0,4} = 3 & w_{0,5} = -1 \\w_{1,3} = 1 & w_{1,4} = -4 & w_{3,5} = 4 \\w_{2,3} = -2 & w_{2,4} = 2 & w_{4,5} = 2\end{array}$$

Input layer:

$$\begin{array}{ll}a_0 = 1 & a_1 = x_1 = 3 \\a_2 = x_2 = 4 &\end{array}$$

Output layer:

$$\begin{aligned}in_5 &= w_{0,5}a_0 + w_{3,5}a_3 + w_{4,5}a_4 \\&= -1 \cdot 1 + 4 \cdot 0.5 + 2 \cdot 0.27 = 1.54 \\a_5 &= g(in_5) \approx 0.82\end{aligned}$$

$$h(\mathbf{x}) = a_5 \approx 0.82$$

Compute node outputs in order, one at a time!

CS 457/557: Machine Learning

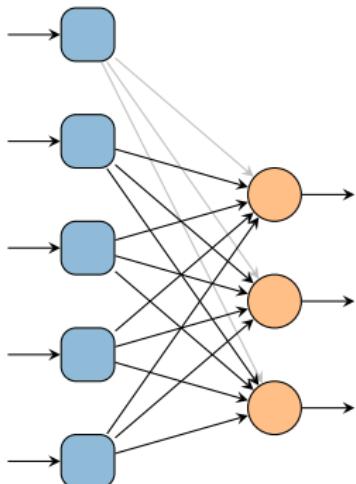
Lecture 07-2: Neural Networks

Lecture 07-2a: Single-Layer Perceptron Networks

Single-Layer Perceptron Networks

Definition

A **single-layer perceptron network** consists of an input layer and an output layer, with no hidden layers.

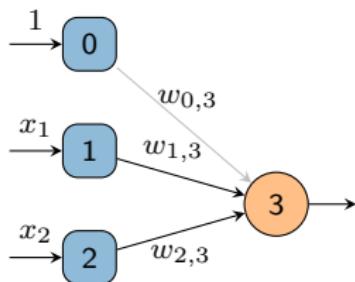


- Input neurons are connected directly to output neurons
- Each output neuron provides the probability of belonging to a particular class (e.g., for **multi-class classification**)
- Rosenblatt's perceptron algorithm is a special case with one output node using the binary threshold activation function

A Simple Example

Consider a network with a single processing neuron in the output layer.

With a logistic activation function, we have:



$$\begin{aligned}in_3 &= w_{0,3}a_0 + w_{1,3}a_1 + w_{2,3}a_2 \\&= w_{0,3} + w_{1,3}x_1 + w_{2,3}x_2 \\a_3 &= g(in_3) = \sigma(in_3) \\&= \frac{1}{1 + e^{-(w_{0,3} + w_{1,3}x_1 + w_{2,3}x_2)}}\end{aligned}$$

This is **logistic regression!**

Training is the process of finding optimal weights \mathbf{w}^* . This requires:

- Loss function
- Cost function
- Fitting method (optimization algorithm)

Some Loss Functions

Some loss functions that we have seen so far (treating $y_i \in \{0, 1\}$ for classification):

$$L_2(y, h_{\mathbf{w}}(\mathbf{x})) = (y - h_{\mathbf{w}}(\mathbf{x}))^2$$

$$L_{0/1}(y, h_{\mathbf{w}}(\mathbf{x})) = \begin{cases} 0 & \text{if } y = h(\mathbf{x}) \\ 1 & \text{otherwise} \end{cases}$$

$$L_{\pi}(y, h_{\mathbf{w}}(\mathbf{x})) = \max\{0, (1 - 2y)\mathbf{w} \cdot \mathbf{x}\}$$

$$L_{\log}(y, h_{\mathbf{w}}(\mathbf{x})) = -1 [y \log(h_{\mathbf{w}}(\mathbf{x})) + (1 - y) \log(1 - h_{\mathbf{w}}(\mathbf{x}))]$$

To keep things simple, we'll use L_2 loss for now for both regression and classification.

- Recall: L_2 loss leads to a **non-convex** cost function for logistic regression
- However, for networks with more than one layer, most cost functions are non-convex regardless of the loss function that is used

The Cost Function and Fitting Process

We'll use the cost function $J(\mathbf{w})$ as the average loss over the training set:

$$J(\mathbf{w}) = \frac{1}{N} \sum_{i=1}^N L_2(y_i, h_{\mathbf{w}}(\mathbf{x}_i)).$$

For fitting, we'll use gradient descent, specifically **stochastic gradient descent** (SGD), which has appealing properties for neural network training. Recall:

- Gradient descent converges to a local minimum
- If cost function is **convex**, all local minima are global minima
- If cost function is **non-convex**, gradient descent may not find the global optimum

SGD can avoid getting stuck in “shallow” local minima during search.

SGD Weight Updates

With stochastic gradient descent, each weight update examines a **single example** in the training set. For convenience, we'll denote this training example as (\mathbf{x}, y) (no subscripts), and we'll treat the cost function $J(\mathbf{w})$ as only using this training example, i.e.:

$$J(\mathbf{w}) = L_2(y, h_{\mathbf{w}}(\mathbf{x})).$$

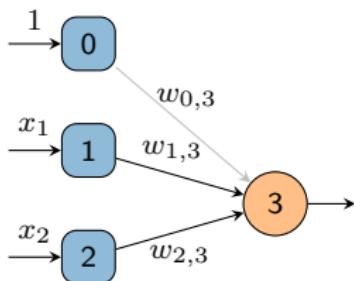
In SGD, the weight updates require the partial derivatives of $J(\mathbf{w})$ with respect to each weight w_{ij} :

$$w_{ij}^{(t+1)} \leftarrow w_{ij}^{(t)} - \alpha \frac{\partial}{\partial w_{ij}} J(\mathbf{w}).$$

To compute $\frac{\partial}{\partial w_{ij}} J(\mathbf{w})$, we'll make use of the **chain rule** from calculus:

$$\frac{\partial}{\partial x} g(f(x)) = g'(f(x)) \frac{\partial}{\partial x} f(x) = g'(f(x))f'(x).$$

Computing the Partial Derivatives



For the network on the left, we have:

$$\begin{aligned} J(\mathbf{w}) &= L_2(y, h_{\mathbf{w}}(\mathbf{x})) \\ &= (y - h_{\mathbf{w}}(\mathbf{x}))^2 \\ &= (y - a_3)^2 \\ &= (y - g(in_3))^2 \\ in_3 &= w_{0,3}a_0 + w_{1,3}a_1 + w_{2,3}a_2 \\ &= w_{0,3} + w_{1,3}x_1 + w_{2,3}x_2 \end{aligned}$$

So:

$$\begin{aligned} \frac{\partial}{\partial w_{ij}} J(\mathbf{w}) &= \frac{\partial}{\partial w_{ij}} (y - g(in_3))^2 && [\text{By above derivation}] \\ &= 2(y - g(in_3)) \frac{\partial}{\partial w_{ij}} (y - g(in_3)) && [\text{By chain rule}] \\ &= -2(y - g(in_3)) \frac{\partial}{\partial w_{ij}} g(in_3) \\ &= -2(y - g(in_3)) g'(in_3) \frac{\partial}{\partial w_{ij}} in_3 && [\text{By chain rule}] \\ &= -2(y - g(in_3)) g'(in_3) \frac{\partial}{\partial w_{ij}} (w_{0,3} + w_{1,3}x_1 + w_{2,3}x_2) && [\text{Defn. of } in_3] \end{aligned}$$

Computing the Partial Derivatives

From the previous slide, we had

$$\frac{\partial}{\partial w_{ij}} J(\mathbf{w}) = -2(y - g(in_3))g'(in_3) \frac{\partial}{\partial w_{ij}} (w_{0,3} + w_{1,3}x_1 + w_{2,3}x_2),$$

so

$$\frac{\partial}{\partial w_{0,3}} J(\mathbf{w}) = -2(y - g(in_3))g'(in_3)$$

$$\frac{\partial}{\partial w_{1,3}} J(\mathbf{w}) = -2(y - g(in_3))g'(in_3)x_1$$

$$\frac{\partial}{\partial w_{2,3}} J(\mathbf{w}) = -2(y - g(in_3))g'(in_3)x_2.$$

- $y - g(in_3)$ is the **error** in prediction given example (\mathbf{x}, y)
- $g'(in_3)$ is the derivative of the activation function evaluated at in_3 ; if g is the logistic function σ , then $g'(t) = g(t)(1 - g(t))$ for all $t \in \mathbb{R}$.

Putting It All Together

Using the logistic activation function, the gradient descent updates are:

$$\begin{aligned}w_{0,3}^{(t+1)} &\leftarrow w_{0,3}^{(t)} + 2\alpha(y - g(in_3))g(in_3)(1 - g(in_3)) \\w_{1,3}^{(t+1)} &\leftarrow w_{1,3}^{(t)} + 2\alpha(y - g(in_3))g(in_3)(1 - g(in_3))x_1 \\w_{2,3}^{(t+1)} &\leftarrow w_{2,3}^{(t)} + 2\alpha(y - g(in_3))g(in_3)(1 - g(in_3))x_2.\end{aligned}$$

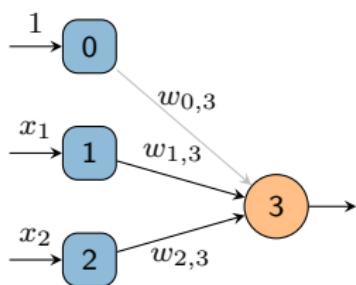
where $in_3 = w_{0,3} + w_{1,3}x_1 + w_{2,3}x_2$.

This allows us to train a neural network with no hidden layers using SGD:

- ① Select an example (\mathbf{x}, y) from the training set
- ② Compute in_3 and $g(in_3)$ using \mathbf{x}
- ③ Update weights as above
- ④ Repeat until convergence

Lecture 07-2b: Example SGD Iteration

Example Iteration



Suppose at iteration t we have:

$$w_{0,3}^{(t)} = 0.1$$

$$w_{1,3}^{(t)} = 0.1$$

$$w_{2,3}^{(t)} = 0.1$$

- ① We select example (\mathbf{x}, y) for training with $\mathbf{x} = (0.4, 0.5)$ and $y = 1$.
- ② We compute:

$$\begin{aligned}in_3 &= w_{0,3} + w_{1,3}x_1 + w_{2,3}x_2 & g(in_3) &= \frac{1}{1 + e^{-0.19}} \\&= 0.1 + (0.1)(0.4) + (0.1)(0.5) & &\approx 0.5474 \\&= 0.19\end{aligned}$$

Updating the Weights

③ Update the weights:

$$(y - g(in_3)) = (1 - 0.5474) = 0.4526$$

$$g(in_3)(1 - g(in_3)) = 0.5474(1 - 0.5474) \approx 0.2478$$

Using $\alpha = 0.5$ for simplicity:

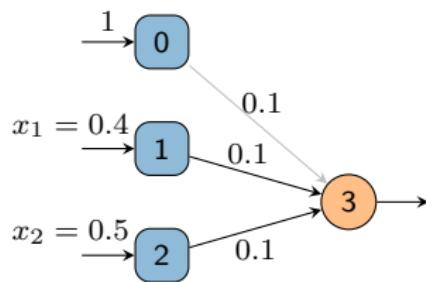
$$\begin{aligned} w_{0,3}^{(t+1)} &\leftarrow w_{0,3}^{(t)} + 2\alpha(y - g(in_3))g(in_3)(1 - g(in_3)) \\ &= 0.1 + (0.4526)(0.2478) \\ &\approx 0.2122 \end{aligned}$$

$$\begin{aligned} w_{1,3}^{(t+1)} &\leftarrow w_{1,3}^{(t)} + 2\alpha(y - g(in_3))g(in_3)(1 - g(in_3))x_1 \\ &= 0.1 + (0.4526)(0.2478)(0.4) \\ &\approx 0.1449 \end{aligned}$$

$$\begin{aligned} w_{2,3}^{(t+1)} &\leftarrow w_{2,3}^{(t)} + 2\alpha(y - g(in_3))g(in_3)(1 - g(in_3))x_2 \\ &= 0.1 + (0.4526)(0.2478)(0.5) \\ &\approx 0.1561 \end{aligned}$$

Evaluating the Change in Weights

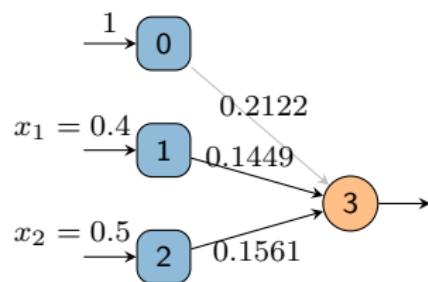
At iteration t :



$$\begin{aligned}in_3 &= 0.1 + (0.1)(0.4) + (0.1)(0.5) \\&= 0.19\end{aligned}$$

$$g(in_3) \approx 0.5474$$

At iteration $t + 1$:



$$\begin{aligned}in_3 &= 0.2122 + (0.1449)(0.4) + (0.1561)(0.5) \\&= 0.34821\end{aligned}$$

$$g(in_3) \approx 0.5862$$

After the weight update, the neuron connections have been **strengthened**:

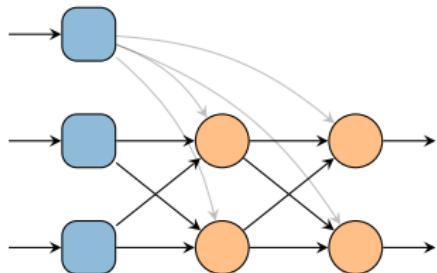
- Each weight increased in proportion to the error made as well as the input to that connection
- This causes the value computed by the output neuron to increase as well, reducing (but not eliminating) the error on this training example

Lecture 07-2c: Feedforward Neural Networks

Feedforward Neural Networks

Definition

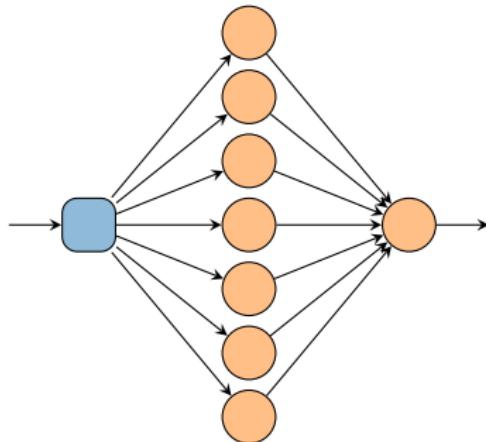
A **feedforward neural network (FFNN, multilayer perceptron)** consists of an input layer, an output layer, and one or more hidden layers. All node connections go from a layer to the subsequent layer.



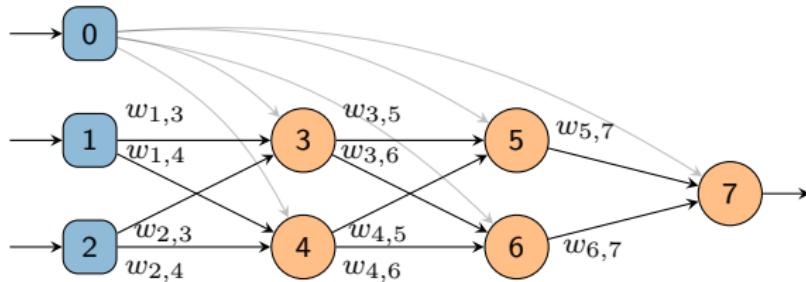
- Using hidden layers overcomes XOR problem of single-layer perceptrons
- The layout of the network is called the **architecture**; different architectures are suitable for different types of learning; fully-connected FFNN is a reasonable default
- Multilayer networks do feature engineering **automatically!**

Usefulness of Hidden Layers

The *Universal Approximation Theorem* states that under certain mild conditions, **any** continuous function can be approximated by a neural network with a **single hidden layer** and a large enough number of neurons.



The Effects of Hidden Layers



Output neurons process inputs **indirectly** via values that come from the hidden layers:

$$\begin{aligned}a_7 &= g(w_{0,7} + w_{5,7}a_5 + w_{6,7}a_6) \\&= g(w_{0,7} + w_{5,7}g(w_{0,5} + w_{3,5}a_3 + w_{4,5}a_4) + w_{6,7}g(w_{0,6} + w_{3,6}a_3 + w_{4,6}a_4)) \\&= g(w_{0,7} + w_{5,7}g(w_{0,5} + w_{3,5}g(w_{0,3} + w_{1,3}a_1 + w_{2,3}a_2) \\&\quad + w_{4,5}g(w_{0,4} + w_{1,4}a_1 + w_{2,4}a_2))) \\&\quad + w_{6,7}g(w_{0,6} + w_{3,6}g(w_{0,3} + w_{1,3}a_1 + w_{2,3}a_2) \\&\quad + w_{4,6}g(w_{0,4} + w_{1,4}a_1 + w_{2,4}a_2)))\end{aligned}$$

Computing Partial Derivatives in a Multi-Layer Network

For the network on the previous slide, let's try to compute some partial derivatives of $J(\mathbf{w})$ given training example (\mathbf{x}, y) .

$$\begin{aligned}\frac{\partial}{\partial w_{ij}} J(\mathbf{w}) &= \frac{\partial}{\partial w_{ij}} (y - a_7)^2 \\&= \frac{\partial}{\partial w_{ij}} (y - g(in_7))^2 \\&= 2(y - g(in_7)) \frac{\partial}{\partial w_{ij}} (y - g(in_7)) \\&= -2(y - g(in_7)) \frac{\partial}{\partial w_{ij}} g(in_7) \\&= -2(y - g(in_7)) g'(in_7) \frac{\partial}{\partial w_{ij}} in_7 \\&= \textcolor{green}{-2(y - g(in_7)) g'(in_7)} \frac{\partial}{\partial w_{ij}} (w_{0,7} + w_{5,7}a_5 + w_{6,7}a_6)\end{aligned}$$

Computing Partial Derivatives: Weights on Edges to Output Layer

On the previous slide, we saw that:

$$\frac{\partial}{\partial w_{ij}} J(\mathbf{w}) = -2(y - g(in_7)) g'(in_7) \frac{\partial}{\partial w_{ij}} (w_{0,7} + w_{5,7}a_5 + w_{6,7}a_6)$$

So for $w_{5,7}$, we have:

$$\begin{aligned}\frac{\partial}{\partial w_{5,7}} J(\mathbf{w}) &= -2(y - g(in_7)) g'(in_7) \frac{\partial}{\partial w_{5,7}} (w_{0,7} + w_{5,7}a_5 + w_{6,7}a_6) \\ &= -2(y - g(in_7)) g'(in_7)a_5.\end{aligned}$$

This looks **similar** to what we had earlier with no hidden layers!

Computing Partial Derivatives: Weights on Edges to Hidden Layer

For $w_{3,5}$, we have:

$$\begin{aligned}\frac{\partial}{\partial w_{3,5}} J(\mathbf{w}) &= -2(y - g(in_7)) g'(in_7) \frac{\partial}{\partial w_{3,5}} (w_{0,7} + w_{5,7}a_5 + w_{6,7}a_6) \quad [\text{Where is } w_{3,5}?] \\ &= -2(y - g(in_7)) g'(in_7) \frac{\partial}{\partial w_{3,5}} w_{5,7}a_5 \quad [\text{Buried in } a_5!] \\ &= -2(y - g(in_7)) g'(in_7) w_{5,7} \frac{\partial}{\partial w_{3,5}} g(in_5) \\ &= -2(y - g(in_7)) g'(in_7) \cancel{w_{5,7}g'(in_5)} \frac{\partial}{\partial w_{3,5}} in_5 \\ &= -2(y - g(in_7)) g'(in_7) \cancel{w_{5,7}g'(in_5)} \frac{\partial}{\partial w_{3,5}} (w_{0,5} + w_{3,5}a_3 + w_{4,5}a_4) \\ &= -2(y - g(in_7)) g'(in_7) \cancel{w_{5,7}g'(in_5)} a_3\end{aligned}$$

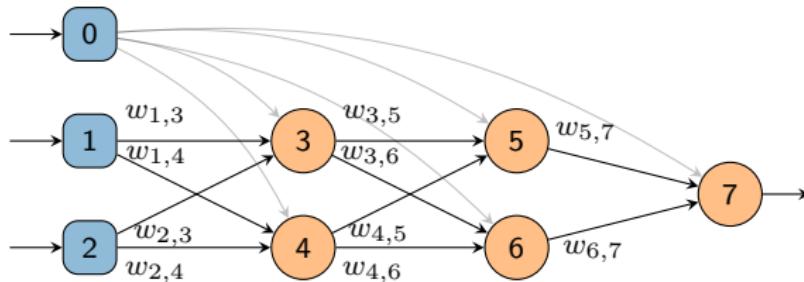
This takes more work, but in principle we can do this for any weight w_{ij} .

CS 457/557: Machine Learning

Lecture 07-3: The Backpropagation Algorithm

Lecture 07-3a: Learning in Multi-Layer Neural Networks

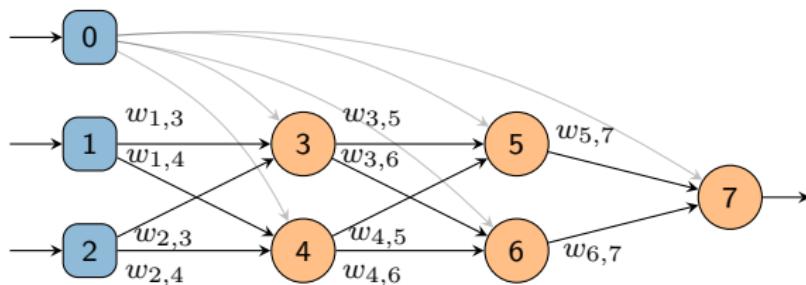
Learning in Multi-Layer Neural Networks



For a neural network $\mathcal{N} = (V, A)$, we can use **stochastic gradient descent** to search for appropriate weights w_{ij} for all $(i, j) \in A$, **provided** that we know the partial derivatives of $J(\mathbf{w})$!

However, computing partial derivatives $\frac{\partial}{\partial w_{ij}} J(\mathbf{w})$ in a multi-layer network requires a bit more work...

Partial Derivatives in a Multi-Layer Network



For the above network, we have:

$$\frac{\partial}{\partial w_{5,7}} J(\mathbf{w}) = -2(y - g(in_7)) g'(in_7) a_5$$

$$\frac{\partial}{\partial w_{6,7}} J(\mathbf{w}) = -2(y - g(in_7)) g'(in_7) a_6$$

$$\frac{\partial}{\partial w_{3,5}} J(\mathbf{w}) = -2(y - g(in_7)) g'(in_7) w_{5,7} g'(in_5) a_3$$

$$\frac{\partial}{\partial w_{3,6}} J(\mathbf{w}) = -2(y - g(in_7)) g'(in_7) w_{6,7} g'(in_6) a_3$$

Computing Partial Derivatives: Weights on Edges to Hidden Layer

For $w_{1,3}$, we have:

$$\begin{aligned}\frac{\partial}{\partial w_{1,3}} J(\mathbf{w}) &= -2(y - g(in_7)) g'(in_7) \frac{\partial}{\partial w_{1,3}} (w_{0,7} + w_{5,7}a_5 + w_{6,7}a_6) \quad [\text{Where is } w_{1,3}?] \\ &= -2(y - g(in_7)) g'(in_7) \frac{\partial}{\partial w_{1,3}} (w_{5,7}a_5 + w_{6,7}a_6) \quad [\text{Buried in both } a_5 \text{ and } a_6!] \\ &= -2(y - g(in_7)) g'(in_7) \left(w_{5,7} \frac{\partial}{\partial w_{1,3}} g(in_5) + w_{6,7} \frac{\partial}{\partial w_{1,3}} g(in_6) \right) \\ &= -2(y - g(in_7)) g'(in_7) \left(w_{5,7}g'(in_5) \frac{\partial}{\partial w_{1,3}} in_5 + w_{6,7}g'(in_6) \frac{\partial}{\partial w_{1,3}} in_6 \right) \\ &= \vdots\end{aligned}$$

This takes more work, but in principle we can do this for any weight w_{ij} .

- We see some repeated terms showing up though – we can exploit this to make the partial derivative calculations easier!

Lecture 07-3b: Computing Arbitrary Partial Derivatives

The Chain Rule Revisited

Chain Rule

Suppose $z = g(f(x))$ for some functions f and g , and let $y = f(x)$. Then

$$\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dz}.$$

Intuitively:

$$\begin{aligned}\text{change in } z \text{ per change in } x &= \text{change in } z \text{ per change in } y \\ &\quad \times \text{change in } y \text{ per change in } x.\end{aligned}$$

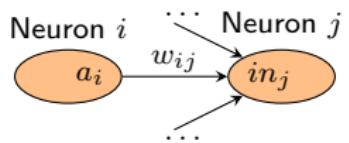
We can use this to **reformulate** our partial derivative calculations!

- We know that w_{ij} is a variable which influences $J(\mathbf{w})$.
- However, in_j is **another variable** that also influences $J(\mathbf{w})$, and in_j **depends on** w_{ij} because w_{ij} is used to compute in_j !

Reframing Partial Derivatives

For any $(i, j) \in A$, we have:

$$\begin{aligned}\frac{\partial}{\partial w_{ij}} J(\mathbf{w}) &= \left[\frac{\partial}{\partial in_j} J(\mathbf{w}) \right] \left[\frac{\partial}{\partial w_{ij}} in_j \right] \\ &= \left[\frac{\partial}{\partial in_j} J(\mathbf{w}) \right] \left[\frac{\partial}{\partial w_{ij}} \left(\sum_{i':(i',j) \in A} w_{i'j} a_{i'} \right) \right] \\ &= \left[\frac{\partial}{\partial in_j} J(\mathbf{w}) \right] a_i\end{aligned}$$



- a_i : Output of neuron i
- in_j : Input to neuron j

Intuitively,

$$\begin{aligned}\text{change in } J(\mathbf{w}) \text{ per change in } w_{ij} &= \text{change in } J(\mathbf{w}) \text{ per change in } in_j \\ &\quad \times \text{change in } in_j \text{ per change in } w_{ij}.\end{aligned}$$

Now we just need to calculate $\frac{\partial}{\partial in_j} J(\mathbf{w})$!

Computing Partial Derivatives With Respect To in_j : Base Case

To make notation easier, for any neuron $j \in V$, let

$$\Delta_j = \frac{\partial}{\partial in_j} J(\mathbf{w}).$$

If neuron j is in the **output layer** (the **base case**), then with squared loss we have:

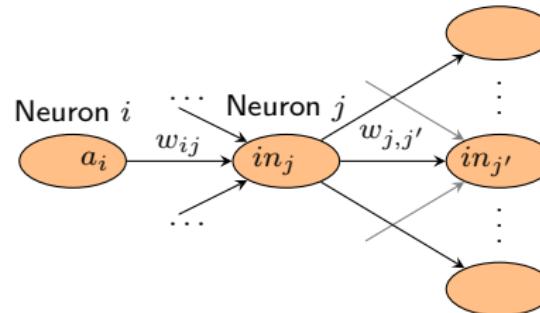
$$\begin{aligned}\Delta_j &= \frac{\partial}{\partial in_j} J(\mathbf{w}) = \frac{\partial}{\partial in_j} (y - a_j)^2 \\&= 2(y - a_j) \frac{\partial}{\partial in_j} (y - g(in_j)) \\&= -2(y - a_j) g'(in_j) \frac{\partial}{\partial in_j} in_j \\&= -2(y - a_j) g'(in_j).\end{aligned}$$

(This matches what we had seen earlier for the multi-layer network examples.)

Computing Partial Derivatives With Respect To in_j : Recursive Case

If neuron j is in a **hidden layer**, then we can use the chain rule to reformulate our computation for Δ_j as:

$$\begin{aligned}\Delta_j &= \frac{\partial}{\partial in_j} J(\mathbf{w}) = \sum_{j':(j,j') \in A} \left[\frac{\partial}{\partial in_{j'}} J(\mathbf{w}) \right] \left[\frac{\partial}{\partial in_j} in_{j'} \right] \\ &= \sum_{j':(j,j') \in A} \Delta_{j'} \left[\frac{\partial}{\partial in_j} in_{j'} \right].\end{aligned}$$



More Work

Now we just need to compute $\frac{\partial}{\partial \text{in}_j} \text{in}_{j'}$:

$$\begin{aligned}\frac{\partial}{\partial \text{in}_j} \text{in}_{j'} &= \frac{\partial}{\partial \text{in}_j} \sum_{i':(i',j') \in A} w_{i'j'} a_{i'} && [\text{Defn. of } \text{in}_{j'}] \\ &= \frac{\partial}{\partial \text{in}_j} \sum_{i':(i',j') \in A} w_{i'j'} g(\text{in}_{i'}) && [\text{Defn. of } a_{i'}] \\ &= \frac{\partial}{\partial \text{in}_j} w_{j,j'} g(\text{in}_j) \\ &= w_{j,j'} g'(\text{in}_j) \frac{\partial}{\partial \text{in}_j} \text{in}_j \\ &= w_{j,j'} g'(\text{in}_j)\end{aligned}$$

Then

$$\Delta_j = \sum_{j':(j,j') \in A} \Delta_{j'} \left[\frac{\partial}{\partial \text{in}_j} \text{in}_{j'} \right] = \sum_{j':(j,j') \in A} \Delta_{j'} w_{j,j'} g'(\text{in}_j). = g'(\text{in}_j) \sum_{j':(j,j') \in A} w_{j,j'} \Delta_{j'}.$$

Putting It All Together

For any $(i, j) \in A$, we have:

$$\frac{\partial}{\partial w_{ij}} J(\mathbf{w}) = \left[\frac{\partial}{\partial \text{in}_j} J(\mathbf{w}) \right] a_i = \Delta_j a_i.$$

We also saw that

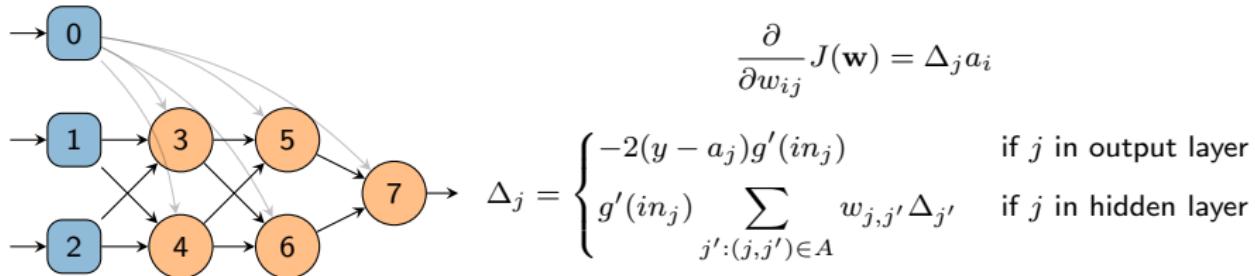
$$\Delta_j = \begin{cases} -2(y - a_j)g'(in_j) & \text{if } j \text{ belongs to output layer} \\ g'(in_j) \sum_{j':(j,j') \in A} w_{j,j'} \Delta_{j'} & \text{if } j \text{ belongs to hidden layer} \end{cases}$$

We can compute these Δ_j values in a “backwards” fashion, starting from the neurons in the output layer and moving towards the input layer.

- Using a different loss function will change the above expression for Δ_j for the output layer, but not for the hidden layers

Lecture 07-3c: Example Calculations

Some Calculations



Computing some partial derivatives:

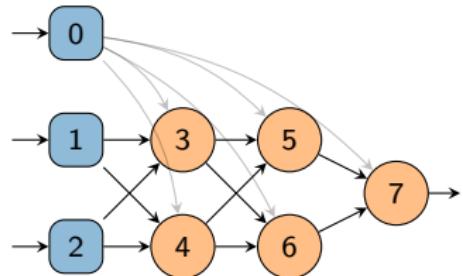
$$\frac{\partial}{\partial w_{5,7}} J(\mathbf{w}) = \Delta_7 a_5$$
$$= -2(y - a_7)g'(in_7) a_5$$

$$\frac{\partial}{\partial w_{3,5}} J(\mathbf{w}) = \Delta_5 a_3$$
$$= \Delta_7 w_{5,7}g'(in_5) a_3$$
$$= -2(y - a_7)g'(in_7) w_{5,7}g'(in_5) a_3$$

Computing some Δ_j 's:

$$\Delta_7 = -2(y - a_7)g'(in_7)$$
$$\Delta_5 = g'(in_5) \sum_{j':(5,j') \in A} \Delta_{j'} w_{5,j'}$$
$$= \Delta_7 w_{5,7}g'(in_5)$$
$$\Delta_6 = g'(in_6) \sum_{j':(6,j') \in A} \Delta_{j'} w_{6,j'}$$
$$= \Delta_7 w_{6,7}g'(in_6)$$

Some More Calculations



$$\frac{\partial}{\partial w_{ij}} J(\mathbf{w}) = \Delta_j a_i$$

$$\Delta_j = \begin{cases} -2(y - a_j)g'(in_j) & \text{if } j \text{ in output layer} \\ g'(in_j) \sum_{j':(j,j') \in A} w_{j,j'} \Delta_{j'} & \text{if } j \text{ in hidden layer} \end{cases}$$

Computing some partial derivatives:

$$\frac{\partial}{\partial w_{1,3}} J(\mathbf{w}) = \Delta_3 a_1$$

Computing some Δ_j 's:

$$\Delta_7 = -2(y - a_7)g'(in_7)$$

$$\Delta_5 = \Delta_7 w_{5,7}g'(in_5)$$

$$\Delta_6 = \Delta_7 w_{6,7}g'(in_6)$$

$$\Delta_3 = g'(in_3) \sum_{j':(3,j') \in A} w_{3,j'} \Delta_{j'}$$

$$= g'(in_3) (w_{3,5}\Delta_5 + w_{3,6}\Delta_6)$$

$$= g'(in_3) (w_{3,5}\Delta_7 w_{5,7}g'(in_5) + w_{3,6}\Delta_7 w_{6,7}g'(in_6))$$

Lecture 07-3d: The Backpropagation Algorithm

Backpropagation Algorithm

The Backpropagation Algorithm

```
procedure BACKPROPUPDATE((x, y); N = (V, A); w(t))
    /* Forward Propagation */
    for each neuron j in input layer do
        aj ← xj
    for each layer ℓ from 2 to L do
        for each neuron j in layer ℓ do
            inj ← ∑i:(i,j) ∈ A wij ai
            aj ← g(inj)
    /* Backpropagation */
    for each neuron j in the output layer do
        Δj ← g'(inj) · (-2(y - aj))           ▷ Modify if different loss function and/or multiple outputs
    for each layer ℓ from L - 1 down to 2 do
        for each neuron j in layer ℓ do
            Δj ← g'(inj) ∑j':(j,j') ∈ A wj,j' Δj'
    /* Weight update */
    for each edge (i, j) ∈ A do
        wij(t+1) ← wij(t) - α(Δj ai)
return w(t+1)
```

CS 457/557: Machine Learning

Lecture 07-4: Neural Network Optimization

Lecture 07-4a: Issues in Neural Network Training: Vanishing Gradient

Quick Recap: Partial Derivatives and Backpropagation

In the last lecture, we saw how to compute the partial derivatives of our cost function $J(\mathbf{w})$ (involving the loss on a single training example) with respect to any weight w_{ij} in the network:

$$\frac{\partial}{\partial w_{ij}} J(\mathbf{w}) = \left[\frac{\partial}{\partial \text{in}_j} J(\mathbf{w}) \right] a_i = \Delta_j a_i.$$

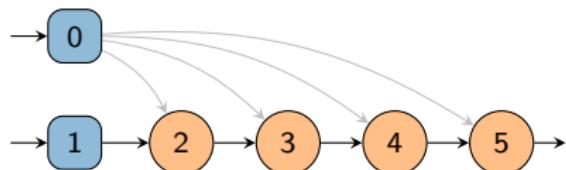
We also saw that

$$\Delta_j = \begin{cases} g'(\text{in}_j)(-2(y - a_j)) & \text{if } j \text{ belongs to output layer} \\ g'(\text{in}_j) \sum_{j':(j,j') \in A} w_{j,j'} \Delta_{j'} & \text{if } j \text{ belongs to hidden layer} \end{cases}$$

The **backpropagation algorithm** provides an efficient way to calculate all a_i and Δ_j values. Let's take a closer look at these formulas.

Computing Δ_j Values

Consider the network below:



For this network, we have:

$$\Delta_j = \begin{cases} g'(in_j)(-2(y - a_j)) & \text{if } j = 5 \\ g'(in_j)w_{j,j+1}\Delta_{j+1} & \text{otherwise} \end{cases}$$

Then:

$$\begin{aligned}\Delta_2 &= [g'(in_2)w_{2,3}] \Delta_3 \\ &= [g'(in_2)w_{2,3}] [g'(in_3)w_{3,4}] \Delta_4 \\ &= [g'(in_2)w_{2,3}] [g'(in_3)w_{3,4}] [g'(in_4)w_{4,5}] \Delta_5 \\ &= [g'(in_2)w_{2,3}] [g'(in_3)w_{3,4}] [g'(in_4)w_{4,5}] [g'(in_5)(-2(y - a_5))]\end{aligned}$$

There are a lot of $g'(\cdot)$ evaluations here!

Vanishing Gradient Problem

From previous slide, we have

$$\Delta_2 = [g'(in_2)w_{2,3}] [g'(in_3)w_{3,4}] [g'(in_4)w_{4,5}] [g'(in_5)(-2(y - a_5))].$$

With a logistic activation function, $g(t) = \sigma(t) = 1 / (1 + e^{-t})$. Then:

- We have $g'(t) = g(t)(1 - g(t))$.
- Recall: $g(t) \in [0, 1]$ always. Therefore, $\max_{t \in \mathbb{R}} g'(t) = 0.25$.
- Repeated multiplication yields values **closer and closer to zero**
- As $\frac{\partial}{\partial w_{1,2}} J(\mathbf{w}) = \Delta_2 a_1$, the partial derivative will be very close to 0
⇒ we only make **very small changes** to $w_{1,2}$

With a ReLU activation function, $g(t) = \max\{0, t\}$. Then:

- We have $g'(t) = 1$ if $t > 0$, 0 if $t < 0$, and undefined if $t = 0$
- Might seem problematic if any $in_j < 0$ anywhere along the chain, but most networks aren't chains, and instead have Δ_j being a sum of $\Delta_{j'}$ values in the next layer

Lecture 07-4b: Stochastic Gradient Descent for Neural Networks

Neural Network Training

Training a Neural Network with Stochastic Gradient Descent

```
procedure NEURALNETTRAIN( $\{(\mathbf{x}_i, y_i)\}_{i=1}^N$ ;  $\mathcal{N} = (V, A)$ )
    /* Weight initialization */
    for each edge  $(i, j) \in A$  do
         $w_{ij}^{(0)} \leftarrow \text{RAND}(-\epsilon, \epsilon)$ 
    /* Stochastic gradient descent */
     $e \leftarrow 0, t \leftarrow 0$ 
    while stopping conditions not met do
        Create random permutation  $P$  of  $\{1, 2, \dots, N\}$ 
        for each  $i$  in  $P$  do
             $\mathbf{w}^{(t+1)} \leftarrow \text{BACKPROPUPDATE}((\mathbf{x}_i, y_i), \mathcal{N}, \mathbf{w}^{(t)})$ 
             $t \leftarrow t + 1$ 
         $e \leftarrow e + 1$                                 ▷ an epoch ends once we process all training examples once
```

- Using the same value for all weights is problematic, so randomize!
- Typical stopping conditions are reaching an epoch limit, getting errors close to zero for all training examples, and/or having minimal change in cost across an epoch

Notes About Stochastic Gradient Descent

When we have **many** training examples, our cost function $J(\mathbf{w})$ is:

$$J(\mathbf{w}) = \frac{1}{N} \sum_{i=1}^N L(y_i, h_{\mathbf{w}}(\mathbf{x}_i)).$$

In **stochastic gradient descent**, we **estimate** the gradient of $J(\mathbf{w})$ instead of computing it exactly.

- SGD weight updates are faster (relative to full batch GD)
- Not all steps we make are directions of descent, though!
 - This may seem **bad**, but it actually allows SGD to potentially **escape from shallow local minima**
 - Neural network cost functions are generally non-convex with lots of local minima
- SGD tends to have poor convergence properties though

Lecture 07-4c: Mini-Batch Gradient Descent for Neural Networks

Mini-Batch Gradient Descent

Recall that **mini-batch gradient descent** offers a compromise between computing the gradient of $J(\mathbf{w})$ exactly versus estimating it using a single training point.

To use mini-batch gradient descent, we need to revisit our partial derivative calculations.

In SGD, we use a single training example (\mathbf{x}, y) with:

$$J(\mathbf{w}) \approx L(y, h_{\mathbf{w}}(\mathbf{x}))$$

$$\begin{aligned}\frac{\partial}{\partial w_{ij}} J(\mathbf{w}) &= \frac{\partial}{\partial w_{ij}} L(y, h_{\mathbf{w}}(\mathbf{x})) \\ &= \Delta_j a_i\end{aligned}$$

where Δ_j and a_i are computed by running backpropagation on training example (\mathbf{x}, y) .

In mini-batch GD, we use a subset of training examples with indices in a set B :

$$\begin{aligned}J(\mathbf{w}) &\approx \frac{1}{|B|} \sum_{n \in B} L(y_n, h_{\mathbf{w}}(\mathbf{x}_n)) \\ \frac{\partial}{\partial w_{ij}} J(\mathbf{w}) &= \frac{\partial}{\partial w_{ij}} \frac{1}{|B|} \sum_{n \in B} L(y_n, h_{\mathbf{w}}(\mathbf{x}_n)) \\ &= \frac{1}{|B|} \sum_{n \in B} \frac{\partial}{\partial w_{ij}} L(y_n, h_{\mathbf{w}}(\mathbf{x}_n)) \\ &= \frac{1}{|B|} \sum_{n \in B} (\Delta_j^{(n)} a_i^{(n)})\end{aligned}$$

where $\Delta_j^{(n)}$ and $a_i^{(n)}$ are the Δ_j and a_i values computed during backpropagation using training example (\mathbf{x}_n, y_n) for all $n \in B$. (So we have to run backpropagation $|B|$ times!)

Training Improvement: Batch Normalization

Batch normalization is a technique that can be used to improve training of neural networks with mini-batch gradient descent. It involves **normalizing** the outputs from each layer of the network before feeding them into the next layer.

For mini-batch B , let $a_j^{(n)}$ be output of neuron j on example n . Then batch normalization updates $a_j^{(n)}$ as

$$\hat{a}_j^{(n)} = \gamma \frac{a_j^{(n)} - \mu}{\sqrt{\epsilon + \sigma^2}} + \beta$$

- μ and σ are the mean and standard deviation of the $a_j^{(n)}$ values within B
- ϵ is a small positive constant to prevent divide by zero
- γ and β are parameters that need to be learned during training

Researchers don't fully understand **why** this tends to work well, though!

Lecture 07-4d: Regularization in Neural Networks

Generalization and Overfitting

Remember the ultimate goal of learning a hypothesis function: we want to **do well on out-of-sample data** (i.e., test data), not just on in-sample training data.

With linear regression, we saw that:

- Using a model that was too simple resulted in **underfitting** (high bias)
- Using a model that was too complex resulted in **overfitting** (high variance)

Having more parameters (i.e., weights) makes it easier to **overfit**.

A fully connected feed-forward neural network can have **lots** of parameters: with p features and p neurons in the first hidden layer, there are p^2 weights just between these first two layers alone!

Because of this, neural networks are prone to overfitting if there is insufficient training data available.

Corrections for Overfitting

Some remedies to overfitting:

- Get more training data
 - Adjusting parameters to memorize the noise on a single training example is likely to increase error on other training examples, so learning algorithm won't do this!
 - This is why “big data” has helped renew interest in neural networks.
- Add regularization! Regularized cost function:

$$J(\mathbf{w}) = \sum_{n=1}^N L(y_n, h_{\mathbf{w}}(\mathbf{x}_n)) + \lambda \sum_{(i,j) \in A} w_{ij}^2$$

Recall: λ is a model hyperparameter that needs to be picked ahead of time (we could use cross-validation for this).

Regularization in Neural Networks: Weight Decay

In the context of stochastic gradient descent where the cost function involves only the loss of a single training example (\mathbf{x}, y) , the partial derivatives of the regularized cost function are given by:

$$\begin{aligned}\frac{\partial}{\partial w_{ij}} J(\mathbf{w}) &= \frac{\partial}{\partial w_{ij}} \left[L(y, h_{\mathbf{w}}(\mathbf{x})) + \lambda \sum_{(i',j') \in A} w_{i'j'}^2 \right] \\ &= \left[\frac{\partial}{\partial w_{ij}} L(y, h_{\mathbf{w}}(\mathbf{x})) \right] + \left[\lambda \sum_{(i',j') \in A} \frac{\partial}{\partial w_{ij}} w_{i'j'}^2 \right] \\ &= \Delta_j a_i + 2\lambda w_{ij}.\end{aligned}$$

Then the weight update is:

$$w_{ij}^{(t+1)} \leftarrow w_{ij}^{(t)} - \alpha \Delta_j a_i - 2\alpha \lambda w_{ij}.$$

This is called **weight decay**, because the weights tend to move towards zero unless pushed back by $\Delta_j a_i$.

CS 457/557: Machine Learning

Lecture 07-5: Additional Notes on Neural Networks

Lecture 07-5a: More on Regularization

Regularization in Neural Networks

One way to reduce overfitting in neural networks is to use L_2 regularization:

$$J(\mathbf{w}) = \sum_{n=1}^N L(y_n, h_{\mathbf{w}}(\mathbf{x}_n)) + \lambda \sum_{(i,j) \in A} w_{ij}^2.$$

Regularization tends to keep weights small. With small weights, in_j doesn't get too far from 0:

$$in_j = \sum_{i:(i,j) \in A} w_{ij} a_i$$

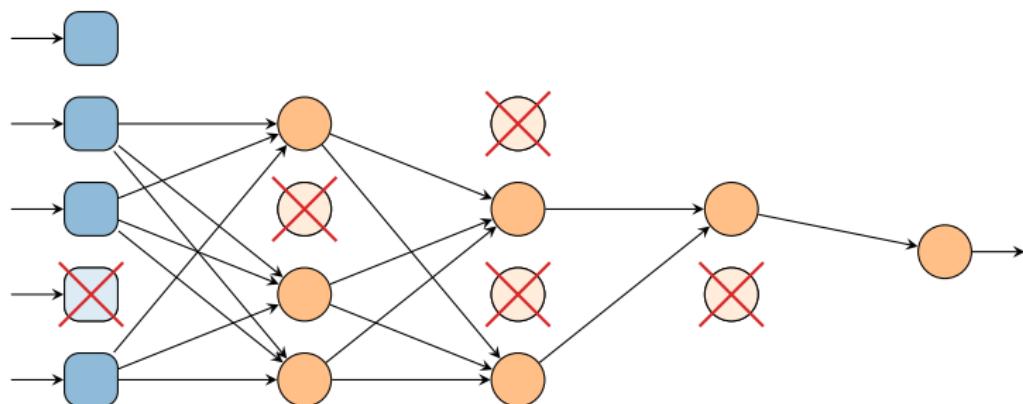
- With a logistic activation function, this helps ensure that $g(in_j)$ is not too close to either 0 or 1, and hence that $g'(in_j)$ isn't close to zero

So weight decay mitigates the vanishing gradient problem!

Dropout Regularization

Dropout regularization is another way to reduce overfitting:

- At the start of each SGD iteration, temporarily drop some nodes in the network (chosen at random)
- Run the backprop update process as before, but rescale the outputs from the surviving neurons in each layer to compensate for the loss of output from the dropped neurons



Benefits of Dropout Regularization

Dropout regularization prevents any neuron from relying too heavily on any one input connection (because it might disappear in the future), and thus makes it such that the weights get distributed across incoming connections.

Key benefits:

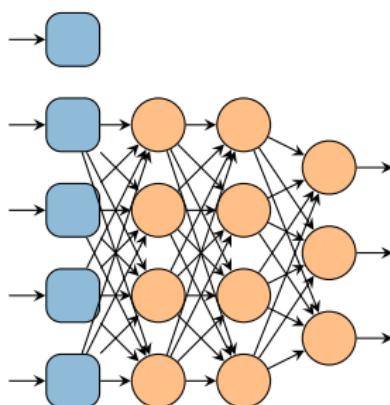
- Avoids overemphasis on single features
- Hidden units forced to work well with others
- Dropout introduces noise which forces the network to be robust to it
- Approximates creating an ensemble of networks instead of just a single one (this idea is more general than this, and is quite useful)

Lecture 07-5c: Regression and Multi-Class Classification

Regression and Multi-Class Classification

We can modify a feedforward neural network to handle regression by removing the activation function g from the neuron in the output layer (and updating the partial derivative calculations appropriately).

For multi-class classification with $K > 2$ classes, we typically use one output neuron per class:



Instead of encoding the output as $y \in \{1, 2, \dots, K\}$, we use one-hot encoding on y . E.g., if $K = 4$, then

$$y = 1 \Rightarrow \mathbf{y} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}, \quad y = 2 \Rightarrow \mathbf{y} = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}, \quad \text{etc.}$$

The squared error loss function is then a sum of squared errors across the classes:

$$L_2(\mathbf{y}, h_{\mathbf{w}}(\mathbf{x})) = \sum_{k=1}^K (y_k - a_k)^2.$$

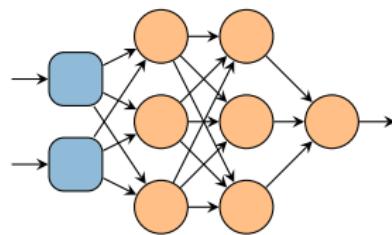
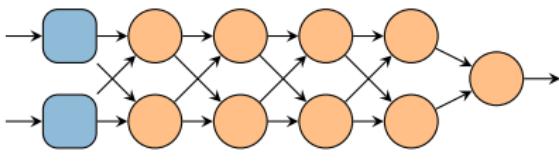
Lecture 07-5c: Neural Net Architectures

Designing a Network

Definition

The layout of a network is called the **network architecture**.

There are **lots** of different ways to structure a network!



- Both networks have 18 weights to learn (excluding the bias weights)
- For a fixed number of weights, deep but narrow tends to do better than shallow but wide

There is not yet a complete understanding for when and/or why some architectures work better than others.

In Search of Good Network Architecture

Figuring out the “right” architecture is kind of like **hyperparameter tuning**, where the number of layers, neurons per layer, and connections serve as hyperparameters.

Lots of performance improvements have come from experimentation with these hyperparamters (a process that is often called GSD for *graduate student descent*).

Definition

Neural architecture search is the process of automatically searching for a good network architecture.

Doing an **exhaustive search** through the hyperparameter values (e.g., via grid search) is infeasible because of the sheer number of different combinations of parameters that are available, so metaheuristics are often used instead.

Neural Architecture Search

Neural architecture search has been conducted using:

- Genetic algorithms and other metaheuristics
- Reinforcement learning approaches
- Gradient descent in network architecture space

Most options require evaluating any given architecture's performance, which generally requires training the architecture first! Training is slow, so various ways to speed it up have been proposed:

- Use smaller training data
- Stop training early and try to predict performance improvements based on trend in cost over epochs
- Use a reduced version of the architecture and hope its performance is comparable to a scaled up version
- Learn a heuristic evaluation function to map network architectures to performance estimates

For Further Reading

Network architectures have also been tailored for specific applications.

Two prominent examples are:

- **Convolutional neural networks** (CNNs) are designed for image data, which is very high-dimensional (each pixel is a feature)
- **Recurrent neural networks** (RNNs) have back edges and feedback loops in the network, which make them suitable for natural language processing and similar tasks

For additional reading:

- *Deep Learning* by Goodfellow et al.
- *Dive into Deep Learning* by Zhang et al.
- Deep Learning Specialization course on Coursera

This looks fun too: <https://playground.tensorflow.org/>

CS 457/557: Machine Learning

Lecture 08-1: Markov Decision Processes

Lecture 08-1a: Reasoning in a Complex Environment

What Do We Want AI to Do?

Short answer: Lots of things!

- Intelligent robot and vehicle navigation
- Better web search
- Automated personal assistants
- Scheduling for delivery vehicles, air traffic control, industrial processes, ...
- Simulated agents in video games
- Automated translation systems

Limitations of Supervised Learning

Consider teaching a machine to play chess.

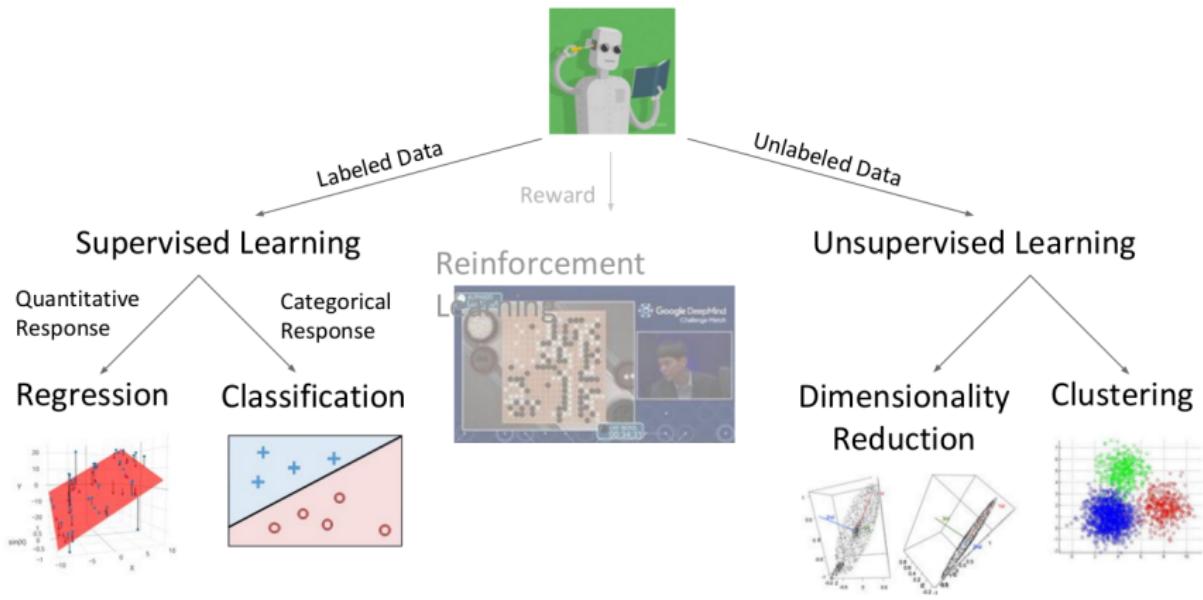
- We could try to take a training set of (board configuration, best move) pairs and learn from that
- But lots of board positions are possible (about 10^{40}), far more than we could possibly include in a training set
- Sooner rather than later, the machine will encounter a configuration that it has no ideas about, and start doing **very poorly**

Not enough data to actually do well here...

“The AI revolution will not be supervised.”

— Yann LeCun and Alyosha Efros

Machine Learning Taxonomy



(Image source: DS 100 lecture notes)

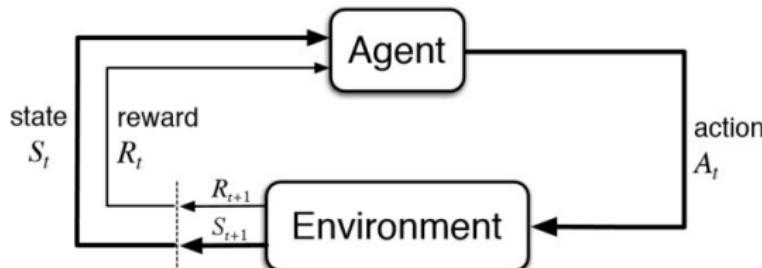
Alternative Learning Methods

Definition

Reinforcement learning is learning what to do—how to map situations to actions—so as to maximize a numerical reward signal. The learner is not told which actions to take, but instead must discover which actions yield the most reward by trying them.

(*Reinforcement Learning* by Sutton and Barto)

Useful for learning how to play games (e.g., Tic-tac-toe, Go)



(Image source: <https://www.kdnuggets.com/2018/03/5-things-reinforcement-learning.html>)

Reinforcement Learning

Instead of providing a machine with labeled training data, reinforcement learning lets the machine explore its environment on its own. Periodically during exploration, the machine will receive a **reward (reinforcement)**. The goal is for the machine to figure out how to maximize its total reward.

Human example: You play a new game without knowing the rules. After a little while you are told that you lost. Then you restart and try again. And again. And again...

- Providing a reward signal is usually much easier than providing labeled examples of how to behave.
- Also, we don't need to know the "correct" answer for every example (as would be necessary in supervised learning)

Components of Reinforcement Learning

Reinforcement learning (RL) considers a machine (agent) operating in a complex and uncertain environment over time with periodic rewards.

- Notion of uncertainty requires **probability theory**
- Notion of rewards requires **value-based planning**, as in decision theory
- Notion of time requires a **temporal model** of how the environment can change

For getting started, we will use a model known as a **Markov decision process!**

- World is made up of states; the states change based on the actions of the agent, which is trying to maximize its long-term rewards
- The state changes include a **stochastic** component (so that the model is non-deterministic)

Lecture 08-1b: Markov Decision Processes

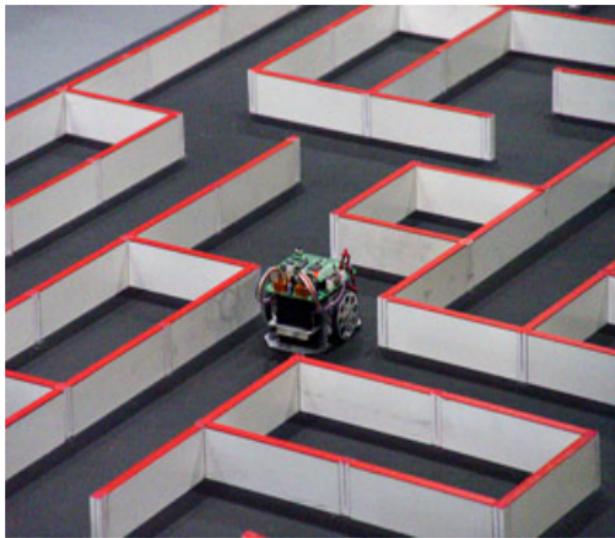
Markov Decision Processes

Definition

A **Markov decision process** (MDP) $M = (S, A, P, R, T)$ is a 5-tuple containing the following components:

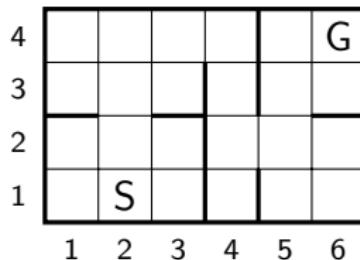
- S is a set of **states** in the world;
- A is a set of **actions** that the agent can take;
- P is a **state-transition function**: $P(s, a, s')$ denotes the probability of ending up in state s' if action a is taken in state s ;
- R is a **reward function**: $R(s, a, s')$ is the one-step reward obtained if the agent enters state s' after taking action a in state s ;
- T is a **time horizon** (i.e., a limit on the number of steps that can be taken).

An Example: Maze Navigation



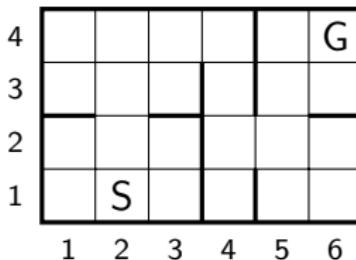
- ▶ Suppose we have a robot in a maze, looking for exit
- ▶ The robot can see where it is currently, and where surrounding walls are, but doesn't know anything else
- ▶ We would like it to be able to learn the shortest route out of the maze, no matter where it starts
- ▶ How can we formulate this problem as an MDP?

Modeling the Maze as an MDP



- States: $S = \{(x, y) \in \mathbb{Z} \times \mathbb{Z} \mid 1 \leq x \leq 6 \wedge 1 \leq y \leq 4\}$
- Actions: $A = \{\text{Left}, \text{Right}, \text{Up}, \text{Down}\}$
- State-transition function:
 - Can include **deterministic** movement, e.g.,
 - $P((2, 1), \text{Up}, (2, 2)) = 1$ and
 - $P((2, 1), \text{Up}, (x, y)) = 0$ for all other $(x, y) \in S$
 - Can be used to enforce walls, e.g., $P((2, 1), \text{Down}, (2, 1)) = 1$

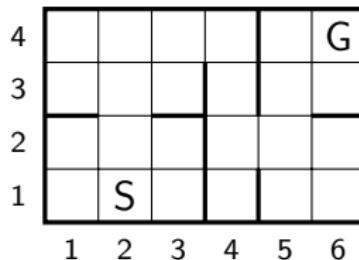
Adding Uncertainty



- The state-transition function can also include **non-deterministic movement**, e.g.:
 - $P((2, 1), \text{Up}, (2, 2)) = 0.8$
 - $P((2, 1), \text{Up}, (1, 1)) = 0.1$
 - $P((2, 1), \text{Up}, (3, 1)) = 0.1$

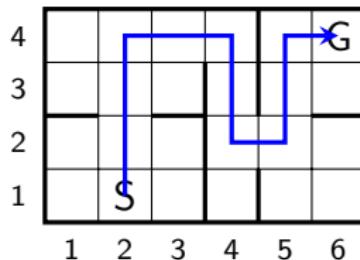
(could represent an agent that staggers occasionally)

Rewards



- Reward function:
 - Can be used to **encourage** the agent to find the goal state G, e.g.:
 - $R((5, 4), \text{Right}, (6, 4)) = +100$
 - $R((6, 3), \text{Up}, (6, 4)) = +100$
 - Can also **discourage** spending too much time in the maze, e.g.:
 - $R((x, y), a, (x', y')) = -1$ for all other states (x, y) and (x', y') in S

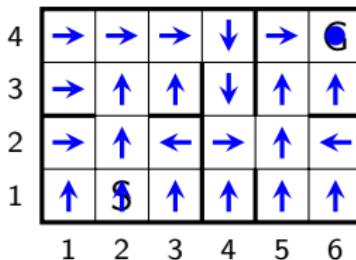
Solving an MDP: The Wrong Way



Providing a **single path** through the maze doesn't constitute a solution.

- What happens if the agent deviates from the path by accident?
- How would it know what to do next?

Solving an MDP: The Correct Way



Definition

For an MDP $M = (S, A, P, R, T)$, a **policy** π is a function from S to A (i.e., $\pi : S \rightarrow A$) that specifies the action to take in each state.

- Having a policy takes care of all possible contingencies
- The goal state is terminal, so no further action gets taken

Lecture 08-1c: Evaluating a Policy

Planning and Learning

- ▶ How do we **find** policies?
- ▶ If we **know the entire problem**, we **plan**
 - ▶ e.g., if we **already know** the whole maze, and know all the MDP dynamics, we can solve it to find the best policy of action (even if we have to take into account the probability that some movements fail some of the time)
- ▶ If we **don't know it all** ahead of time, we **learn**
 - ▶ **Reinforcement Learning**: use the positive and negative feedback from the **one-step reward** in an MDP, and figure out a policy that gives us **long-term** value

Evaluating a Policy

Suppose we start in initial state s_0 and follow policy π . What happens?

We observe some **sequence** of actions and states

$$[s_0, a_0, s_1, a_1, s_2, a_2, s_3, \dots, s_{n-1}, a_{n-1}, s_n]$$

where

- $a_i = \pi(s_i)$ for all $i \in \{0, 1, 2, \dots, n - 1\}$,
- s_n is a terminal state (assuming we reach one).

This sequence is called an **environment history**.

How **good** was the policy π ? We can measure it via the sum of **one-step** rewards obtained during each transition!

$$r_0 = R(s_0, a_0, s_1), \quad r_1 = R(s_1, a_1, s_2), \quad \dots, \quad r_{n-1} = R(s_{n-1}, a_{n-1}, s_n).$$

Issues with Policy Evaluation

An **optimal policy** π is one that **maximizes the total reward** obtained.

Two issues:

- ① The environment history that we obtain is **random** given the probabilistic transitions
- ② The time horizon T may impact our policy

If an MDP M has a **finite time horizon** T , then:

- The optimal policy π may depend on both the current state s **and** the time that is left. Such a policy is called **nonstationary**.
 - E.g., an agent moving through a maze may take a shorter but riskier path if time is scarce, but would prefer a longer but safer path if time is plentiful.
- We can always calculate the total reward obtained from any given environment history $[s_0, a_0, s_1, a_1, s_2, \dots, s_n]$ because $n \leq T$.

Infinite Time Horizon MDPs

If an MDP M has an **infinite time horizon** (i.e., $T = \infty$), then:

- The optimal policy π **only depends** on the current state (i.e., there is no reason to act differently based on the current time). Such a policy is called **stationary**.
- **However**, evaluating the environment history is more complicated!

With an infinite time horizon, the environment history may be infinite if the agent never reaches a terminal state:

$$[s_0, a_0, s_1, a_1, s_2, a_2, s_3, \dots].$$

Then

$$R_{\text{total}} = \sum_{t=0}^{\infty} R(s_t, a_t, s_{t+1}),$$

so the total reward can be **infinitely large** (or **infinitely small**)!

CS 457/557: Machine Learning

Lecture 08-2: Markov Decision Processes

Lecture 08-2a: Discounted Rewards and Uncertainty

Rewards with an Infinite Time Horizon

In an MDP $M = (S, A, P, R, T)$ with $T = \infty$, we saw that an environment history $[s_0, a_0, s_1, a_1, s_2, a_2, s_3, \dots]$ of infinite length can have sum of rewards

$$R_{\text{total}} = \sum_{t=0}^{\infty} R(s_t, a_t, s_{t+1})$$

that is also infinite!

This makes policy comparison **problematic**:

- Suppose $\pi^{(1)}$, $\pi^{(2)}$, and $\pi^{(3)}$ are three policies giving total rewards $+\infty$, $+\infty$, and $-\infty$, respectively
- Obviously we prefer $\pi^{(1)}$ and $\pi^{(2)}$ over $\pi^{(3)}$
- But how should we compare $\pi^{(1)}$ and $\pi^{(2)}$?

Discounted Rewards

To solve the problem of infinite rewards in MDPs, we use a **discount rate** $\gamma \in [0, 1]$. Then the total reward is

$$\begin{aligned} R_{\text{total}} &= R(s_0, a_0, s_1) + \gamma R(s_1, a_1, s_2) + \gamma^2 R(s_2, a_2, s_3) + \dots \\ &= \sum_{t=0}^{\infty} \gamma^t R(s_t, a_t, s_{t+1}). \end{aligned}$$

- If time horizon T is finite, then $\gamma = 1$ gives standard sum of rewards.
- If time horizon T is infinite, then $\gamma < 1$ ensures that

$$R_{\text{total}} = \sum_{t=0}^{\infty} \gamma^t R(s_t, a_t, s_{t+1}) \leq \sum_{t=0}^{\infty} \gamma^t R_{\max} = \frac{R_{\max}}{1 - \gamma},$$

where R_{\max} is the maximum value of any one-step reward.

- If $\gamma = 0$, then we have a greedy algorithm!

Dealing with Uncertainty

With discounted rewards, we can compute the total reward obtained for any **particular** environment history $[s_0, a_0, s_1, a_1, s_2, \dots]$.

Problem: The environment history that we get is **random**!

Solution: Treat states as **random variables**! E.g., let S_t be the random variable representing the state that the agent is in at time t . Then the total reward we get is

$$R_{\text{total}} = \sum_{t=0}^{\infty} \gamma^t R(S_t, A_t, S_{t+1})$$

where A_t is the random variable representing the action taken at time t .

Informally, a **random variable** is a variable whose value depends on some random process (e.g., RV X could represent the value from a die roll).

Probability Background: Random Variables

Motivating example: Let X be a random variable that represents the outcome from rolling a fair six-sided die.

- **Before we roll** the die, we don't know what value X will have!
- But we **can** talk about possible outcomes and their likelihoods.

Definition

The **range** of a (discrete) random variable is the set of values that the random variable can have. Each value in the range has an associated likelihood, or probability, of occurring.

For above, $\text{range}(X) = \{1, 2, 3, 4, 5, 6\}$, and each value is **equally likely**:

$$P(X = 1) = \frac{1}{6}, P(X = 2) = \frac{1}{6}, \dots, P(X = 6) = \frac{1}{6},$$

where $P(X = x)$ denotes the probability of X having value x .

Probability Background: Expected Value

Definition

For a (discrete) random variable X , the **expected value** of X , denoted $E[X]$, is

$$E[X] = \sum_{x \in \text{range}(X)} x \cdot P(X = x).$$

The value $E[X]$ is a weighted average of the possible values that X might have, where the values are weighted by their likelihood.

For the die-rolling example from the previous slide, we have:

$$E[X] = \sum_{i=1}^6 i \cdot P(X = i) = \sum_{i=1}^6 i \cdot \frac{1}{6} = \frac{21}{6} = 3.5.$$

(So the expected value of a random variable is just a number, and the number might not even be a value that the random variable itself could have.)

Lecture 08-2b: The State-Value Function

The Value of a State

For an MDP M and policy π , we can define the **state-value function** $V^\pi : S \rightarrow \mathbb{R}$ where $V^\pi(s)$ is the expected total reward (expected utility) obtained by executing π starting from **initial state** s :

$$V^\pi(s) = E [R_{\text{total}} \mid S_0 = s] = E \left[\sum_{t=0}^{\infty} \gamma^t R(S_t, \pi(S_t), S_{t+1}) \mid S_0 = s \right],$$

where the expectation is taken over the probability distribution of all environment histories generated by π starting from s .

(Intuitively, this is the average total reward across **all possible histories**.)

- $V^\pi(s)$ tells us how **desirable** it is to be in state s when following policy π
- We can use this information to **evaluate** (and also **improve!**) a policy

Looking at the Next State

Using the state-value function $V^\pi(s)$

$$\begin{aligned} V^\pi(s) &= E \left[\sum_{t=0}^{\infty} \gamma^t R(S_t, \pi(S_t), S_{t+1}) \mid S_0 = s \right] \\ &= E_{S_1} \left\{ E \left[\sum_{t=0}^{\infty} \gamma^t R(S_t, \pi(S_t), S_{t+1}) \mid S_0 = s, S_1 \right] \right\} \\ &= \sum_{s' \in S} P(S_1 = s' \mid S_0 = s) \cdot E \left[\sum_{t=0}^{\infty} \gamma^t R(S_t, \pi(S_t), S_{t+1}) \mid S_0 = s, S_1 = s' \right] \\ &= \sum_{s' \in S} P(s, \pi(s), s') \cdot E \left[\sum_{t=0}^{\infty} \gamma^t R(S_t, \pi(S_t), S_{t+1}) \mid S_0 = s, S_1 = s' \right] \end{aligned}$$

- Here we are using the **law of total expectation** to condition on the possible options for state S_1
- Slight notational overlap: in $P(S_1 = s' \mid S_0 = s)$, the P denotes a (conditional) probability, not the state-transition function P from the MDP

Some Further Derivations

Expanding out the expected value portion:

$$\begin{aligned} & E \left[\sum_{t=0}^{\infty} \gamma^t R(S_t, \pi(S_t), S_{t+1}) \mid S_0 = s, S_1 = s' \right] \\ &= E \left[R(S_0, \pi(S_0), S_1) + \sum_{t=1}^{\infty} \gamma^t R(S_t, \pi(S_t), S_{t+1}) \mid S_0 = s, S_1 = s' \right] \\ &= E \left[R(s, \pi(s), s') + \gamma \sum_{t=1}^{\infty} \gamma^{t-1} R(S_t, \pi(S_t), S_{t+1}) \mid S_0 = s, S_1 = s' \right] \\ &= R(s, \pi(s), s') + \gamma E \left[\sum_{t=1}^{\infty} \gamma^{t-1} R(S_t, \pi(S_t), S_{t+1}) \mid S_0 = s, S_1 = s' \right] \\ &= R(s, \pi(s), s') + \gamma E \left[\sum_{t=0}^{\infty} \gamma^t R(S_t, \pi(S_t), S_{t+1}) \mid S_0 = s' \right] \\ &= R(s, \pi(s), s') + \gamma V^\pi(s') \end{aligned}$$

where the last two lines follow by shifting the state indices down by 1 and then recognizing that the expected value is just the state-value function evaluated at s' .

The Bellman Equation

Combining this all together then gives the **Bellman equation**:

$$V^\pi(s) = \sum_{s' \in S} P(s, \pi(s), s') \cdot [R(s, \pi(s), s') + \gamma V^\pi(s')] .$$

- $V^\pi(s)$: expected value of following policy π starting in state s
- $\sum_{s' \in S}$: Sum over all possible next states
- $P(s, \pi(s), s')$: transition probability of going from s to s' following action $\pi(s)$
- $R(s, \pi(s), s')$: One-step reward for going from s to s' following action $\pi(s)$
- $\gamma V^\pi(s')$: discounted expected value of continuing to follow policy π starting from state s'

This is a **recursive** definition for the state-value function V^π which says that the value of starting in state s can be calculated based on the values of the **next** possible state(s) that can be reached by taking the action dictated by the policy π .

Lecture 08-2c: Policy Evaluation

Solving the Bellman Equation

Suppose we have $S = \{s_1, s_2, \dots, s_n\}$ for our MDP M . If we write the Bellman equation out for each state, then we get:

$$V^\pi(s_1) = \sum_{i=1}^n P(s_1, \pi(s_1), s_i) \cdot [R(s_1, \pi(s_1), s_i) + \gamma V^\pi(s_i)]$$

$$V^\pi(s_2) = \sum_{i=1}^n P(s_2, \pi(s_2), s_i) \cdot [R(s_2, \pi(s_2), s_i) + \gamma V^\pi(s_i)]$$

⋮

$$V^\pi(s_n) = \sum_{i=1}^n P(s_n, \pi(s_n), s_i) \cdot [R(s_n, \pi(s_n), s_i) + \gamma V^\pi(s_i)]$$

This is a system of $|S|$ equations in $|S|$ unknowns (the $V^\pi(s)$ values).

- We can solve this system of equations in $O(|S|^3)$ time.
- For large S , we'll use an **iterative approach** to find the $V^\pi(s)$ values, similar to how we used gradient descent to find the optimal weights in linear regression instead of solving the normal equations directly.

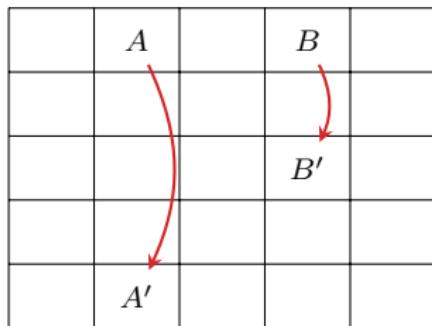
Evaluating a Policy Iteratively

Evaluating a Policy for an MDP

```
procedure POLICYEVALUATION(MDP  $M$ ; Policy  $\pi$ ; Discount factor  $\gamma$ ; Threshold  $\Theta$ )
    for each state  $s \in S$  do
         $v_s^{(0)} \leftarrow 0$                                  $\triangleright v_s^{(t)}$  is estimate of  $V^\pi(s)$  at iteration  $t$ 
         $t \leftarrow 0$ 
    repeat
         $\Delta \leftarrow 0$ 
        for each state  $s \in S$  do
             $v_s^{(t+1)} \leftarrow \sum_{s' \in S} P(s, \pi(s), s') [R(s, \pi(s), s') + \gamma v_s^{(t)}]$        $\triangleright$  Update  $v_s$  estimate
             $\Delta \leftarrow \max \{ \Delta, |v_s^{(t+1)} - v_s^{(t)}| \}$            $\triangleright$  Update max change in estimates
         $t \leftarrow t + 1$ 
    until  $\Delta \leq \Theta$                                  $\triangleright$  If  $\Theta = \frac{\epsilon(1-\gamma)}{\gamma}$ , then error is at most  $\epsilon$ 
    for each state  $s \in S$  do
         $V^\pi(s) \leftarrow v_s^{(t)}$                        $\triangleright$  For  $\epsilon$  sufficiently small,  $v_s^{(t)} \approx V^\pi(s)$ 
    return  $V^\pi$ 
```

The above is an **iterative process** for finding a function that satisfies the Bellman equation, which is exactly the **state-value** function V^π .

Using the State-Value Function



3.3	8.8	4.4	5.3	1.5
1.5	3.0	2.3	1.9	0.5
0.1	0.7	0.7	0.4	-0.4
-1.0	-0.4	-0.4	-0.6	-1.2
-1.9	-1.3	-1.2	-1.4	-2.0

Simple grid world:

- Policy: Agent moves randomly (Up, Down, Left, Right)
- Any action in states A or B causes move to state A' and B' , respectively
- Rewards:
 - Hitting a wall incurs reward of -1 (and no movement)
 - Moving from A to A' has reward of $+10$; from B to B' is $+5$
 - All other movement has reward of 0

State-value function V^π values shown on right with discount factor $\gamma = 0.9$.

Even with a **random policy**, we can see what states **look promising**!

CS 457/557: Machine Learning

Lecture 08-3: Finding Optimal Policies

Lecture 08-3a: Finding an Optimal Policy: Policy Iteration

Quick Recap: Bellman Equations and the State-Value Function

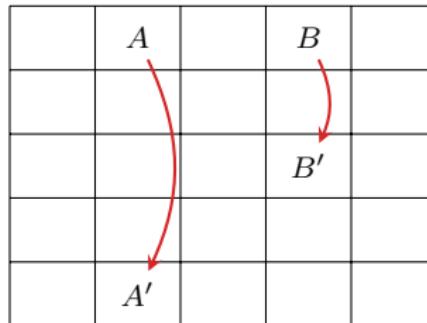
The Bellman equation specifies the state-value function V^π for a policy π recursively:

$$V^\pi(s) = \sum_{s' \in S} P(s, \pi(s), s') \cdot [R(s, \pi(s), s') + \gamma V^\pi(s')] .$$

For any policy π , we can compute the $V^\pi(s)$ values by

- ① Solving a system of $|S|$ equations in $|S|$ unknowns
- ② Using an iterative process to find a solution

The below grid world uses a **random** movement policy, with state-value function values shown on the right:



3.3	8.8	4.4	5.3	1.5
1.5	3.0	2.3	1.9	0.5
0.1	0.7	0.7	0.4	-0.4
-1.0	-0.4	-0.4	-0.6	-1.2
-1.9	-1.3	-1.2	-1.4	-2.0

Once we know the values of states (even for a **bad** policy), we can try to **improve** the policy!

Additional Notation: The Action-Value Function

The state-value function V^π tells us the **expected total reward** for following policy π starting from state s :

$$V^\pi(s) = \sum_{s' \in S} P(s, \pi(s), s') \cdot [R(s, \pi(s), s') + \gamma V^\pi(s')] .$$

We can also define the **action-value function** $Q^\pi : S \times A \rightarrow \mathbb{R}$ as

$$Q^\pi(s, a) = \sum_{s' \in S} P(s, a, s') \cdot [R(s, a, s') + \gamma V^\pi(s')] .$$

That is, $Q^\pi(s, a)$ is the **expected total reward** of

- **first** taking action a in state s , **regardless of policy**, and
- **then** following policy π in all subsequent steps.

We'll use the Q function to figure out the best actions to take!

Policy Improvement: Finding Better Actions

When creating a **new policy** π' from π , we pick, for each state, the action that yields the **most perceived benefit** using one-step look-ahead:

$$\begin{aligned}\pi'(s) &= \arg \max_{a \in A} \{Q^\pi(s, a)\} \\ &= \arg \max_{a \in A} \left\{ \sum_{s' \in S} P(s, a, s') \cdot [R(s, a, s') + \gamma V^\pi(s')] \right\}.\end{aligned}$$

- This is a **greedy** choice based on the **current** V^π state values
- If $\pi' \neq \pi$, then $V^{\pi'}(s) \neq V^\pi(s)$, and so we need to update the state-value function for new policy π'

Policy Iteration Algorithm

Policy Iteration for Solving an MDP

```
procedure POLICYITERATION(MDP  $M$ ; Initial Policy  $\pi$ ; Discount factor  $\gamma$ ; Threshold  $\Theta$ )
 $\pi^{(0)} \leftarrow \pi$ 
 $t \leftarrow 0$ 
do
     $V^{(t)} \leftarrow \text{POLICYEVALUATION}(M, \pi^{(t)}, \gamma, \Theta)$             $\triangleright V^{(t)}$  is state-value func. for  $\pi^{(t)}$ 
    policyChanged  $\leftarrow$  false
    for each state  $s \in S$  do
         $\pi^{(t+1)}(s) \leftarrow \arg \max_{a \in A} \left\{ \sum_{s' \in S} P(s, a, s') \cdot [R(s, a, s') + \gamma V^{(t)}(s')] \right\}.$ 
        if  $\pi^{(t+1)}(s) \neq \pi^{(t)}(s)$  then
            policyChanged  $\leftarrow$  true
     $t \leftarrow t + 1$ 
while (policyChanged)
return  $\pi^{(t)}$ 
```

This is an **iterative process** for finding an **optimal policy** π^* for the given MDP.

- The POLICYEVALUATION procedure computes the state-value function V^π for the current policy as before

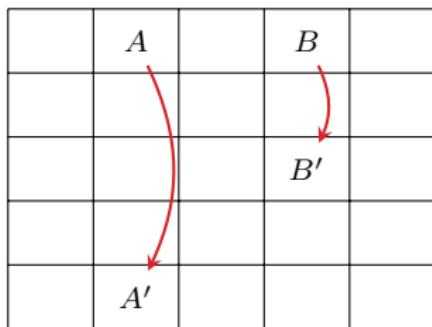
Policy Iteration Convergence

For any MDP M , the POLICYITERATION procedure is **guaranteed to converge** to an **optimal** policy π^* :

- If the current policy is not optimal, then the policy update process will find a better policy
- For an MDP with finite state and action spaces, the number of possible policies is finite, $|A|^{|S|}$
- Eventually, we'll arrive at the best policy!

There are a few caveats for numerical issues and tolerances if we use the iterative procedure for estimating the state-value functions $V^{(t)}$, but these can be removed if we compute $V^{(t)}$ exactly by solving the system of Bellman equations.

Example with Optimal Policy



22.0	24.4	22.0	19.4	17.5
19.8	22.0	19.8	17.8	16.0
17.8	19.8	17.8	16.0	14.4
16.0	17.8	16.0	14.4	13.0
14.4	16.0	14.4	13.0	11.7

Simple grid world example with policy iteration.

The top right shows the state values for the optimal policy.

The bottom right shows the permissible actions in an optimal policy.

→	↔	←	↔	←
↑→	↑	↑←	↑←	↑←
↑→	↑	↑←	↑←	↑←
↑→	↑	↑↔	↑↔	↑↔
↑→	↑	↑←	↑←	↑←

Lecture 08-3b: Bellman Optimality Equations and Value Iteration

Optimal State Values

We saw earlier that the state-value function $V^\pi : S \rightarrow \mathbb{R}$ tells us the **expected total reward** when starting in state s and following policy π :

$$V^\pi(s) = \sum_{s' \in S} P(s, \pi(s), s') \cdot [R(s, \pi(s), s') + \gamma V^\pi(s')] .$$

The **optimal state-value function** $V^* : S \rightarrow \mathbb{R}$ tells us the **best possible expected total reward** when starting in state s , across **all possible policies**:

$$V^*(s) = \max_{\pi} V^\pi(s).$$

We are going to use the above equation to find an optimal policy in another way!

The Bellman Optimality Equation

For the optimal state-value function V^* , we have:

$$\begin{aligned} V^*(s) &= \max_{\pi} V^{\pi}(s) \\ &= \max_{\pi} \left[\sum_{s' \in S} P(s, \pi(s), s') \cdot [R(s, \pi(s), s') + \gamma V^{\pi}(s')] \right] \\ &= \max_{a \in A} \left[\sum_{s' \in S} P(s, a, s') \cdot [R(s, a, s') + \gamma \max_{\pi} V^{\pi}(s')] \right] \\ &= \max_{a \in A} \left[\sum_{s' \in S} P(s, a, s') \cdot [R(s, a, s') + \gamma V^*(s')] \right] \end{aligned}$$

This is the **Bellman optimality equation**.

- Also a recursive formula, just like the Bellman equation!
- The $\max_{a \in A}$ component complicates evaluation, though.

Comparing Bellman Equations

The **Bellman equation** for a fixed policy π is

$$V^\pi(s) = \sum_{s' \in S} P(s, \pi(s), s') \cdot [R(s, \pi(s), s') + \gamma V^\pi(s')] .$$

We saw earlier how to use V^π for the current policy to find a **better** policy.

The **Bellman optimality equation** is

$$V^*(s) = \max_{a \in A} \left\{ \sum_{s' \in S} P(s, a, s') \cdot [R(s, a, s') + \gamma V^*(s')] \right\} .$$

This doesn't require policies at all! Once we have V^* , we can find an **optimal** policy using

$$\pi^*(s) = \arg \max_{a \in A} \left\{ \sum_{s' \in S} P(s, a, s') \cdot [R(s, a, s') + \gamma V^*(s')] \right\} .$$

Solving the Bellman Optimality Equations: Value Iteration

Value Iteration for Solving an MDP

```
procedure VALUEITERATION(MDP  $M$ ; Discount factor  $\gamma$ ; Threshold  $\Theta$ )
    for each state  $s \in S$  do
         $v_s^{(0)} \leftarrow 0$                                  $\triangleright v_s^{(t)}$  is estimate of  $V^*(s)$  at iteration  $t$ 
         $t \leftarrow 0$ 
    do
         $\Delta \leftarrow 0$ 
        for each state  $s \in S$  do
             $v_s^{(t+1)} \leftarrow \max_{a \in A} \left\{ \sum_{s' \in S} P(s, a, s') [R(s, a, s') + \gamma v_s^{(t)}] \right\}$        $\triangleright$  Update  $v_s$  estimate
             $\Delta \leftarrow \max \left\{ \Delta, \left| v_s^{(t+1)} - v_s^{(t)} \right| \right\}$            $\triangleright$  Update max change in estimates
         $t \leftarrow t + 1$ 
    while ( $\Delta > \Theta$ )                                 $\triangleright$  If  $\Theta = \frac{\epsilon(1-\gamma)}{\gamma}$ , then error is at most  $\epsilon$ 
    for each state  $s \in S$  do
         $V^*(s) \leftarrow v_s^{(t)}$                        $\triangleright$  For  $\epsilon$  sufficiently small,  $v_s^{(t)} \approx V^*(s)$ 
    return  $V^*$ 
```

Two Methods for Solving MDPs: Policy Iteration and Value Iteration

Policy iteration:

- ① Maintains current policy $\pi^{(t)}$
- ② Solves the (regular) Bellman equations for policy $\pi^{(t)}$ to obtain state-value function $V^{\pi^{(t)}}$ (or $V^{(t)}$)
 - Solving is done iteratively (e.g., using POLICYEVALUATION) or exactly via linear algebra
- ③ Generates a new policy $\pi^{(t+1)}$ by using the state values of the current policy
- ④ Repeats until no changes are made

Each method has pros and cons; policy iteration tends to work well for small state spaces, but value iteration is better for larger ones.

Value iteration:

- ① Does not maintain a current policy
- ② Solves the Bellman **optimality** equations to obtain the optimal state-value function V^*
 - Solving is done iteratively (as on previous slide) or via linear programming
- ③ Uses V^* to generate an **optimal** policy by picking the best available action in each state

Lecture 08-3c: The Optimal Action-Value Function

The Optimal Action-Value Function

With the Bellman equation, we have the state-value function V^π and action-value function Q^π :

$$V^\pi(s) = \sum_{s' \in S} P(s, \pi(s), s') \cdot [R(s, \pi(s), s') + \gamma V^\pi(s')]$$

$$Q^\pi(s, a) = \sum_{s' \in S} P(s, a, s') \cdot [R(s, a, s') + \gamma V^\pi(s')].$$

With the Bellman optimality equations, we have the **optimal state-value function** V^* and **optimal action-value function** Q^* :

$$V^*(s) = \max_{a \in A} \left\{ \sum_{s' \in S} P(s, a, s') \cdot [R(s, a, s') + \gamma V^*(s')] \right\}$$

$$Q^*(s, a) = \sum_{s' \in S} P(s, a, s') \cdot [R(s, a, s') + \gamma V^*(s')].$$

It follows that $V^*(s) = \max_{a \in A} \{Q^*(s, a)\}$.

Additional Notes on the Optimal Action-Value Function

- ① We can derive the optimal policy π^* from Q^* using:

$$\begin{aligned}\pi^*(s) &= \arg \max_{a \in A} \left\{ \sum_{s' \in S} P(s, a, s') \cdot [R(s, a, s') + \gamma V^*(s')] \right\} \\ &= \arg \max_{a \in A} \{Q^*(s, a)\}\end{aligned}$$

- ② The observation that $V^*(s) = \max_{a \in A} Q^*(s, a)$ allows us to write the optimal action-value function recursively:

$$\begin{aligned}Q^*(s, a) &= \sum_{s' \in S} P(s, a, s') \cdot [R(s, a, s') + \gamma V^*(s')] \\ &= \sum_{s' \in S} P(s, a, s') \cdot \left[R(s, a, s') + \gamma \max_{a' \in A} \{Q^*(s', a')\} \right].\end{aligned}$$

Lecture 08-3d: More Complex Models

Partially Observable MDPs (POMDPs)

- ▶ The basic MDP model assumes that the agent **fully observes** their current state, and can therefore follow any policy with certainty
- ▶ Not all AI planning problems are accurately modeled in this way, since we may have, among other things:
 1. Incomplete information
 2. Incorrect information
 3. Noisy sensors
- ▶ The POMDP is a model for some of these situations

Formal Definition of a POMDP

- ▶ A POMDP extends the MDP by adding two new components:

$$M = \langle S, A, P, \Omega, O, R, T \rangle$$

1. S = a set of states of the world
2. A = a set of actions an agent can take
3. P = a state-transition function: $P(s, a, s')$ is the probability of state s' , starting in state s and taking action a : $P(s'|s, a)$
4. Ω = a set of **observations** an agent can make
5. O = an **observation function**: $O(s, e)$ is the probability of observing evidence e when in state s : $P(e|s)$
6. R = a reward function: $R(s, a, s')$ is the one-step reward you get if you go from state s to state s' after taking action a
7. T = a time horizon: we assume that every state-transition, following a single action, takes a single unit of time

Tractability of the Models

- ▶ Basic complexity analysis has shown that there are tractable algorithms for finite/infinite horizon MDPs
 - ▶ Linear programming techniques are among the most efficient
 - ▶ Very complex MDPs for real-world problems (including control of commercial aircraft) have been solved optimally
- ▶ POMDPs are considerably more complex
 - ▶ Conversion to a “belief-state” MDP produces models that are generally intractable to solve
 - ▶ It has been shown that finding optimal policies for infinite-horizon POMDPs is non-computable
 - ▶ Many advances have been made in optimal algorithms for finite-horizon problems, and in the use of approximation methods

Even More Complex Models

- ▶ Both MDPs and POMDPs are **single-agent** models
 - ▶ In each case, a single decision-maker is assumed to be acting
- ▶ Things get much more complicated once **multiple agents** are involved, and must interact with one another
 1. **Competing** agents: when agents have distinct reward functions, and do not always share the same interests, then **game-theoretic** techniques come into play
 2. **Cooperating** agents: when agents share a common reward function, and want to work as a team, it can be shown that the complexity of finding optimal/approximate plans increases greatly, making such planning infeasible almost always

CS 457/557: Machine Learning

Lecture 09-1: Passive Reinforcement Learning

Lecture 09-1a: Passive Reinforcement Learning

Moving Into the Unknown...

Markov decision processes can be used to model **known environments**:

- S : A set of states
- A : A set of actions
- P : Probabilities of all state transitions for all actions
- R : Rewards for all state transitions for all actions
- T : Time horizon

We saw how to **solve** an MDP via the Bellman optimality equation:

$$V^*(s) = \max_{a \in A} \left\{ \sum_{s' \in S} P(s, a, s') \cdot [R(s, a, s') + \gamma V^*(s')] \right\}.$$

In reinforcement learning proper, things aren't quite so easy though!

Basic Ideas in RL

As a starting point, we can think of reinforcement learning as an MDP where we **don't know** the state-transition probabilities P and the one-step rewards R .

Despite this lack of information, we would like to be able to:

- ① Evaluate the quality of any given policy $\pi : S \rightarrow A$
(passive reinforcement learning)
- ② Find an optimal (or at least good) policy
(active reinforcement learning)

To make this happen, we need to **learn**!

We say that a machine learns with respect to a particular task T , performance metric P , and type of experience E , if the system reliably improves its performance P at task T , following experience E .

— Mitchell, *The Discipline of Machine Learning*

Learning from Experience

The only winning move is not to play.

You can't win if you don't play.

We **need to play to learn!** General idea:

- Perform many **learning episodes** (trials)
- For each episode:
 - ① Begin in a start state
 - ② Execute an action (possibly dictated by policy) and move to a next state, collecting a one-step reward along the way
 - ③ Repeat until the episode ends
- Based on experiences in each episode, update estimates of:
 - ① the state-transition probabilities P and reward function R
 - ② the state-value function V^π

(such updates may happen during or after each episode)

“Follow the policy, and see what happens”

Lecture 09-1b: Monte Carlo Methods

Monte Carlo Methods

Basic idea is to play many episodes, and record observations during each episode.

① Method One: in each episode, record:

- The states that get visited
- The one-step rewards for each transition

At end of episode, do the following for each visited state:

- Compute the sum of rewards earned *after* the first visit to the state
- Estimate $V(s)$ as this sum of rewards

Use $V(s)$ to make decisions (e.g., for policy improvement)

② Option 2: in each episode, record, for each $(s, a, s') \in S \times A \times S$:

- The number of times we visited s and did a
- The number of times we ended up in state s' after this
- The reward we obtained by entering s' from s by a

Estimate $P(s, a, s')$ and $R(s, a, s')$ from this data, and then apply MDP methods with estimates of P and R

Lecture 09-1c: Temporal Difference Learning

Learning a State-Value Function: Temporal Difference Updates

Passive Reinforcement Learning: Evaluating a Policy

```
procedure TD POLICY EVALUATION(MDP  $M$ ; Policy  $\pi$ ; Discount factor  $\gamma$ ; Step size  $\alpha$ )
  for each state  $s \in S$  do
     $V(s) \leftarrow 0$                                  $\triangleright V(s)$  is current estimate of  $V^\pi(s)$ 
    for each episode do
       $s \leftarrow s_0$                              $\triangleright s$  is current state,  $s_0$  is fixed start state
      while episode has not ended do
        Execute action  $\pi(s)$                        $\triangleright \pi$  may be a random action policy
        Observe next state  $s'$  and one-step reward  $r = R(s, \pi(s), s')$ 
         $V(s) \leftarrow V(s) + \alpha [r + \gamma V(s') - V(s)]$      $\triangleright$  Update estimated value of state  $s$ 
         $s \leftarrow s'$                                  $\triangleright$  Move to next state
    return  $V$                                      $\triangleright V(s) \approx V^\pi(s)$  for all  $s \in S$ 
```

This is called the **temporal difference** update method:

- Use the estimated value of the state at the **next time step** $V(s')$ to update the estimated value of the state at **current time step** $V(s)$

The Basic TD(0) Update

The TD(0) update:

$$V(s) = V(s) + \alpha [r + \gamma V(s') - V(s)]$$

Every time we leave a state s , we update its estimated value based on

- the one-step reward we get, $r = R(s, \pi(s), s')$
- the difference in the (discounted) value of where we end up, $V(s')$, and the value of where we start, $V(s)$

Intuition:

- If $V(s') > V(s)$, then the value of s tends to go **up**
- If $V(s') < V(s)$, then the value of s tends to go **down**

The value of the current state should be **comparable** to the value of the next state, after factoring in the one-step reward and discounting.

Managing the Updates

The **step-size parameter** α controls how quickly we update:

$$\begin{aligned}V(s) &= V(s) + \alpha [r + \gamma V(s') - V(s)] \\&= (1 - \alpha)V(s) + \alpha [r + \gamma V(s')]\end{aligned}$$

Recall the Bellman equation:

$$\begin{aligned}V^\pi(s) &= \sum_{s'' \in S} P(s, a, s'') \cdot [R(s, a, s'') + \gamma V^\pi(s'')] \\&\approx R(s, a, s') + \gamma V^\pi(s') \quad \text{if } P(s, a, s') = 1\end{aligned}$$

- If $\alpha = 1$, then the TD(0) update looks like a Bellman update where we assume $P(s, a, s') = 1$ for the one state s' that we actually observe!
- If α shrinks over time, then eventually the values stop changing.
 - By decreasing α slowly over time, we can ensure that the $V(s)$ values **converge** to the actual $V^\pi(s)$ values

The expression $r + \gamma V(s')$ is the **TD target**, and $r + \gamma V(s') - V(s)$ is the **TD error**.

Pros and Cons of TD Updates

With TD updates:

- We can estimate the values of states **without** knowing the actual state-transition probabilities $P(s, a, s')$
- We only update the value of **one state** at a time
- We don't know the values of states that we **never reach**

This last point may be advantageous if these states aren't interesting and/or if the current policy would always avoid them.

However, if these unknown states are **actually useful**, then this can be problematic for doing policy improvement. We'll revisit this idea in more detail later on.

Lecture 09-1d: Towards Active Reinforcement Learning

Finding Better Policies in RL

The preceding discussion focused on **passive reinforcement learning**: figuring out how to evaluate a **fixed policy** π .

In **active reinforcement learning**, we are interested in finding an **optimal** (or at least good) policy.

- Still perform episodes to acquire experiences
- Agent should improve its policy over time

How do we make this happen?

Policy Improvement Idea

Recall in policy improvement, we generate a new policy π' from the current policy π using

$$\pi'(s) = \arg \max_{a \in A} \left\{ \sum_{s' \in S} P(s, a, s') \cdot [R(s, a, s') + \gamma V^\pi(s')] \right\}.$$

With TD(0) updates, the agent **learns** V^π for its current policy π .

Problem: Without knowing P and R , the agent cannot figure out how to pick better actions!

One Solution: Estimate P and R from the data

Another Solution: Use the action-value function Q^π instead!

Policy Improvement via the Action-Value Function

The policy improvement process can be reformulated in terms of the action-value function Q^π instead of the state-value function V^π :

$$\begin{aligned}\pi'(s) &= \arg \max_{a \in A} \left\{ \sum_{s' \in S} P(s, a, s') \cdot [R(s, a, s') + \gamma V^\pi(s')] \right\} \\ &= \arg \max_{a \in A} \{Q^\pi(s, a)\}.\end{aligned}$$

So if we know Q^π , we can figure out how to create a potentially better policy π' from it, **without** needing to know P or R !

We can use the same **temporal difference update** that we saw earlier for V^π to help learn Q^π .

TD Updates for the Action-Value Function

Recall the definitions of the state-value and action-value functions:

$$V^\pi(s) = \sum_{s' \in S} P(s, \pi(s), s') \cdot [R(s, \pi(s), s') + \gamma V^\pi(s')]$$

$$Q^\pi(s, a) = \sum_{s' \in S} P(s, a, s') \cdot [R(s, a, s') + \gamma V^\pi(s')]$$

We have that $V^\pi(s) = Q^\pi(s, \pi(s))$, so

$$Q^\pi(s, a) = \sum_{s' \in S} P(s, a, s') \cdot [R(s, a, s') + \gamma Q^\pi(s', \pi(s'))].$$

During **learning**, when we take action a in state s and end up in state s' with one-step reward r , we treat $P(s, a, s') = 1$ and $R(s, a, s') = r$, leading to the approximation

$$Q^\pi(s, a) \approx r + \gamma Q^\pi(s', \pi(s')).$$

TD Updates for Action-Values

Temporal difference updates for Q^π are similar to those for V^π :

TD Updates for Action Values: Evaluating a Policy

```
procedure TDPOLEMULATION(MDP  $M$ ; Policy  $\pi$ ; Discount factor  $\gamma$ ; Step size  $\alpha$ )
  for each  $(s, a) \in S \times A$  do
     $Q(s, a) \leftarrow 0$                                  $\triangleright Q(s, a)$  is current estimate of  $Q^\pi(s, a)$ 
  for each episode do
     $s \leftarrow s_0$                                  $\triangleright s$  is current state,  $s_0$  is fixed start state
     $a \leftarrow \pi(s)$                                  $\triangleright a$  is next action to take
    while episode has not ended do
      Execute action  $a$ 
      Observe next state  $s'$  and one-step reward  $r = R(s, a, s')$ 
      Determine  $a' \leftarrow \pi(s')$                        $\triangleright a'$  is what we're going to do next
       $Q(s, a) \leftarrow Q(s, a) + \alpha [r + \gamma Q(s', a') - Q(s, a)]$ 
       $s \leftarrow s'$ ,  $a \leftarrow a'$                            $\triangleright$  Update current state and next action
  return  $Q$                                           $\triangleright Q(s, a) \approx Q^\pi(s, a)$  for all  $(s, a) \in S \times A$ 
```

CS 457/557: Machine Learning

Lecture 09-2: Active Reinforcement Learning

Lecture 09-2a: Greedy Policy Improvement

Towards an Optimal Policy

We can use the following process to find an optimal policy:

- ① Start with initial policy π (possibly random)
- ② Learn Q^π via TD updates (policy evaluation)
- ③ Construct π' from Q^π values (policy improvement)
- ④ Update policy $\pi \leftarrow \pi'$ and repeat until no changes

This is potentially **slow**, though: we have to wait until we've learned Q^π before we can improve the policy.

Generalized policy iteration refers to the general idea of combining policy evaluation and policy improvement within a single process.

In our particular case, we're going to try to learn an **optimal** policy from the get-go, by potentially **updating** our policy after each step.

Greedy Policy Improvement

A greedy policy always picks the action that looks best at each step:

$$\pi^{\text{greedy}}(s) \leftarrow \arg \max_{a \in A} Q(s, a) \quad (\text{ties broken at random})$$

Greedy Policy Improvement

```
procedure GREEDYPOLICYIMPROVEMENT(MDP  $M$ ; Discount factor  $\gamma$ ; Step size  $\alpha$ )
  for each  $(s, a) \in S \times A$  do
     $Q(s, a) \leftarrow 0$ 

  for each episode do
     $s \leftarrow s_0$                                  $\triangleright s$  is current state,  $s_0$  is fixed start state
     $a \leftarrow \pi^{\text{greedy}}(s)$                    $\triangleright a$  is next action to take, chosen greedily
    while episode has not ended do
      Execute action  $a$ 
      Observe next state  $s'$  and one-step reward  $r = R(s, a, s')$ 
      Determine  $a' \leftarrow \pi^{\text{greedy}}(s')$            $\triangleright a'$  is what we're going to do next
       $Q(s, a) \leftarrow Q(s, a) + \alpha [r + \gamma Q(s', a') - Q(s, a)]$ 
       $s \leftarrow s'$ ,  $a \leftarrow a'$                        $\triangleright$  Update current state and next action

  return learned policy  $\pi$ , with actions chosen greedily for each state
```

Exploration and Exploitation

Greedy isn't always good. E.g., consider an agent exploring a maze.

- Once the agent learns a sequence of actions that arrive at the goal, it will be strongly incentivized to use those same actions on subsequent episodes
- This may cause the agent to overlook or miss other action sequences that would get it to the goal faster

A greedy improvement process may end up with a **suboptimal** policy.

We need to balance:

- Exploitation:** Taking the best-looking actions based on what we currently know
- Exploration:** Trying out new actions even if they don't look good based on what we currently know

In the real world, one constantly has to decide between continuing in a comfortable existence, versus striking out into the unknown in the hopes of a better life.

Lecture 09-2b: ϵ -Greedy Policies and SARSA

Almost-Greedy Policies

The greedy policy focuses **almost exclusively** on **exploitation** (aside from breaking ties at random).

A relatively simple way to incorporate **exploration** is to adjust the policy to be **mostly greedy**, but **not always**.

An **epsilon-greedy** (ϵ -greedy) policy $\pi^{\epsilon\text{-greedy}}$, where ϵ is a small positive number, determines actions at any state s by:

- ① Generate a random number $R \in [0, 1]$
- ② If $R \leq \epsilon$, choose an action at random
- ③ If $R > \epsilon$, choose an action greedily via $\arg \max_{a \in A} Q(s, a)$

By using an ϵ -greedy policy in the policy improvement process, the **learned** action-value function can be made to converge to the **optimal** action-value function Q^* .

SARSA Learning

ϵ -Greedy Policy Improvement: SARSA

```
procedure SARSALEARNING(MDP  $M$ ;  $\gamma$ ;  $\alpha$ ;  $\epsilon$ )
    for each  $(s, a) \in S \times A$  do
         $Q(s, a) \leftarrow 0$                                  $\triangleright Q(s, a)$  is current estimate of  $Q^*(s, a)$ 
    for each episode do
         $s \leftarrow s_0$                                  $\triangleright s$  is current state,  $s_0$  is fixed start state
         $a \leftarrow \pi^{\epsilon\text{-greedy}}(s)$                  $\triangleright a$  is next action to take, chosen  $\epsilon$ -greedily
        while episode has not ended do
            Execute action  $a$ 
            Observe next state  $s'$  and one-step reward  $r = R(s, a, s')$ 
            Determine  $a' \leftarrow \pi^{\epsilon\text{-greedy}}(s')$            $\triangleright a'$  is what we're going to do next
             $Q(s, a) \leftarrow Q(s, a) + \alpha [r + \gamma Q(s', a') - Q(s, a)]$ 
             $s \leftarrow s'$ ,  $a \leftarrow a'$                              $\triangleright$  Update current state and next action
    return learned policy  $\pi$ , with actions chosen greedily for each state
```

- Called **SARSA** for s, a, r, s', a'
- We decrease ϵ over time to stabilize the policy and ensure convergence

Lecture 09-2c: Randomness and Weighting

Randomness and Weighting in Learning

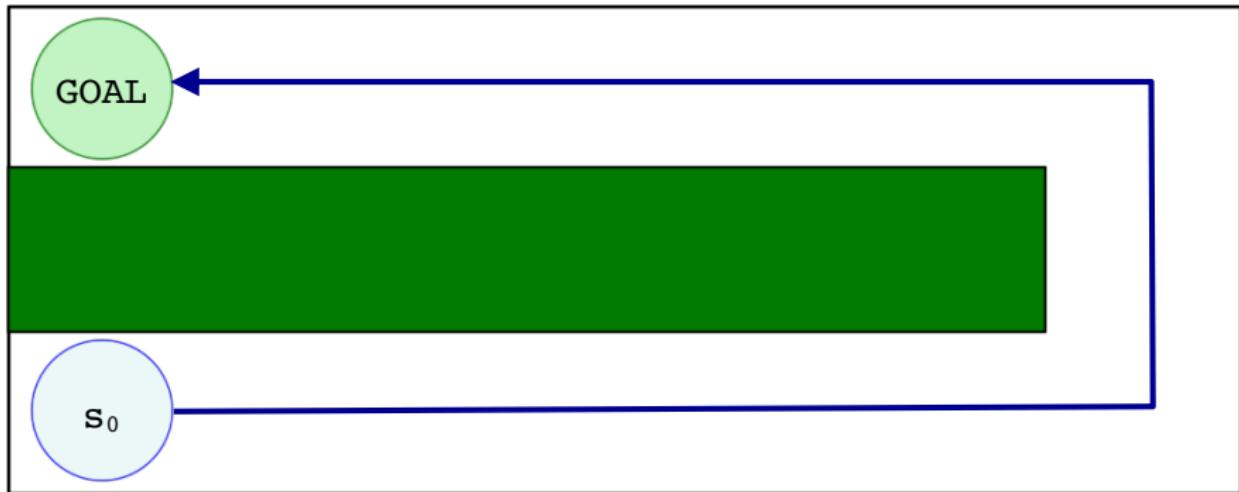
- ▶ Our algorithm uses two parameters, α and ε (plus the usual discount factor γ), to control its overall behavior
- ▶ Each can be adapted over time to control algorithm
 1. ε : the amount of randomness in the policy
 - ▶ When we don't know much, set it to a **high value**, so that we start off with lots of random exploration
 - ▶ We **reduce** this value over time until $\varepsilon = 0$, and we are being purely greedy, and just exploiting what we have learned
 2. α : the weight on each learning-update step
 - ▶ Reduce this over time, as well: when $\alpha = 0$, U -values don't change anymore, and we can converge on final policy values

Randomness and Weighting in Learning

- ▶ The control parameters α and ε give us simple ways to control complex learning behavior
- ▶ We ***don't always*** want to reduce each over time
- ▶ In a purely ***stationary environment***, where system dynamics don't ever change, and all probabilities stay the same, we can simply slowly reduce each until we converge upon a stable learned behavior
- ▶ In a ***non-stationary*** environment, where things may change at some point, learned solutions may quit working

Non-Stationary Environments

- ▶ Suppose environment starts off in one configuration:



- ▶ Over time, we can learn a policy for shortest path to goal
- ▶ By letting ϵ and α go to 0, the policy becomes stable

Non-Stationary Environments

- ▶ The environment may change, however:



- ▶ If ε and α **stay at 0**, policy is **sub-optimal** from now on

Non-Stationary Environments

- ▶ We may be able to tell that environment changes, however



- ▶ If value drops off over a long time, we can **increase** ϵ and α again, to resume learning and find new optimal policy

Lecture 09-2d: On-Policy vs. Off-Policy Updates and Q-Learning

SARSA Learning: On-Policy Updates

ϵ -Greedy Policy Improvement: SARSA

```
procedure SARSALEARNING(MDP  $M$ ;  $\gamma$ ;  $\alpha$ ;  $\epsilon$ )
    for each  $(s, a) \in S \times A$  do
         $Q(s, a) \leftarrow 0$                                  $\triangleright Q(s, a)$  is current estimate of  $Q^*(s, a)$ 
    for each episode do
         $s \leftarrow s_0$                                  $\triangleright s$  is current state,  $s_0$  is fixed start state
         $a \leftarrow \pi^{\epsilon\text{-greedy}}(s)$                  $\triangleright a$  is next action to take, chosen  $\epsilon$ -greedily
        while episode has not ended do
            Execute action  $a$ 
            Observe next state  $s'$  and one-step reward  $r = R(s, a, s')$ 
            Determine  $a' \leftarrow \pi^{\epsilon\text{-greedy}}(s')$            $\triangleright a'$  is what we're going to do next
             $Q(s, a) \leftarrow Q(s, a) + \alpha [r + \gamma Q(s', a') - Q(s, a)]$ 
             $s \leftarrow s'$ ,  $a \leftarrow a'$                              $\triangleright$  Update current state and next action
    return learned policy  $\pi$ , with actions chosen greedily for each state
```

SARSA is an **on-policy** update method.

- We pick our next action a according to our policy.
- To determine the **value** of the new state s' , we pick action a' **according to our policy also**.

The Impact of On-Policy Updates

An on-policy update of $Q(s, a)$ uses the action value $Q(s', a')$ where action a' is chosen according to the ϵ -greedy policy:

$$Q(s, a) \leftarrow Q(s, a) + \alpha [r + \gamma Q(s', a') - Q(s, a)]$$

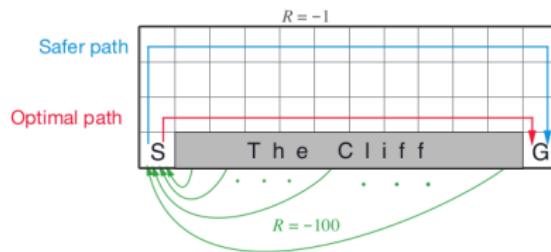
- With probability $(1 - \epsilon)$, a' **is the best possible action** in state s'
- With probability ϵ , a' is a **random** action

This means that the TD update of $Q(s, a)$ is based on a **potentially suboptimal** action a' in the next state:

$$Q(s', a') \leq \max_{a'' \in A} Q(s', a'').$$

On-Policy Updates: The Cliff Problem

Example: Consider an agent navigating to a goal, with two possible paths: a long safe path and a shorter path walking along a cliff.



- The ϵ -greedy policy will **sometimes** pick actions to jump off the cliff
 - How does the agent know **not** to jump off the cliff unless it tries it and sees what happens?
 - The choice of random actions ensures that the agent **explores** its environment, even if it does the wrong thing occasionally
- A consequence is that the **learned values** of states along the cliff will be **lower** than they otherwise would be if the agent was **acting optimally** (by never jumping)

Ensuring Optimal TD Updates

If we want $Q(s, a)$ to converge to the **optimal** action-value $Q^*(s, a)$, then the TD update should be based on the **optimal** action in state s' .

Two ways to correct this:

- ① Ensure that ϵ goes to 0 over time (so agent always acts optimally)
- ② Revise the TD update to use the **optimal** action in state s' (instead of the action that the agent **will take**, which may be **suboptimal**)

We can base the revised TD update on the Bellman **optimality** equation:

$$\begin{aligned} Q^*(s, a) &= \sum_{s'' \in S} P(s, a, s'') \cdot \left[R(s, a, s'') + \gamma \max_{a'' \in A} \{Q^*(s'', a'')\} \right] \\ &\approx r + \gamma \max_{a'' \in A} \{Q^*(s', a'')\} \quad \text{for observed state } s' \end{aligned}$$

Q-Learning: Off-Policy Updates

Q-Learning with ϵ -Greedy and Off-Policy Updates

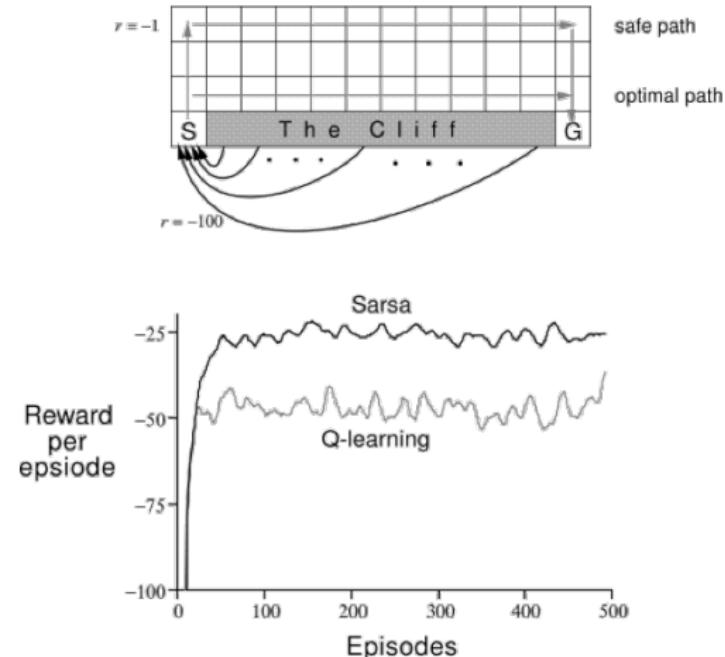
```
procedure QLEARNING(MDP  $M$ ;  $\gamma$ ;  $\alpha$ ;  $\epsilon$ )
    for each  $(s, a) \in S \times A$  do
         $Q(s, a) \leftarrow 0$                                  $\triangleright Q(s, a)$  is current estimate of  $Q^*(s, a)$ 
    for each episode do
         $s \leftarrow s_0$                                  $\triangleright s$  is current state,  $s_0$  is fixed start state
         $a \leftarrow \pi^{\epsilon\text{-greedy}}(s)$                  $\triangleright a$  is next action to take, chosen  $\epsilon$ -greedily
        while episode has not ended do
            Execute action  $a$ 
            Observe next state  $s'$  and one-step reward  $r = R(s, a, s')$ 
            Determine  $a' \leftarrow \pi^{\epsilon\text{-greedy}}(s')$            $\triangleright a'$  is what we're going to do next
             $Q(s, a) \leftarrow Q(s, a) + \alpha [r + \gamma \max_{a'' \in A} \{Q(s', a'')\} - Q(s, a)]$ 
             $s \leftarrow s'$ ,  $a \leftarrow a'$                              $\triangleright$  Update current state and next action
    return learned policy  $\pi$ , with actions chosen greedily for each state
```

Q-Learning is an **off-policy** update method.

- We still pick our actions a and a' according to our policy (e.g., ϵ -greedy)
- We update $Q(s, a)$ using the value of the **optimal** action in state s' , instead of the action a' that we actually take (which may be suboptimal)

Comparing the Methods: Cliff Problem

- ▶ Shortest path to the goal goes along edge of a cliff
- ▶ SARSA learns safer path, since edge-states get lower values due to random falls
- ▶ Q-Learning learns best path, since it **ignores** random jumps off edge
- ▶ Why does QL do worse in the end? How can we fix this over a period of time?



Example from: Sutton & Barto, 1998

Comparing SARSA and Q-Learning

In Q-Learning, the **learned** action-values $Q(s, a)$ **converge** to the **optimal** action-values $Q^*(s, a)$.

SARSA **can also** achieve this provided that ϵ decreases over time.

The primary difference is that Q-Learning often converges to Q^* **faster** than SARSA.

- Q-Learning doesn't have to spend time "fixing" values of states where it has **overestimated** the risk
- This allows for reducing ϵ more rapidly so that we can learn an optimal policy sooner
 - But reducing ϵ **too rapidly** can prevent necessary exploration

CS 457/557: Machine Learning

Lecture 09-3: Reinforcement Learning

Lecture 09-3a: Eligibility Traces

Extending the Value Update Procedure

- ▶ Basic reinforcement learning algorithms update the value of a **single state** (or single state-action pair) at a time
 - ▶ Repeated occurrences of the same state sequences eventually cause observed utility value to “spread out” over time, so that states that tend to lead to particularly good (or bad) states down the road also get higher (or lower) values
- ▶ Sometimes, we can speed up learning by **directly** implementing the process of spreading values over time

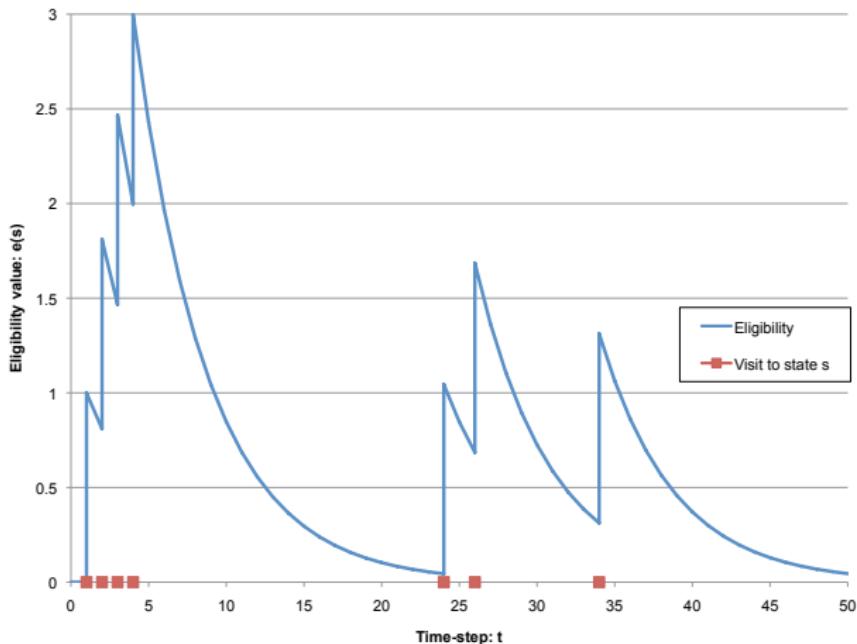
Eligibility Traces

- ▶ For each state, s , we set aside some extra memory, $e(s)$, to keep track of **how recently** we visited it
 - ▶ Each time we visit a state, we **increase** this value
 - ▶ We can choose a **lambda** value, $0 \leq \lambda \leq 1$, to control how quickly the “eligibility” variable $e(s)$ decreases over time
- ▶ At each time step t , we update the eligibility for all states, including the one we are currently visiting, s_t :

$$e_t(s) = \begin{cases} \gamma \lambda e_{t-1}(s) & \text{if } s \neq s_t \\ \gamma \lambda e_{t-1}(s) + 1 & \text{if } s = s_t \end{cases}$$

Eligibility Traces

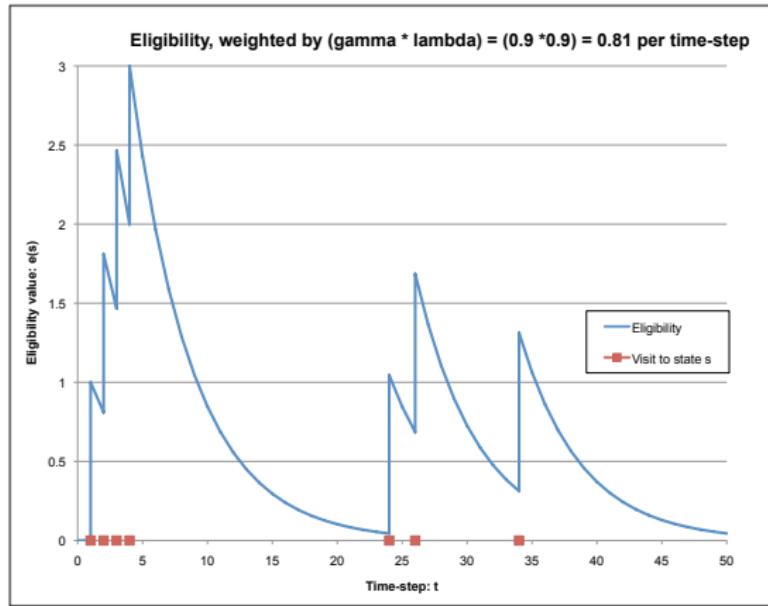
Eligibility, weighted by $(\gamma * \lambda) = (0.9 * 0.9) = 0.81$ per time-step



- ▶ The eligibility trace grows each time a state is visited, and then “decays” towards 0 over time, until that state is visited again

Eligibility Traces

- We then take the **TD error**—the difference between values of the current state and the last state, plus current reward—and **spread this out** over all the past states, *in proportion* to their eligibility values:



$$\delta_t = r_{t+1} + \gamma Q_t(s_{t+1}, a_{t+1}) - Q_t(s_t, a_t)$$

SARSA- λ : Learning with Eligibility Traces

function SARSA- λ (mdp) **returns a policy**

inputs: mdp , an MDP, and π , a policy to be evaluated

$\forall s \in S, \forall a \in A, Q(s, a) = 0$ **and** $e(s, a) = 0$

repeat for each episode E :

set start-state and action $s \leftarrow s_0, a \leftarrow \max_a Q(s_0, a)$

repeat for each time-step t of episode E , until s is terminal:

take action a

observe next state s' , one-step reward r

choose next action a' , chosen ϵ -greedily based on $Q(s', a')$

$\delta \leftarrow r + \gamma Q(s', a') - Q(s, a)$

$e(s, a) \leftarrow e(s, a) + 1$

$\forall s \in S, \forall a \in A :$

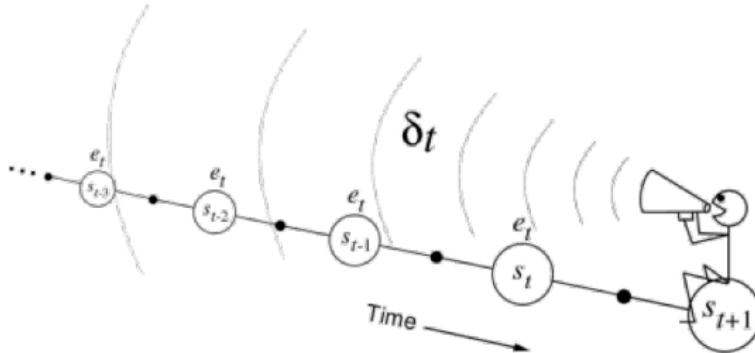
$Q(s, a) \leftarrow Q(s, a) + \alpha \delta e(s, a)$

$e(s, a) \leftarrow \gamma \lambda e(s, a)$

$s \leftarrow s', a \leftarrow a'$

return policy π , set greedily for every state and action based upon Q -values

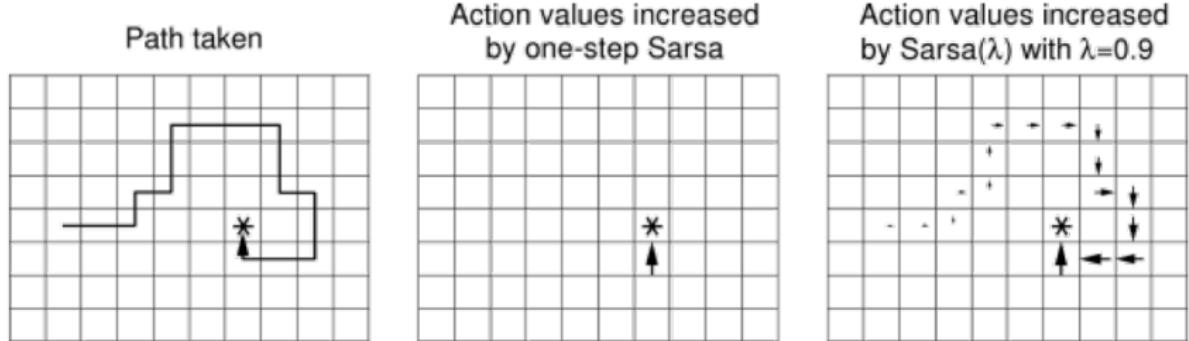
Using Eligibility Traces



- ▶ The algorithm sends the TD-error back over prior time-steps, with a “distance” that is affected by the choice of λ
- ▶ If we choose $\lambda = 0$, the algorithm is simply doing a **single-state backup** as before
- ▶ As λ nears (but never equals) 1, the algorithm pushes the updates further and further back in time

Diagram taken from: Sutton & Barto, 1998

Possible Advantages of Using Traces



- ▶ In many cases, using Eligibility Traces can speed the learning process, since whole chains of states can be updated at once
- ▶ This works especially well for path-planning problems, and things similar to it, since the value updates can affect multiple locations along the path to a goal at the same time

Example from: Sutton & Barto, 1998

Lecture 09-3b: Planning and Learning in RL

Full-Fledged Planning

<--	<--	<--	<--	
Λ	<--	<--	<--	<--
Λ	Λ	<--	<--	<--
Λ	<--	Λ	<--	<--
Λ	<--	<--	<--	<--

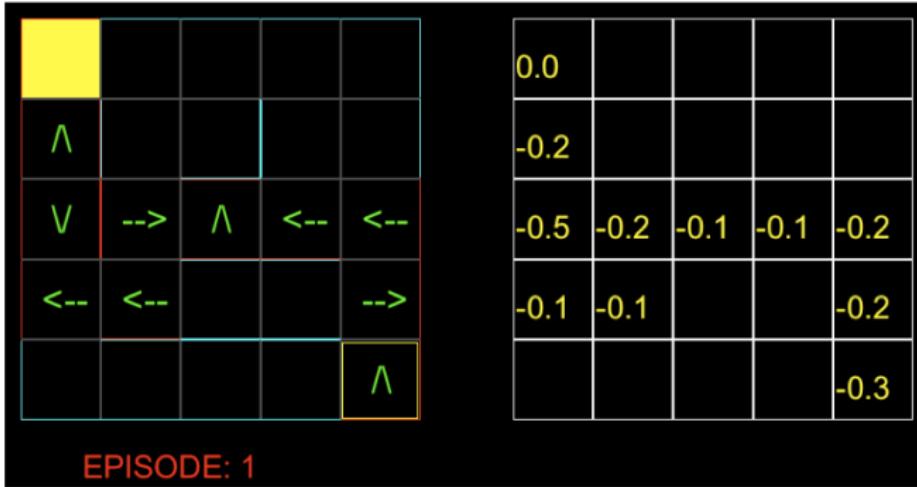
STABLE PI: TRUE

0.0	-1.0	-1.9	-2.71	-3.4
-1.0	-1.9	-2.71	-3.4	-4.0
-1.9	-2.71	-3.4	-4.0	-4.6
-2.71	-3.4	-4.0	-4.6	-5.2
-3.4	-4.0	-4.6	-5.2	-5.6

DELTA: 0.0

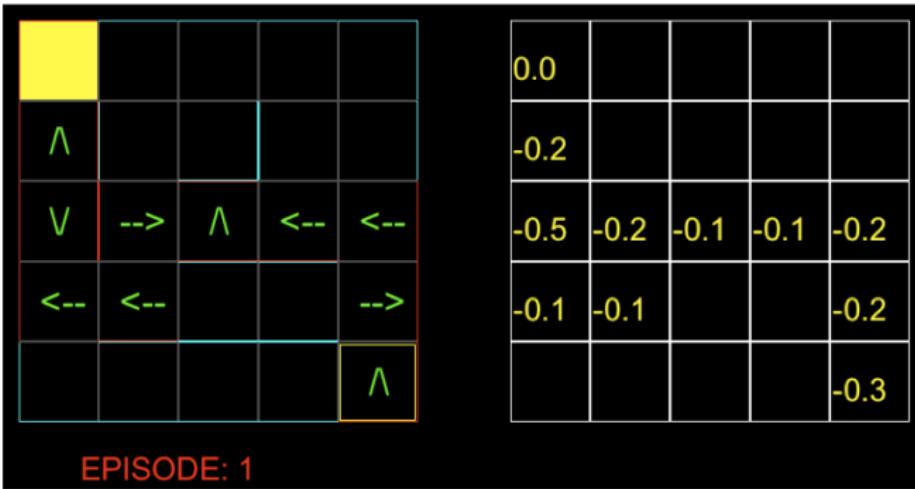
- ▶ Planning provides a full solution:
 - ▶ We have a model of the entire domain
 - ▶ We derive a solution for that whole model
- ▶ What are some potential benefits? Problems?

Learning our Model



- ▶ Learning allows us to **find** a correct model
 - ▶ Don't need to create model ahead of time
 - ▶ Can avoid incorrect models
- ▶ Is there **anything wrong** with the plan learned so far?

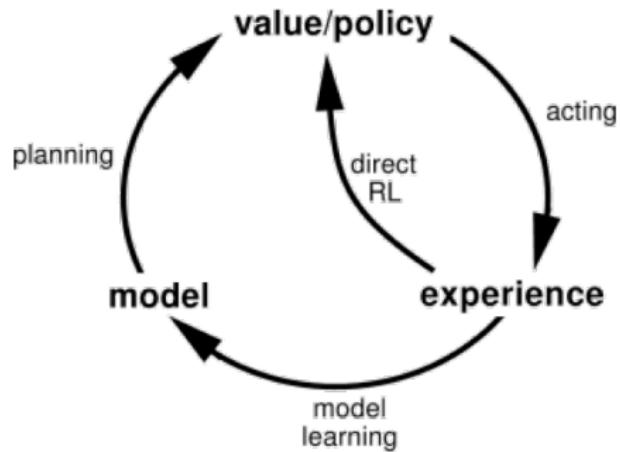
Sub-Optimal Learning



- ▶ Basic TD methods build model of the environment as they go
- ▶ Early on, model is *partial*, and action-policies are *sub-optimal*
- ▶ Even based on what we have learned about the environment so far, we are not doing the right thing!

Combining Planning & Learning

- ▶ What if we combine the two ideas together?
 - ▶ Use “direct RL” to **build** model
 - ▶ Use “indirect RL” to **test policies**, based on that model
 - ▶ Then do **planning** to get best current solution for the model
 - ▶ **Repeat** to get to optimal policy



The *Dyna-Q* Algorithm

function DYNA-Q (*mdp*) returns a policy

inputs: *mdp*, an MDP, and π , a policy to be evaluated

$\forall s \in S, \forall a \in A, Q(s, a) = 0$ and $Model(s, a) = (s, 0)$

 repeat forever :

$s \leftarrow$ current state

 take action a , chosen ϵ -greedily based on $Q(s, a)$

 observe next state s' , one-step reward r

$Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$

$Model(s, a) \leftarrow (s', r)$ (*assumes determinism*)

 repeat N times :

$s \leftarrow$ a randomly previously observed state

$a \leftarrow$ a random action previously taken in state s

$(s', r) \leftarrow Model(s, a)$

$Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$

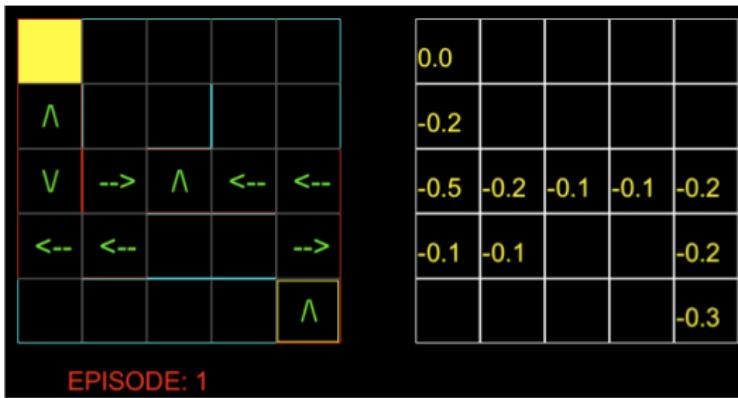
return policy π , set greedily for every (s, a) based upon $Q(s, a)$

▶ This method stores a separate set of results, the *Model*, based on what successor states and rewards have actually been seen while learning

▶ These are used to **simulate outcomes**, and build better value estimates for the states and actions we have actually seen

▶ What happens when $N = 0$?

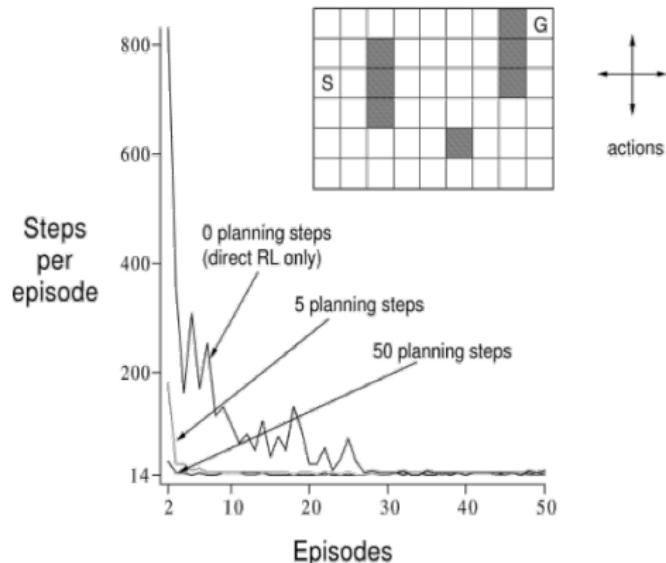
$N = 0$: Regular TD-Learning



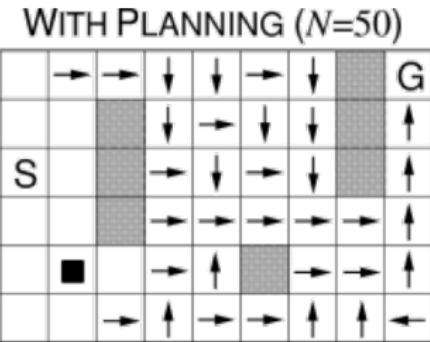
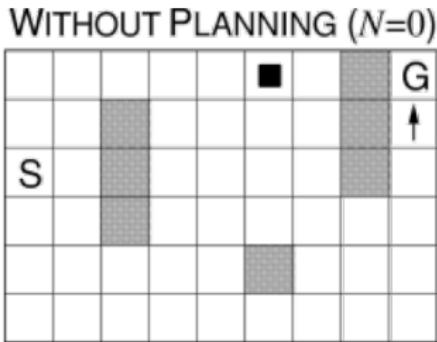
- After a single episode in the maze where we find the goal, **one-step policies** (the last arrow before goal) are already **correct**
- Remaining policy is not really very good in general
- Dyna-Q can fix this problem

Using Dyna-Q Methods

- In this Maze Problem, agents get 0 reward in all states, except the goal, where there is a positive reward (+1)
- We can see effects of different values of N
- As N increases, the agent learns better policies in fewer episodes
- Why does this happen?



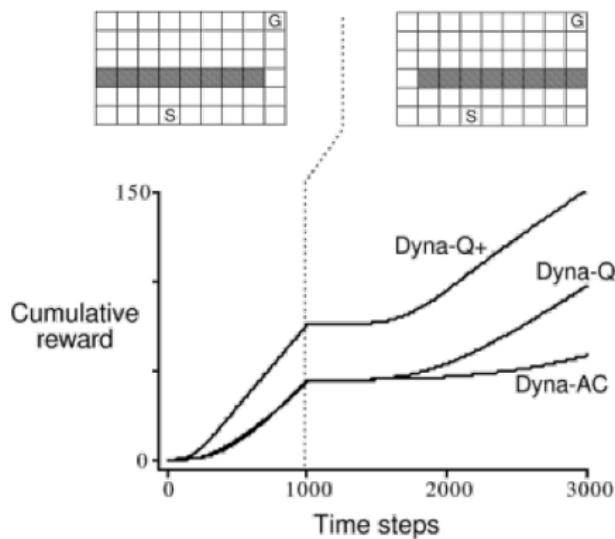
Advantages of Dyna-Q



- ▶ The policies learned after one episode
 - ▶ If all actions **have same value**, we don't show any arrow
 - ▶ Without planning ($N = 0$), only 1-step positive policy is really learned
 - ▶ With increased planning ($N = 50$), actions are chosen correctly for much more of the state space
 - ▶ Simulations keep updating values, finding paths from all **visited states**
 - ▶ Further **exploitation** of the learned model of the maze
 - ▶ Those states not visited yet still need to be learned through **exploration**

Model Recovery

- ▶ What happens if the world **changes** after learning?
- ▶ Dyna-Q adapts to new maze structures
- ▶ Here, we see that Dyna-Q+, which uses greater values of ϵ for its almost-greedy policy, figures things out more quickly, and gets **more value overall**



Lecture 09-3c: Generalization in RL

Extending the Value Update Procedure

```
function Q-LEARNING(mdp) returns a policy
    inputs: mdp, an MDP

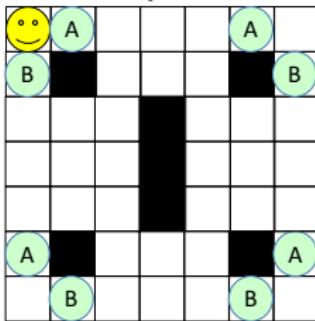
     $\forall s \in S, \forall a \in A, Q(s, a) = 0$ 
    repeat for each episode E:
        set start-state  $s \leftarrow s_0$ 
        repeat for each time-step t of episode E, until s is terminal:
            set action a, chosen  $\epsilon$ -greedily based on  $Q(s, a)$ 
            take action a
            observe next state  $s'$ , one-step reward r
            
$$Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$$

             $s \leftarrow s'$ 
    return policy  $\pi$ , set greedily for every state  $s \in S$ , based upon  $Q(s, a)$ 
```

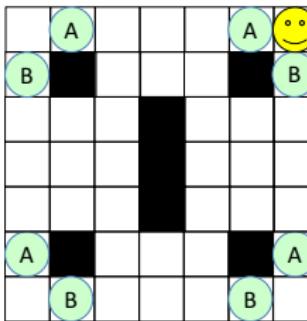
- ▶ Basic reinforcement learning algorithms update the value of a **single state** (or state-action pair) at a time

Lack of Generalization

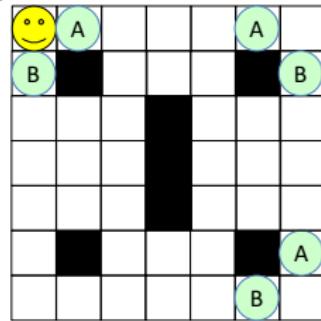
- When doing learning, each state is treated as unique, and must be repeated over and over to learn something about its value



Even if we learn that going right here is best, because type-A objects are better than type-B ones...



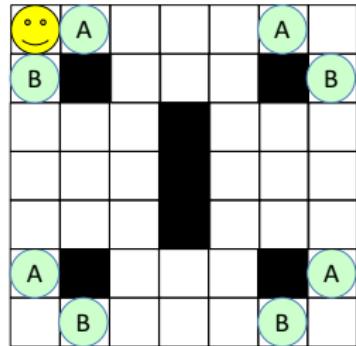
But this really tells us nothing about what to do in a ***similar state*** like this one...



Might even say nothing about what to do in a ***different environment*** with some states ***exactly the same!***

Feature Vectors

- ▶ Rather than use every single detail of a state space, we can try to generalize over multiple states at once, by selecting some **finite number of features** and learning based upon only those
- ▶ States (or state-action pairs) that share the same features are thus treated the same, even if they differ in other ways that we don't pay attention to
- ▶ Using the right features can speed learning significantly
- ▶ Here, we might try representing state-action pairs (s, a) in terms of just four features:



$$f_X(s, a) = \frac{x}{\max_x} \text{ for } x\text{-coordinate after } a \text{ in } s$$

$$f_Y(s, a) = \frac{y}{\max_y} \text{ for } y\text{-coordinate after } a \text{ in } s$$

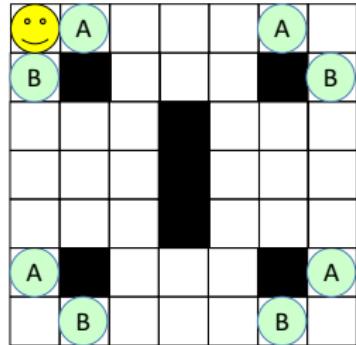
$$f_A(s, a) = \frac{1}{d_A + 1}, \text{ where } d_A \text{ is the distance to nearest } A \text{ after } a \text{ in } s$$

$$f_B(s, a) = \frac{1}{d_B + 1}, \text{ where } d_B \text{ is the distance to nearest } B \text{ after } a \text{ in } s$$

Choosing Feature Vectors

- ▶ We want to choose values that seem to be important to problem success
- ▶ When we represent them, it has been shown that we get better results if we **normalize** features, ensuring that each is in same, unit range:

$$0 \leq f_i(s, a) \leq 1$$



$$f_X(s, a) = \frac{x}{\max_x} \text{ for } x\text{-coordinate after } a \text{ in } s$$

$$f_Y(s, a) = \frac{y}{\max_y} \text{ for } y\text{-coordinate after } a \text{ in } s$$

$$f_A(s, a) = \frac{1}{d_A + 1}, \text{ where } d_A \text{ is the distance to nearest } A \text{ after } a \text{ in } s$$

$$f_B(s, a) = \frac{1}{d_B + 1}, \text{ where } d_B \text{ is the distance to nearest } B \text{ after } a \text{ in } s$$

Value Functions over Features

- ▶ One issue is that when we use simpler features, we don't always know which ones to use
 - ▶ States may share features and still have very different values
 - ▶ Some features may turn out to be more or less important
 - ▶ We want to **learn** a proper function that tells us how much we should pay attention to each feature
- ▶ We may assume this function is **linear**
 - ▶ This means that the value of a state is a simple combination of weights applied to each feature
 - ▶ While this assumption may not be the right one in some domains, it can often be the basis of a good approximation

Linear Functions over Features

- ▶ What we want is to learn the set of **weights** needed to properly calculate our U - or Q -values

$$U(s) = w_1 f_1(s) + w_2 f_2(s) + \cdots + w_n f_n(s)$$

$$Q(s, a) = w_1 f_1(s, a) + w_2 f_2(s, a) + \cdots + w_n f_n(s, a)$$

- ▶ For instance, for our grid problem:

$$Q(s, a) = w_X f_X(s, a) + w_Y f_Y(s, a) + w_A f_A(s, a) + w_B f_B(s, a)$$

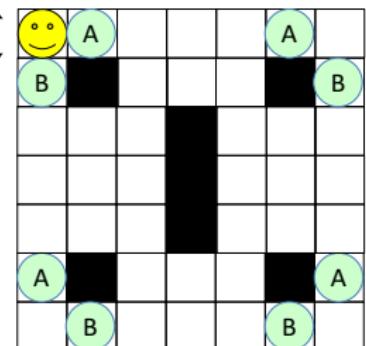
Setting the Weights

- ▶ Initially, we may not know which features really matter
 - ▶ In some cases, we may have knowledge that tells us some are more important than others, and we will weight them more
 - ▶ In other cases, we may treat them all the same
- ▶ For instance, in our grid problem, we might start with all weights the same (1.0)

$$\begin{aligned}Q(s, \text{RIGHT}) &= w_X f_X(s, a) + w_Y f_Y(s, a) + w_A f_A(s, a) + w_B f_B(s, a) \\&= (1.0 \times 2/7) + (1.0 \times 1/7) + (1.0 \times 1) + (1.0 \times 1/3)\end{aligned}$$

$$\begin{aligned}Q(s, \text{DOWN}) &= w_X f_X(s, a) + w_Y f_Y(s, a) + w_A f_A(s, a) + w_B f_B(s, a) \\&= (1.0 \times 1/7) + (1.0 \times 2/7) + (1.0 \times 1/3) + (1.0 \times 1) \\&= 1.76\end{aligned}$$

After Right, at $(x, y) = (2, 1)$,
distance 0 to A, distance 2 to B



Initially, many states may end up with identical value estimates. If it turned out that position didn't matter, and both A- and B-type objects were equally valuable, this would be fine. Typically, however, this is not the case, and we will need to adjust our weights dynamically as we go.

Q-Learning with Function Approximation

- In normal Q-learning, we evaluate a state-action pair (s, a) based on the results we get (r and s') and update the **single pair value**:

$$\delta = r + \gamma \max_{a'} Q(s', a') - Q(s, a)$$

$$Q(s, a) \leftarrow Q(s, a) + \alpha \delta$$

- Now, we will instead update **each of the weights** on our features
 - If outcomes are particularly good or bad, we change weights accordingly
 - This affects **all** (state, action) pairs that share features with current one

$$\delta = r + \gamma \max_{a'} Q(s', a') - Q(s, a)$$

$$\forall i, w_i \leftarrow w_i + \alpha \delta f_i(s, a)$$

Adjusting the Weights

- Now, when we take an action, we adjust weight-values and then compute Q -values
- For example: we take the RIGHT action and get a large positive reward, which means we could increase weights on contributing features

$$\delta = r + \gamma \max_{a'} Q(s', a') - Q(s, \text{Right}) = 10$$

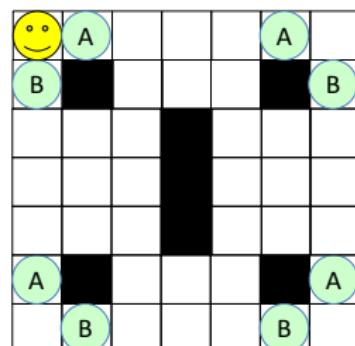
$$w_X \leftarrow w_X + \alpha \delta f_X(s, a)$$
$$\leftarrow 1.0 + 0.9 \times 10 \times 2/7 = 3.57$$

$$w_Y \leftarrow w_Y + \alpha \delta f_Y(s, a)$$
$$\leftarrow 1.0 + 0.9 \times 10 \times 1/7 = 2.29$$

$$w_A \leftarrow w_A + \alpha \delta f_A(s, a)$$
$$\leftarrow 1.0 + 0.9 \times 10 \times 1 = 10.0$$

$$w_B \leftarrow w_B + \alpha \delta f_B(s, a)$$
$$\leftarrow 1.0 + 0.9 \times 10 \times 1/3 = 4.0$$

$$Q(s, \text{RIGHT}) = (3.57 \times 2/7) + (2.29 \times 1/7) + (10.0 \times 1) + (4.0 \times 1/3)$$
$$= 12.69$$



Adjusting the Weights

- Later, if we take the DOWN action from the same state, and get a large negative cost-value, we might down-grade weights on the contributing features

$$\delta = r + \gamma \max_{a'} Q(s', a') - Q(s, \text{DOWN}) = -20$$

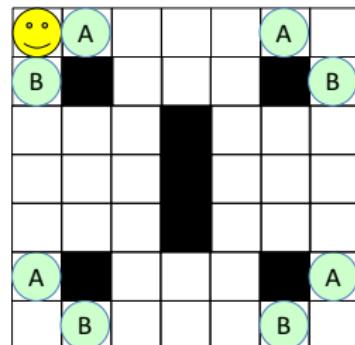
$$w_X \leftarrow w_X + \alpha \delta f_X(s, a) \\ \leftarrow 3.57 + 0.9 \times -20 \times 1/7 = 1.0$$

$$w_Y \leftarrow w_Y + \alpha \delta f_Y(s, a) \\ \leftarrow 2.29 + 0.9 \times -20 \times 2/7 = -2.85$$

$$w_A \leftarrow w_A + \alpha \delta f_A(s, a) \\ \leftarrow 10.0 + 0.9 \times -20 \times 1/3 = 4.0$$

$$w_B \leftarrow w_B + \alpha \delta f_B(s, a) \\ \leftarrow 4.0 + 0.9 \times -20 \times 1 = -14.0$$

$$Q(s, \text{DOWN}) = (1.0 \times 1/7) + (-2.85 \times 2/7) + (4.0 \times 1/3) + (-14.0 \times 1) \\ = -13.34$$



Adjusting the Weights

- ▶ Note: since we adjust the weights that are used to calculate the Q -values of any (state, action) pairs, what happens when we encounter one new outcome actually affects the Q -value of ***all*** the pairs at once
- ▶ We are thus potentially learning a value function over our ***entire space***, even though it is based only on a single outcome at a time, which can speed up learning

$$\delta = r + \gamma \max_{a'} Q(s', a') - Q(s, \text{DOWN}) = -20$$

$$\begin{aligned}Q(s, \text{DOWN}) &= (1.0 \times 1/7) + (-2.85 \times 2/7) + (4.0 \times 1/3) + (-14.0 \times 1) \\&= -13.34\end{aligned}$$

$$\begin{aligned}Q(s, \text{RIGHT}) &= (1.0 \times 2/7) + (-2.85 \times 1/7) + (4.0 \times 1) + (-14.0 \times 1/3) \\&= -0.79\end{aligned}$$

After Down, we have changed weights, which changes the Q -value of not only one state-action pair, but ***all of them***.

Here, we see the updated value for going Right (this was 12.69 before).

CS 457/557: Machine Learning

Lecture 09-4: Reinforcement Learning

Lecture 09-4a: Finding Features

Linear Functions over Features

- ▶ Suppose we have a set of **features** that describe states, or state-action pairs, of an MDP
- ▶ What we want is to learn the set of **weights** needed to properly calculate our U - or Q -values

$$U(s) = w_1 f_1(s) + w_2 f_2(s) + \cdots + w_n f_n(s)$$

$$Q(s, a) = w_1 f_1(s, a) + w_2 f_2(s, a) + \cdots + w_n f_n(s, a)$$

- ▶ We learn by adjusting weights, e.g., in a Q -learning way:

$$\delta = r + \gamma \max_{a'} Q(s', a') - Q(s, a)$$

$$\forall i, w_i \leftarrow w_i + \alpha \delta f_i(s, a)$$

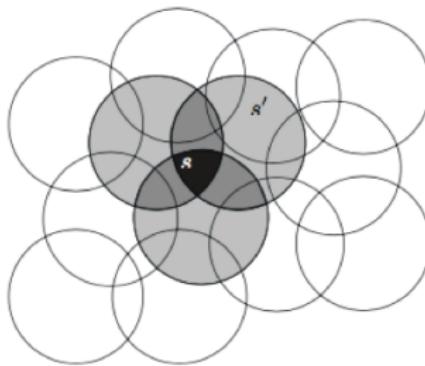
Where Do Features Come From?

- ▶ A variety of approaches can be used to generate useful features for a given learning problem
 - ▶ Sometimes we have good intuitions about what features are useful, and sometimes we don't
- ▶ For example, suppose we have a problem in which states are characterized by (x, y) points in the plane:
 1. We might just use the features (x, y) themselves, with 2 matching weights...
 2. Or a simple polynomial combination, like $(1, x, y, xy)$, with 4 weights to go along with those...
 3. Or a more complex polynomial, like:
$$(1, x, y, xy, x^2, y^2, xy^2, x^2y, x^2y^2)$$

Various Feature Functions

- ▶ Experimentation has been performed with many different functional forms for features:
 1. Polynomial features: linear weights over polynomial combinations of numeric features
 2. Fourier features: combinations of sine and cosine functions over the underlying numbers
 3. Radial basis functions: real-valued features based on computed distances from chosen points in the state-space
- ▶ A common issue is the complexity growth of the features as the dimensionality of the state space increases
 - ▶ A variety of techniques exist to use **sparser, binary** features

Coarse Coding



- ▶ Suppose we have a state-space consisting of points in the plane
- ▶ A binary coding scheme is to use features that each correspond to **circles** in the state-space
 - ▶ A point has a given feature if it lies inside the circle
 - ▶ Two points share all the features that overlap them both

Image: Sutton & Barto, 2018

Advantages of Binary Features

- ▶ Given a coarse coding scheme (set of n circles), we get a feature-vector, and corresponding weight-vector for state s :

$$\mathbf{x}_s = (c_1, c_2, \dots, c_n)$$

$$\mathbf{w}_s = (w_1, w_2, \dots, w_n)$$

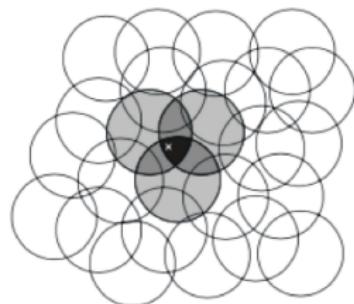
- ▶ Each feature is a binary value:

$$c_i = \begin{cases} 1 & \text{if } s \text{ lies inside circle } i \\ 0 & \text{else} \end{cases}$$

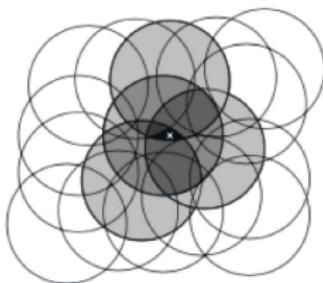
- ▶ If we use these features to compute a utility-value for s , rather than a series of multiplications and additions, we get the simpler summation:

$$U(s) = \mathbf{w}_s \cdot \mathbf{x}_s = \sum_i \{w_i \mid c_i = 1\}$$

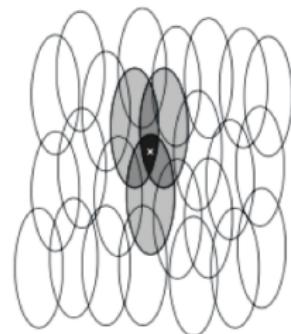
Feature Variation



Narrow generalization



Broad generalization

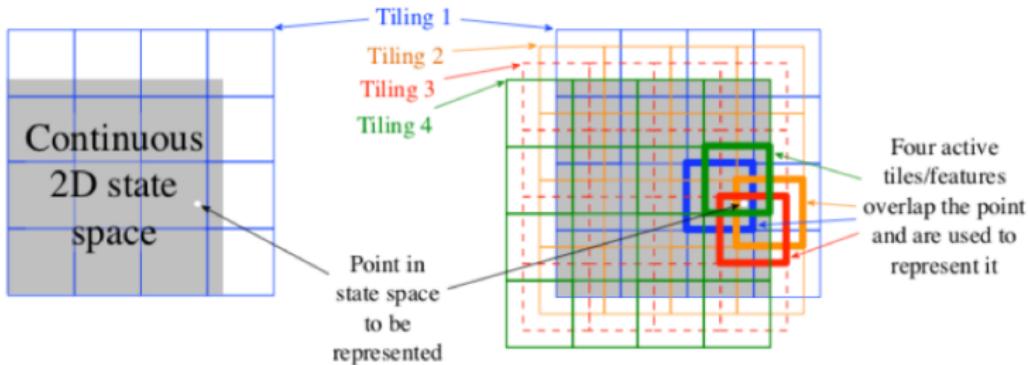


Asymmetric generalization

- ▶ A simple idea, circular coarse coding can still generalize over domains in a variety of ways
- ▶ We get generalization, since the **same** weight vector is used for every state (only the **particular features** change)

Image: Sutton & Barto, 2018

Tile Coding



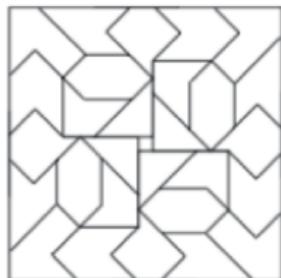
- ▶ Another form of coarse coding is to **tile** the state space
 - ▶ A single, simple tiling is just a **partition**, dividing the state space into uniform regions
 - ▶ A more complex tiling uses ***multiple overlapping*** partitions
 - ▶ Again, each individual tile corresponds to a binary feature

Image: Sutton & Barto, 2018

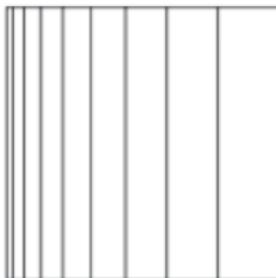
Placement of Tiles

- ▶ Much research has gone into good ways to choose how many tilings to use, and how they should overlap
 - ▶ It has been found that choosing ***uniform offsets*** of the tiles (e.g. moving each new tiling over by 1 unit in each direction) can introduce certain numerical biases
- ▶ Best practices, as recommended by Miller and Glanz (96):
 1. For a state space of dimensionality k , use $2^n \geq 4k$ tilings
 2. Offset each dimension using numbers $(1, 3, 5, \dots, 2k-1)$
- ▶ For example, in 2 dimensions, use 8 or more tilings, offsetting each by 1 unit in the x dimension, and 3 in y
- ▶ In 3 dimensions, use at least 12 tilings, offsetting by 1 unit in x , 3 in y , 5 in z ...

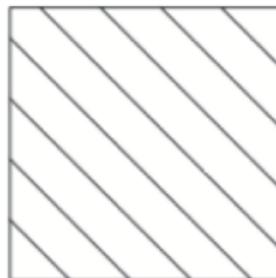
Other Approaches to Tiling



Irregular



Log stripes



Diagonal stripes

- ▶ Any number of tiling patterns and offsets can be used
 - ▶ Tilings need not be regular in shape or size
 - ▶ “Striped” tilings generalize along only certain dimensions
 - ▶ Different sizes of tiles allow finer/coarser discrimination in certain parts of the state space

Image: Sutton & Barto, 2018

Lecture 09-4b: Kanerva Coding

Sparse Distributed Memory

- ▶ Developed by NASA Ames researcher Pentti Kanerva while working on a model of human long-term memory
- ▶ In RL, often now called **Kanerva coding**
- ▶ A model that generalizes over states based on a **similarity** measure
- ▶ State-features are represented again as binary vectors, which can be regarded as lists of **other** states to which they are or are not similar



Basic Kanerva Coding for Q -Learning

- ▶ Rather than save Q -values for all state-action pairs, a Kanerva coding selects a subset of **prototypes**:

$$P = \{p_1, p_2, \dots, p_n\} \subsetneq S$$

- ▶ For any state s and prototype p_i , we say the two are **adjacent** if s and p_i differ by at most 1 feature (other such similarity measures are possible)
- ▶ For example, if we have two state feature-variables:

$$f_1 \in \{1, 2\} \quad f_2 \in \{a, b, c\}$$

- ▶ Then the state $s = (1, a)$ would be:
 1. Adjacent to prototypes $p_i = (2, a), (1, b)$, or $(1, c)$
 2. **Not** adjacent to $p_j = (2, b)$ or $(2, c)$

From Prototypes to Binary Features

- ▶ For our vector of prototypes we then get a vector of binary features for every state:

$$P = \{p_1, p_2, \dots, p_n\}$$

$$\mathbf{x}_s = (f(s)_1, f(s)_2, \dots, f(s)_n)$$

$$f(s)_i = \begin{cases} 1 & \text{if } s \text{ is adjacent to } p_i \\ 0 & \text{else} \end{cases}$$

- ▶ Our Q -learning algorithm then learns weight values over prototypes only:

$$\theta(p_i, a), \forall p_i \in P, \forall a \in A$$

Adjusting Prototype Weights

- ▶ For any state-action pair, we can now compute an approximate **Q -value**, based only on adjacent prototypes:

$$\hat{Q}(s, a) = \sum_i \theta(p_i, a) f(s)_i$$

- ▶ Furthermore, when our learning algorithm takes action a in state s_1 , receives reward r , and ends up in state s_2 we update:

$$\theta(p_i, a) \leftarrow \theta(p_i, a) + f(s)_i \alpha(r + \gamma \max_{a_2} \hat{Q}(s_2, a_2))$$

- ▶ Doing it this way also means that we only update those weights on pairs featuring prototypes that are adjacent to s , since otherwise $f(s)_i = 0$

Lecture 09-4c: Adaptive Kanerva Coding

Choosing Prototypes

- ▶ In the limit, we could use ***all*** states as prototypes ($P = S$), and impose strict identity on the measure of adjacency:

$$f(s)_i = \begin{cases} 1 & \text{if } s = p_i \\ 0 & \text{else} \end{cases}$$

- ▶ In this case, the algorithm is just normal Q-learning, without any generalization at all
- ▶ If we make the set of prototypes very small, on the other hand, then most states will not be adjacent to any prototype, and we won't learn ***anything*** about those states at all

Choosing Prototypes

- ▶ In between the extremes, the usefulness of Kanerva encoding is maximized when:
 1. No prototype is visited too much (such a state is effectively **too abstract**)
 2. No prototype is visited too little (such a state is effectively **too specific**)
- ▶ Achieving this balance with an initial set of prototypes, which is often chosen randomly, or according to some heuristic, is challenging, leading to interest in **adaptive Kanerva Coding**, where we **change** the prototype set over time as we learn

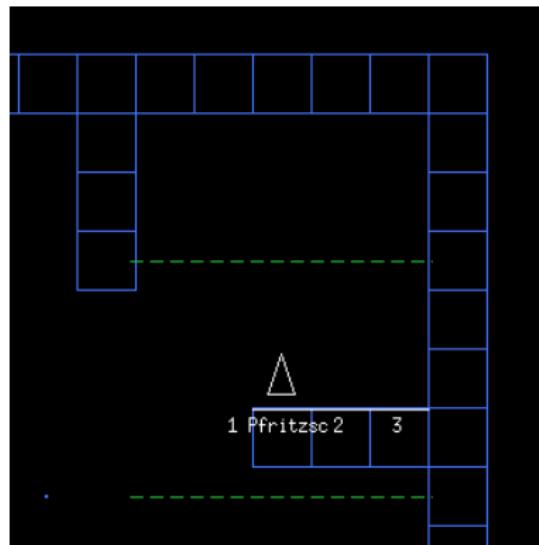
Adaptive Kanerva Coding

- ▶ We can modify the basic approach as follows:
 1. We start with a set of randomly chosen prototypes
 2. For each prototype, we keep track of how many times we visit a state that is adjacent to it
 3. Periodically, we modify the prototype set in two ways:
 - a. **Deletion:** if a prototype has been visited t times, we decide whether or not to delete it randomly, with probability:
$$P_{del} = e^{-t}$$
 - b. **Splitting:** whenever some prototypes are deleted, they are replaced by taking the **most-visited** prototypes and generating new, adjacent ones by changing one of their feature-values

Lecture 09-4d: Kanerva Coding Application

An Application: Xpilot Navigation

- ▶ Xpilot is a space-shooter game with the ability to add complex physics and environments
- ▶ Basic Q -learning is thwarted by the enormous state-space of the full game (even when only trying to learn to navigate successfully without crashing)
- ▶ Even on a small map of size (500 x 500), a ship with 10 possible speeds and full rotation will correspond to approximately 3.24×10^{10} states!



An Application: Xpilot Navigation

TABLE I

STATE VARIABLES FOR XPILOT, WITH RANGES. THE STATE SPACE IS MADE UP OF 5 VARIABLES AND 31104 STATES. THESE COMBINE WITH 4 ACTIONS FOR 124416 STATE-ACTION PAIRS (s, a) . 1024 STATES ARE CHOSEN AS PROTOTYPES, FOR 4096 PROTOTYPE-ACTION PAIRS (p, a) .

State Variable	Range
Heading	1–36
Tracking	1–36
Speed	1–6
Near Wall	{1, 0}
Near Corner	{1, 0}

TABLE II
POSSIBLE AGENT ACTIONS

Action	Effect
Avoid Wall	Turn 10° : away from nearest wall; thrust once
Avoid Corner	Turn 10° : direction 180° opposite Tracking ; thrust once
Thrust	If speed $s < 6$, thrust once
Do Nothing	null

TABLE III
THE REWARD-STRUCTURE.

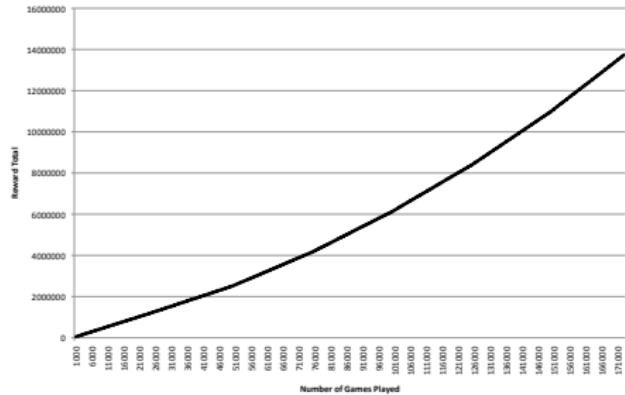
Reward Value	Condition
-10	Agent crashes
+1	Agent is alive for one frame

Even with a simplified and discretized state-action space, things are still far too complex for effective use of basic RL algorithms

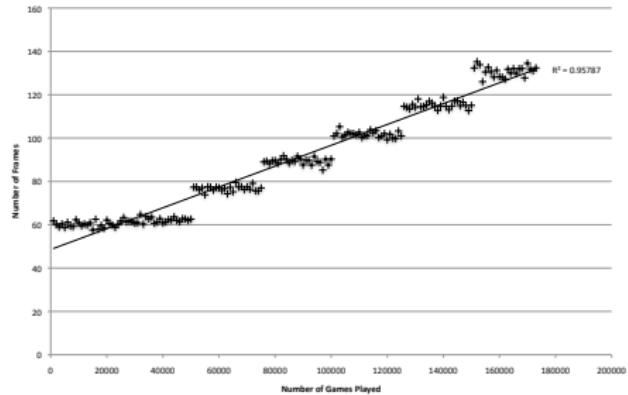
Other Parameters

1. # of prototypes: 1,024 initial prototypes (at random)
 - ▶ A total of 4,096 (prototype, action) pairs
 - ▶ ~3.3% of overall possible (state, action) pairs
2. Learning updates: $\gamma = 0.9$, $\alpha = 0.1$
3. Policy randomness: initially $\varepsilon = 0.9$
 - ▶ Every 25,000 games we update: $\varepsilon = \frac{0.9}{\lfloor \text{totalGames}/25,000 \rfloor + 1}$
4. Updating prototypes: every time any prototype is visited (by encountering an adjacent state) 50 times, we:
 - a. Delete m of them using randomness based upon number of visits
 - b. Split each of the most-visited m prototypes, returning to 1,024
 - c. Re-set all counts of how many times each is visited

Learning Performance



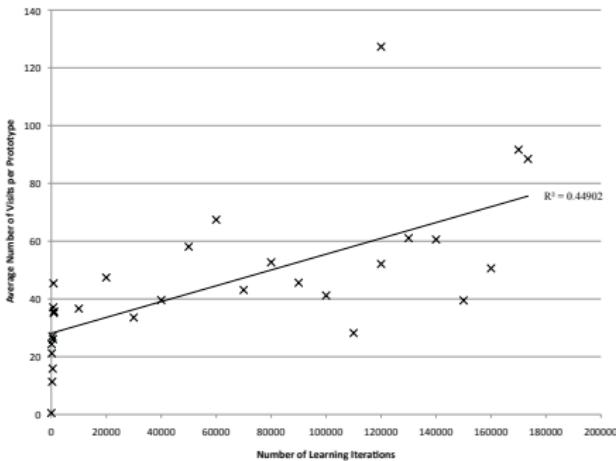
(a) Total Reward Over Time



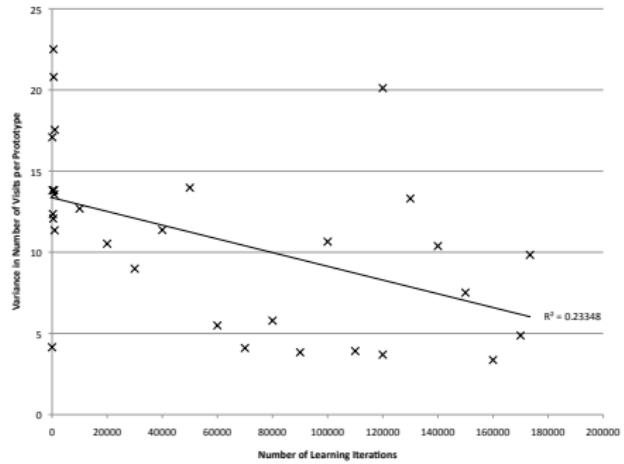
(b) Average Frames Alive Over Time

- ▶ The agent continues to improve performance over many hours of learning, comprising over 170,000 episodes
 - ▶ Note: as they get better, each episode of learning gets longer as they survive more frames

Prototype Visits



(a) Average Visits per Prototype Over Time



(b) Variance: Average Visits per Prototype Over Time

- ▶ As rarely visited prototypes are replaced by more useful ones, overall rate of visits increases
 - ▶ At same time, variance of visits to any prototype decreases

Adding More Prototypes

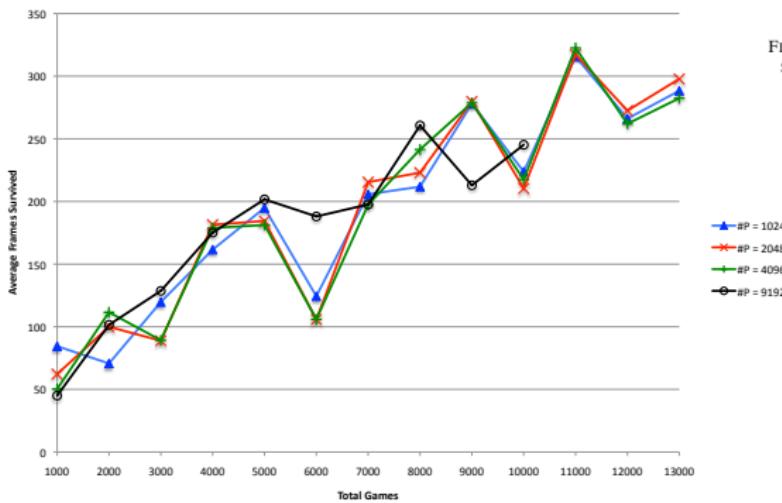


TABLE IV
NUMBERS OF PROTOTYPES USED IN THE EXPERIMENTS DESCRIBED IN FIGURE 4. EACH COMBINES WITH 4 ACTIONS FOR THE GIVEN NUMBER OF STATE-ACTION PAIRS (s, a). ALSO SHOWN IS THE PERCENTAGE OF THE ENTIRE STATE-ACTION SPACE (124416 PAIRS) REPRESENTED.

Prototypes	(s, a)	% Total
1024	4096	3.3%
2048	8192	6.6%
4096	16384	13.2%
9192	36768	29.6%

- ▶ Using 2/4/8 times as many prototypes has little to no effect, meaning that the smaller number is as good as any other
- ▶ For the largest size of prototype set, learning was much slower, and experiments had to be curtailed somewhat

CS 457/557: Machine Learning

Lecture 09-5: Deep Reinforcement Learning

Lecture 09-5a: Deep Reinforcement Learning

Review: Q-Learning with Function Approximation

- ▶ Q-learning: we evaluate a state-action pair (s, a) based on the results we get (r and s') and update the **single pair value**:

$$\delta = r + \gamma \max_{a'} Q(s', a') - Q(s, a)$$

$$Q(s, a) \leftarrow Q(s, a) + \alpha \delta$$

- ▶ The Q-values we learn can simply be stored in a data-structure of our choice, for look-up whenever we need them

Review: Q-Learning with Function Approximation

- ▶ If we decide to do function approximation instead, we can assume that the value of any given state-action pair is a linear function of certain features of the state, just as when we looked at various different classification algorithms:

$$Q(s, a) = w_1 f_1(s, a) + w_2 f_2(s, a) + \cdots + w_n f_n(s, a)$$

- ▶ Learning updates these weights, instead of Q-values directly

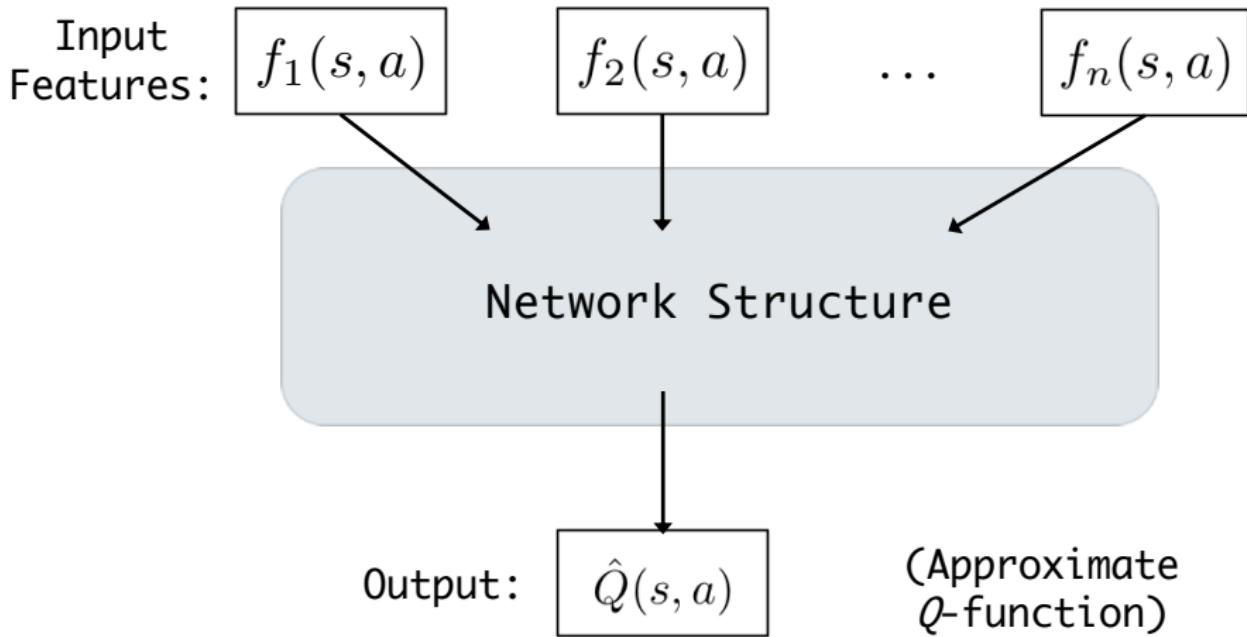
$$\delta = r + \gamma \max_{a'} Q(s', a') - Q(s, a)$$

$$\forall i, w_i \leftarrow w_i + \alpha \delta f_i(s, a)$$

- ▶ Why limit ourselves to **linear** functions of features?

Combining Neural Nets and Reinforcement Learning

- ▶ Basic idea: use a NN as our function approximator for Q



Problems with the Basic Idea

- ▶ The use of NNs (and other non-linear function approximators) in RL has been considered at length
- ▶ It faces some challenges, including:
 1. The **temporal** nature of an MDP process: since actions and states proceed in a (cause/effect) sequence, there are significant correlations between different inputs seen, which is a problem for reliable NN training
 2. The relationship between **values and policies**: since changes in Q -value affect what actions are chosen, the input distribution can change every time we update
- ▶ As a result, Q -learning may no longer converge to stable values over time, and may in fact be impossible
 - ▶ For more information see: Tsitsiklis & Roy, "An analysis of temporal-difference learning with function approximation," *IEEE Transactions on Automatic Control* 42 (1997).

Lecture 09-5b: Deep Q-Networks

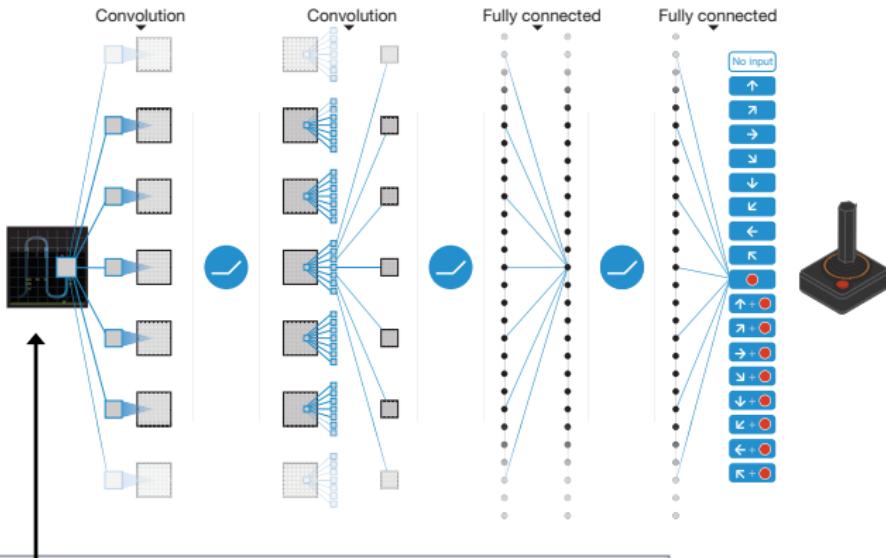
Deep Q-Networks (DQN)



- ▶ Mnih, et al., “Human-level control through deep reinforcement learning,” *Nature* 518 (2015)
- ▶ A recent breakthrough in combining neural nets and reinforcement learning, achieving strong results for a test-bed of Atari 2600 games (played via emulator)

A CNN for Atari Control

Mnih, et al. (2015)

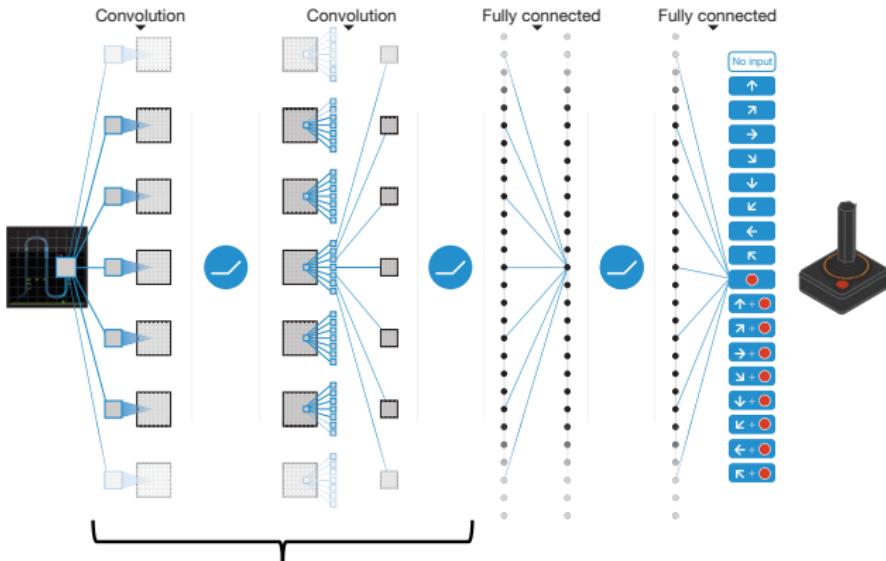


Inputs: $(84 \times 84 \times 4)$

4 frames of graphics from a game,
scaled down to (84×84) pixels

A CNN for Atari Control

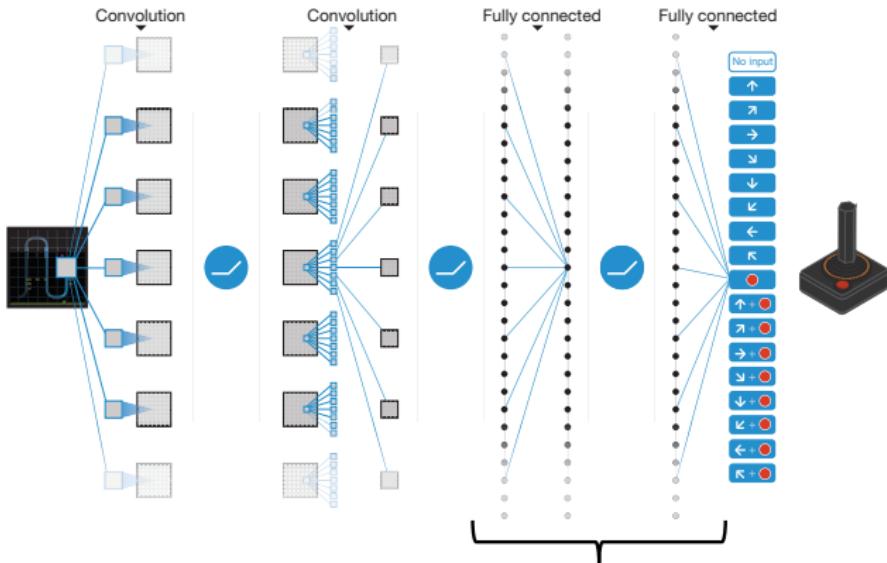
Mnih, et al. (2015)



3 Convolution Layers, with RELU in between each:
[1] $(32 \times 8 \times 8)$, stride = 4
[2] $(64 \times 4 \times 4)$, stride = 2
[3] $(64 \times 3 \times 3)$, stride = 1

A CNN for Atari Control

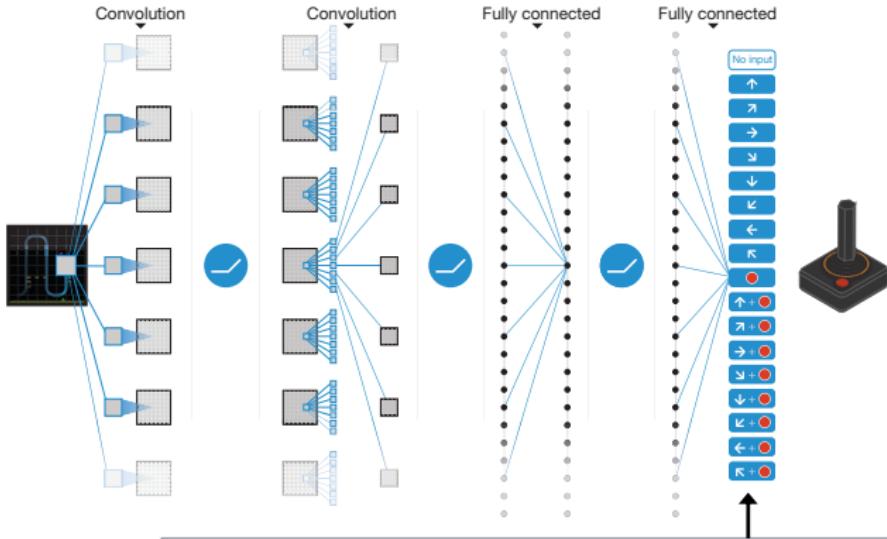
Mnih, et al. (2015)



2 Fully-Connected Layers:
512 RELU layer, followed by
a linear output layer

A CNN for Atari Control

Mnih, et al. (2015)



Outputs: one output node per action
(rather than single Q-value output),
varying from 4 to 18 nodes,
depending on the game

Lecture 09-5c: DQN Features

Deep Q -Learning with Replay

Let D be a *replay memory*, with capacity N

Initialize two neural networks with identical random weights, $\Theta_1 = \Theta_2$

for episode $e = 1 \dots E$ **do**

 Set state $s_t = s_0$, the start state

for time-step $t = 1 \dots T$ **do**

 Choose action a_t using an ϵ -greedy policy, based on Θ_1

 Receive reward r_t and next state s_{t+1}

 Store transition (s_t, a_t, r_t, s_{t+1}) in memory D

for simulation $m = 1 \dots M$ **do**

 Sample a random transition $(s_i, a_i, r_i, s_{i+1}) \in D$

 Set $y_i = \begin{cases} r_i & \text{if } s_{i+1} \text{ is end of game} \\ r_i + \gamma \max_{a'} \Theta_2(s_{i+1}, a') & \text{otherwise} \end{cases}$

 Do gradient descent, using error $(y_i - \Theta_1(s_i, a_i))^2$

endfor

 Every C steps, set $\Theta_2 = \Theta_1$

endfor

endfor

Features of the Algorithm

- ▶ The deep-Q algorithm uses a **replay memory**, which stores state-action transitions encountered during play
 - ▶ Periodically, the algorithm samples random past transitions from this memory and does weight updates based upon those
- ▶ This avoids the first problem for methods that combine neural nets and RL (temporal association): since episodes are sampled at random, correlations between states of game-play no longer bias the learning
 - ▶ Mnih, et al. use a memory of 1,000,000 past transitions
 - ▶ At the start of the algorithm, 50,000 purely random actions are chosen to initially populate the memory

Features of the Algorithm

- The algorithm uses **two** neural nets, one to generate sample outputs, and one updated by back-propagation

```
for simulation  $m = 1 \dots M$  do
```

```
    Sample a random transition  $(s_i, a_i, r_i, s_{i+1}) \in D$ 
```

```
    Set  $y_i = \begin{cases} r_i & \text{if } s_{i+1} \text{ is end of game} \\ r_i + \gamma \max_{a'} \Theta_2(s_{i+1}, a') & \text{otherwise} \end{cases}$ 
```

```
    Do gradient descent, using error  $(y_i - \Theta_1(s_i, a_i))^2$ 
```

```
endfor
```

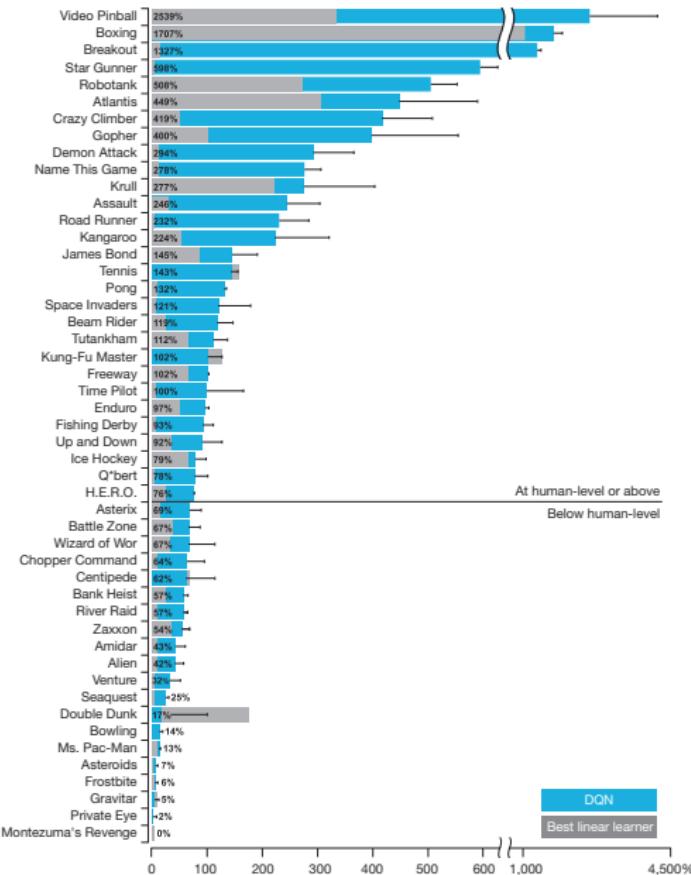
```
Every  $C$  steps, set  $\Theta_2 = \Theta_1$ 
```

Output of Θ_2 used

Net Θ_1 updated

- This avoids second problem (value/policy dependence):
updates to weights don't change output right away
 - Mini-batches of 32 simulated transitions are sampled
 - The sampling network is updated every $C = 10,000$ steps

Lecture 09-5d: DQN Results



Learned Performance

The same algorithm was run on all games in the Atari test-bed (only changing number of available actions).

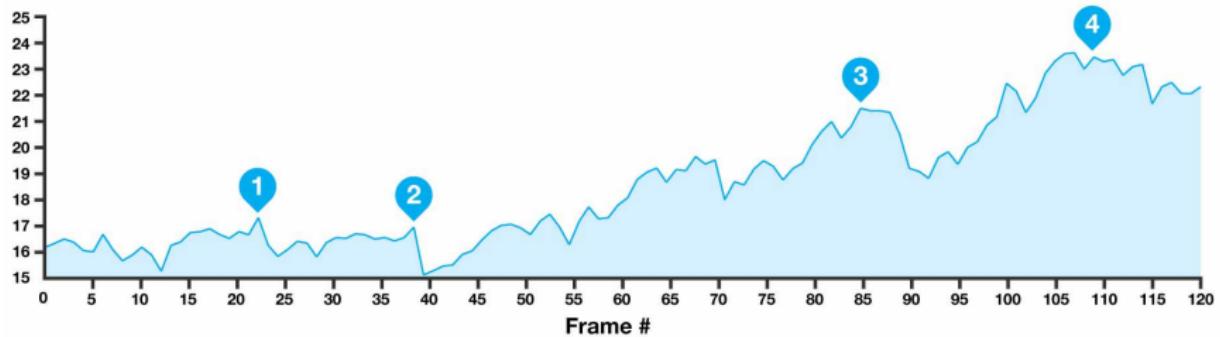
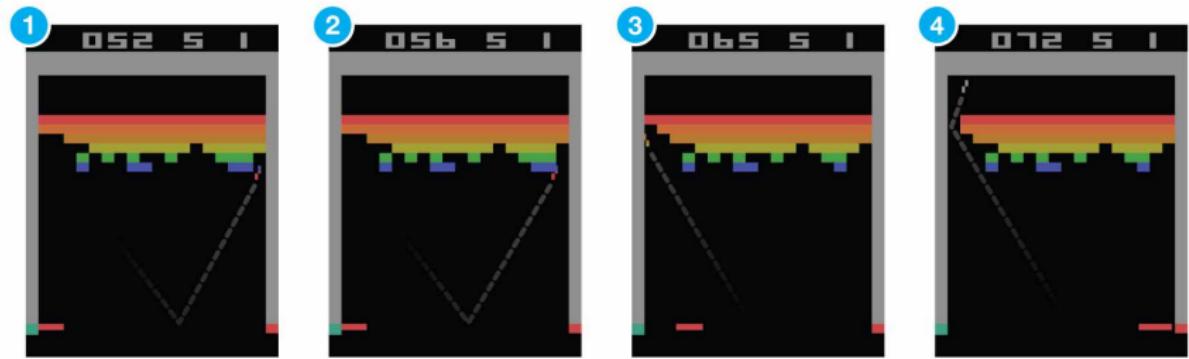
Compared to both the best-performing linear RL algorithm and to an expert human player (allowed to practice for 2 hours on each game).

Outperformed linear in all but two cases, and matched or outperformed the human player in a majority of games, even when limited to choosing an action every 6th frame.

Mnih, et al. (2015)

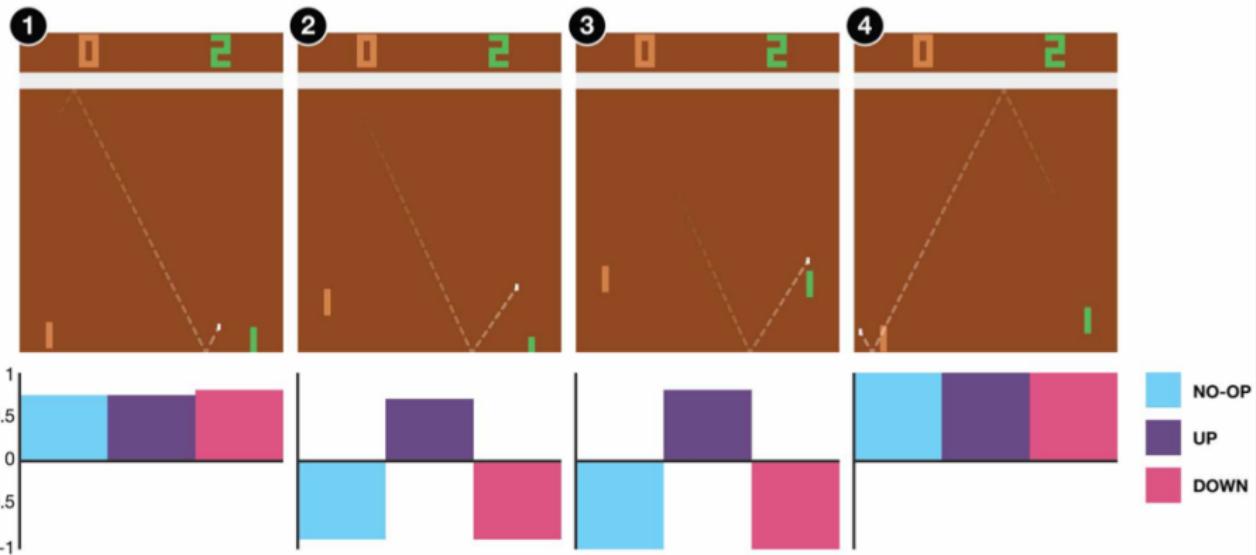
Sample Value Function

Mnih, et al. (2015)



Sample Q-Values

Mnih, et al. (2015)

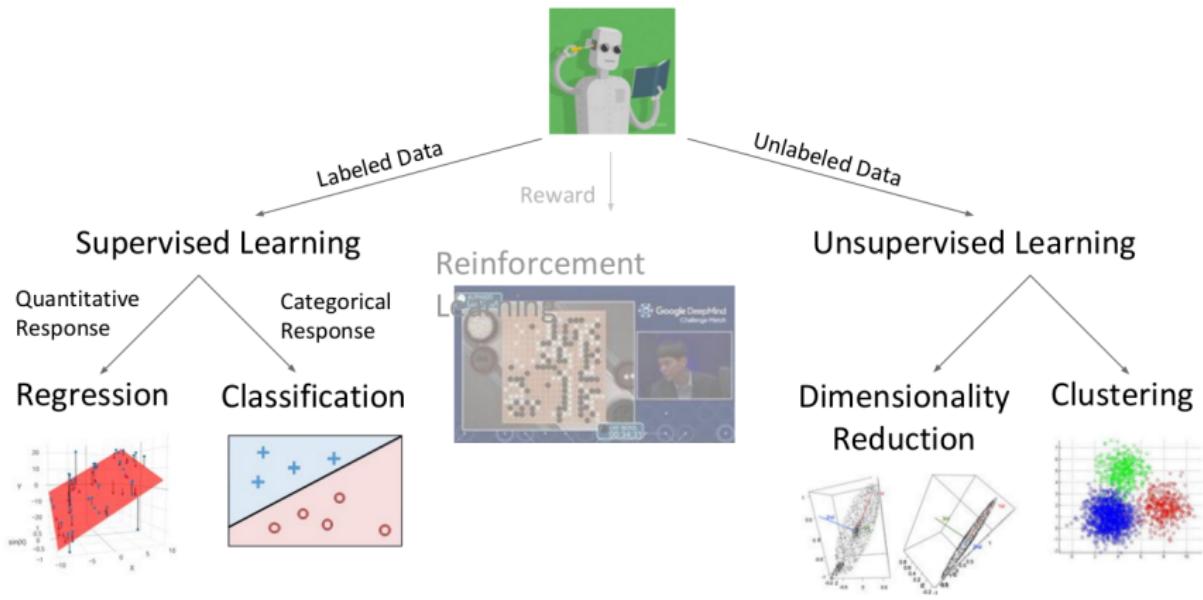


CS 457/557: Machine Learning

Lecture 10-1: Unsupervised Learning: Clustering

Lecture 10-1a: Clustering

Machine Learning Taxonomy



(Image source: DS 100 lecture notes)

Comparing Prediction and Clustering

Definition

Clustering is the process of grouping objects by similarity.

In prediction:

- Input: $\{(\mathbf{x}_i, y_i)\}_{i=1}^N, \mathbf{x}_{N+1}$
- Output: \hat{y}_{N+1}

In clustering:

- Input: $\{\mathbf{x}_i\}_{i=1}^N$
- Output: $\{\hat{y}_i\}_{i=1}^N$

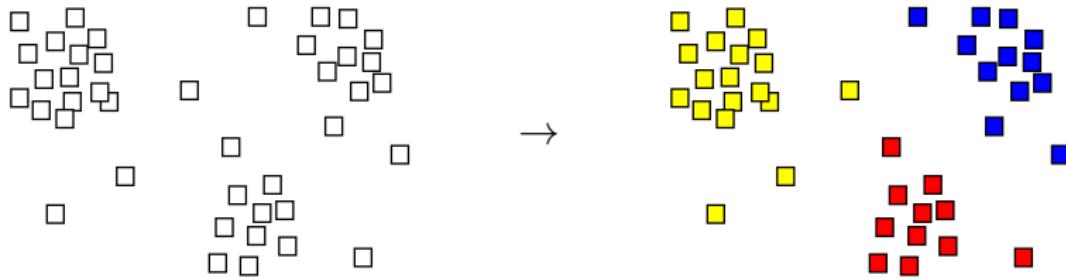
For unsupervised learning in general, we are given *unlabeled* data and we want to extract some structure!

The \hat{y}_i values are **labels** that get assigned to the data such that:

- if $\hat{y}_i = \hat{y}_{i'}$, then $d(\mathbf{x}_i, \mathbf{x}_{i'})$ is **small**
- if $\hat{y}_i \neq \hat{y}_{i'}$, then $d(\mathbf{x}_i, \mathbf{x}_{i'})$ is **large**

(here d is a distance measure)

Purpose of Clustering



Many uses for clustering:

- Identify distinct subpopulations
- Build predictive models within each cluster
- Aggregate data (speeds up comparisons, e.g., in k -NN)
- Detect outliers

The Clustering Problem

How many clusters are there?



- Goal is ill-defined
- Problem becomes harder in higher dimensions

Lecture 10-1b: Clustering Algorithms: k -Means

Clustering as an Optimization Problem

Given a sample $\{\mathbf{x}_i\}_{i=1}^N$ and number of clusters k , find a **partition** of the sample into sets (of indices) C_1, C_2, \dots, C_k that minimizes the sum of **within-cluster variations** (typically *squared Euclidean distance*).

$$\begin{aligned} \min \quad & \sum_{j=1}^k \left(\frac{1}{|C_j|} \sum_{i,i' \in C_j} \|\mathbf{x}_i - \mathbf{x}_{i'}\|_2^2 \right) \\ \text{s.t.} \quad & \bigcup_{j=1}^k C_j = \{1, 2, \dots, n\} \\ & C_j \cap C_{j'} = \emptyset \quad \forall j, j' \in \{1, 2, \dots, k\}, j \neq j' \end{aligned}$$

(here C_j is treated as a set of **indices**, instead of the points themselves)

This is a **hard problem!**

A Reformulation

With a bit of algebra, it is possible to show that

$$\frac{1}{|C_j|} \sum_{i,i' \in C_j} \|\mathbf{x}_i - \mathbf{x}_{i'}\|_2^2 = 2 \sum_{i \in C_j} \|\mathbf{x}_i - \boldsymbol{\mu}_j\|_2^2$$

where $\boldsymbol{\mu}_j = \frac{1}{|C_j|} \sum_{i \in C_j} \mathbf{x}_i$ is the vector of average values for points assigned to cluster C_j , called the **cluster centroid**.

An **idea**:

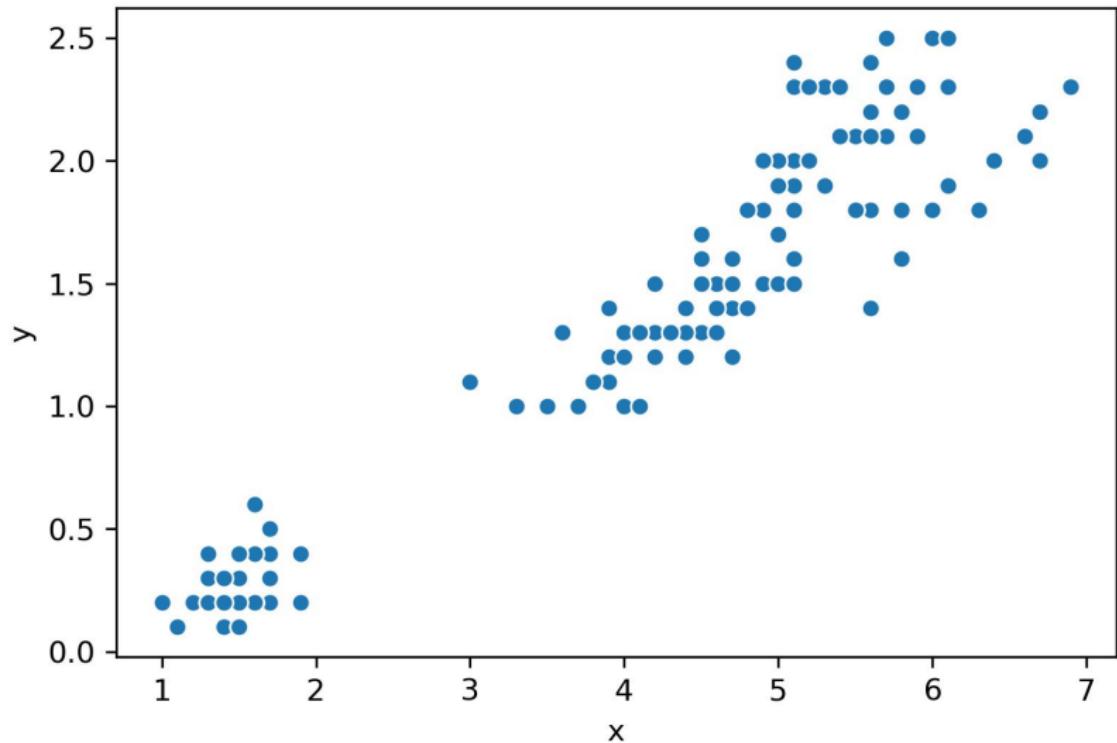
- Specify cluster centroids $\boldsymbol{\mu}_1, \boldsymbol{\mu}_2, \dots, \boldsymbol{\mu}_k$
- Assign points to the cluster whose centroid is closest

k-means Clustering

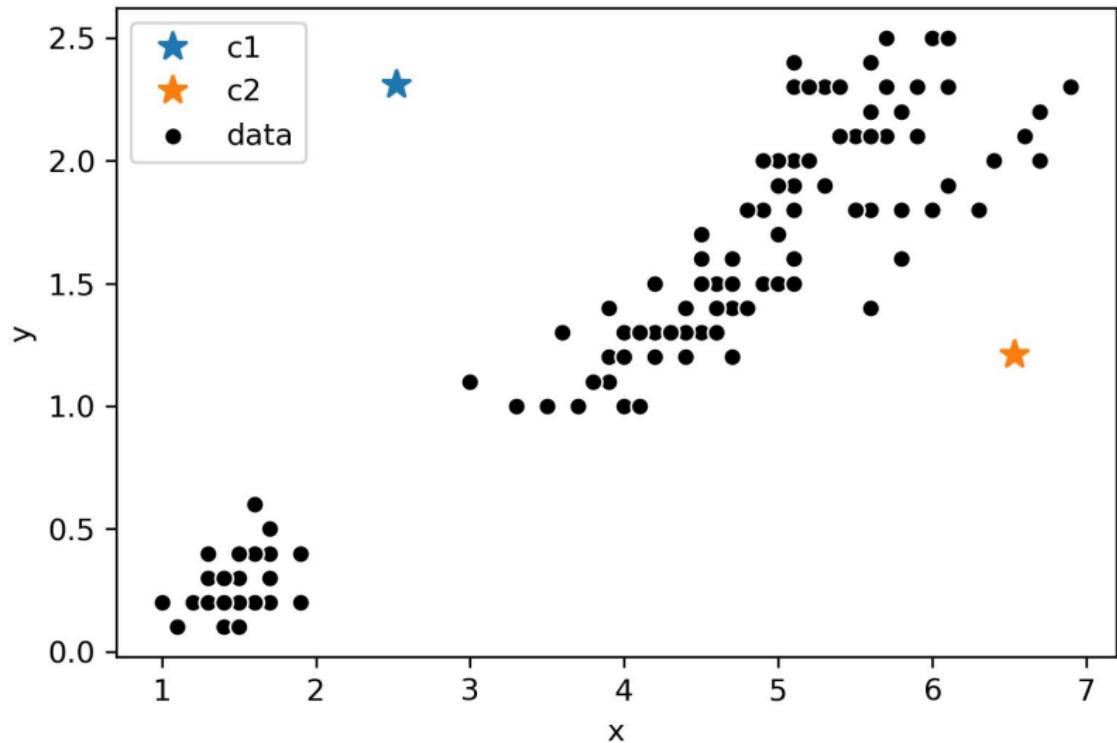
Algorithm 1 *k*-means Clustering Algorithm (Lloyd's Algorithm)

```
procedure KMEANS( $\{\mathbf{x}_i\}_{i=1}^N$ ,  $k$ )
    Select  $k$  points as initial cluster centroids  $\boldsymbol{\mu}_1, \dots, \boldsymbol{\mu}_k$ 
    repeat
        /* Cluster Assignment */
         $C_1, \dots, C_k \leftarrow \emptyset$ 
        for each  $i \in \{1, 2, \dots, N\}$  do
             $j' \leftarrow \arg \min_{j=1}^k \|\mathbf{x}_i - \boldsymbol{\mu}_j\|_2^2$ 
             $C_{j'} \leftarrow C_{j'} \cup \{i\}$ 
        /* Recompute centroids */
        for each  $j \in \{1, 2, \dots, k\}$  do
             $\boldsymbol{\mu}_j \leftarrow \frac{1}{|C_j|} \sum_{i \in C_j} \mathbf{x}_i$ 
    until clusters stop changing
```

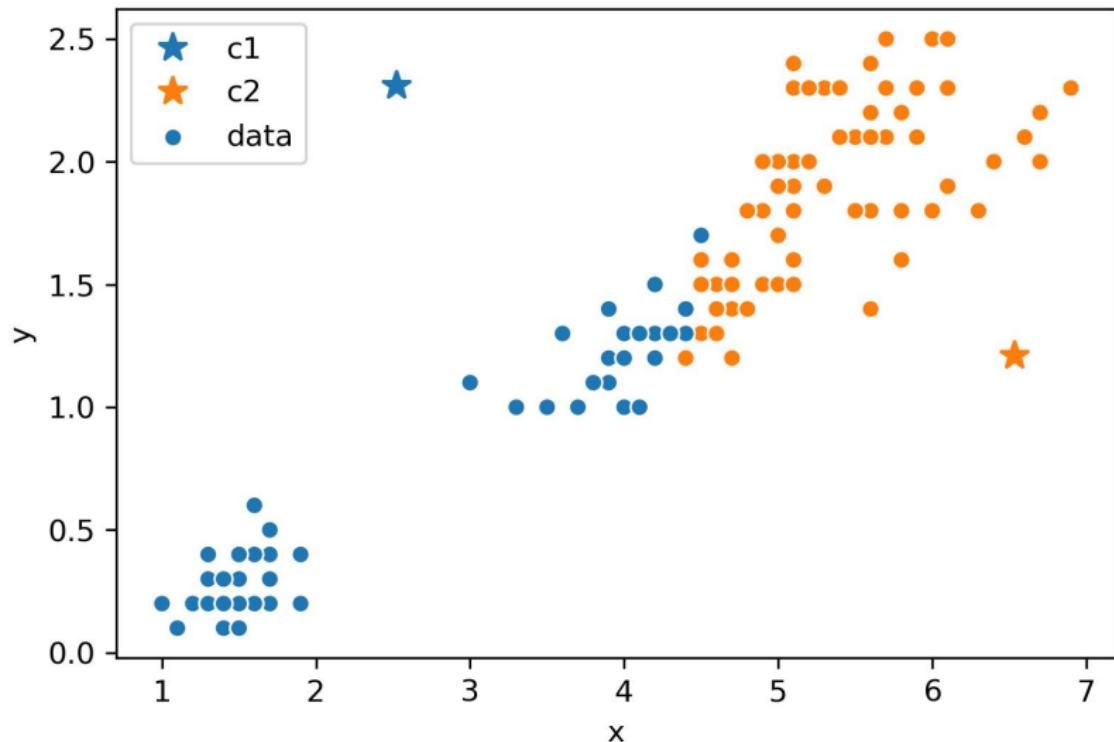
k -means in Action: Raw Data



k-means in Action: Initialization

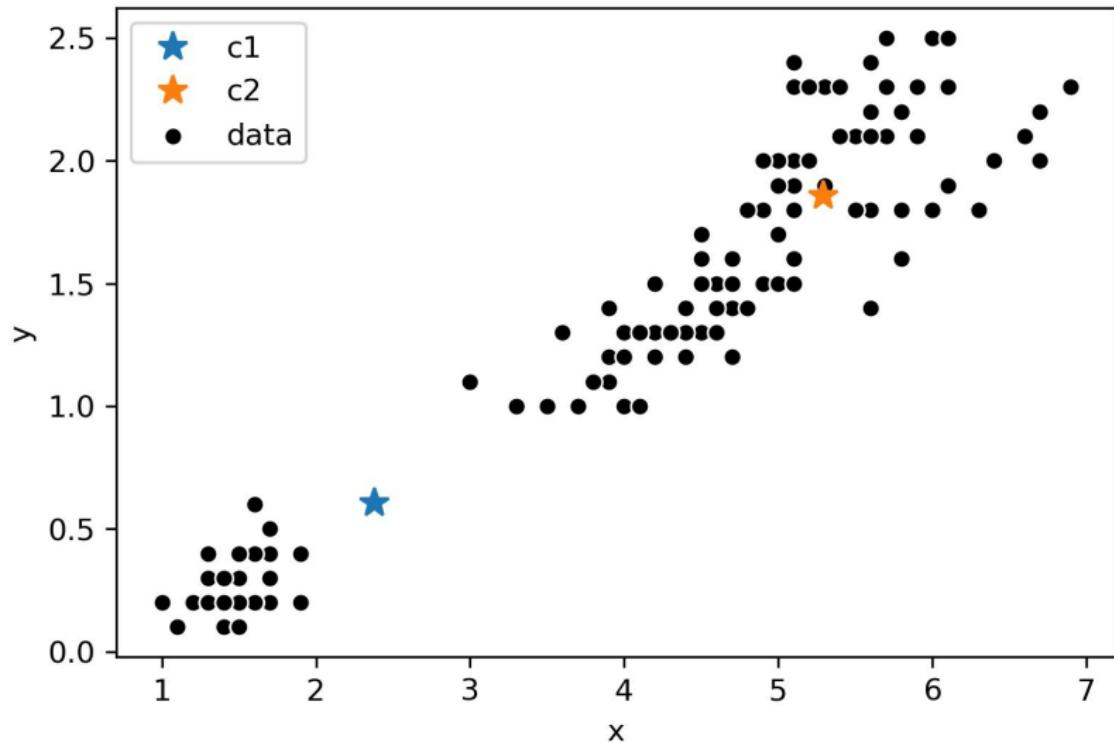


k -means in Action: Iteration 1, Step 1: Assignment

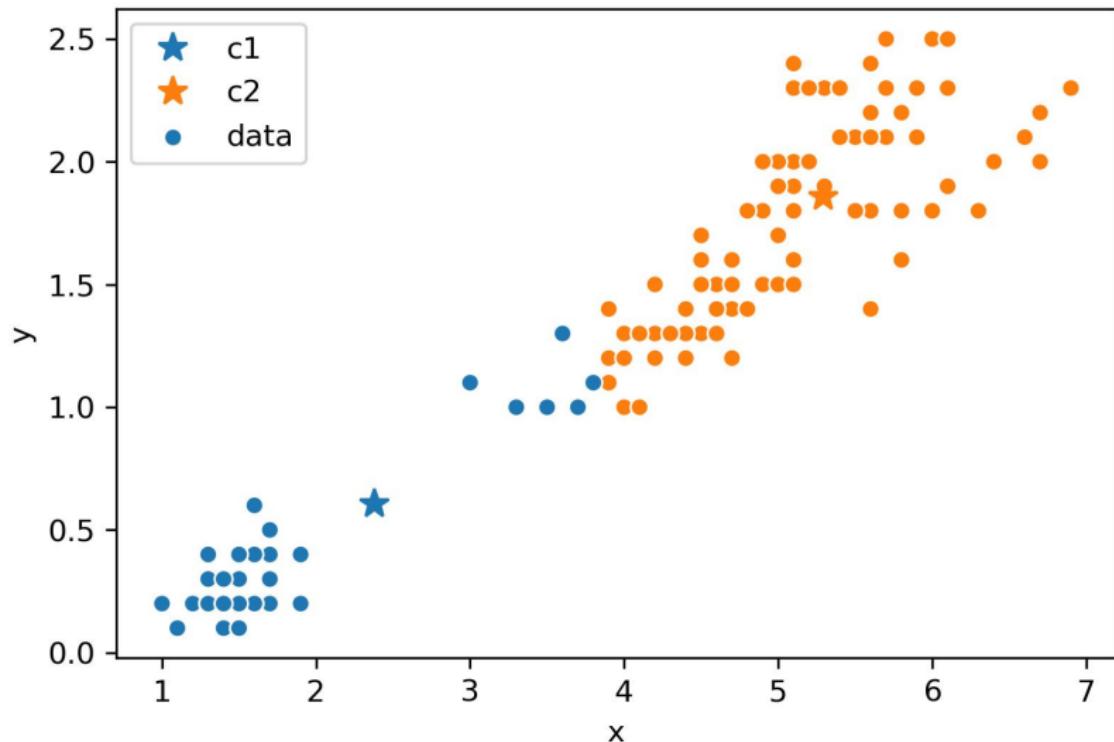


k-means in Action:

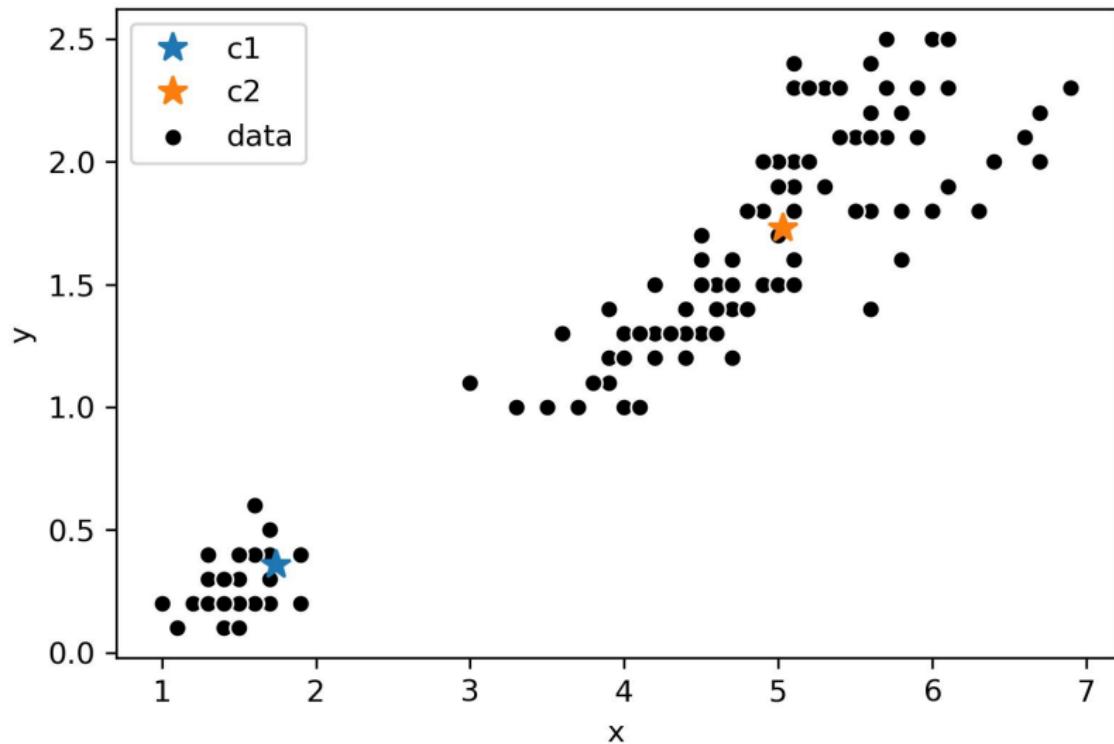
Iteration 1, Step 2: Recentering



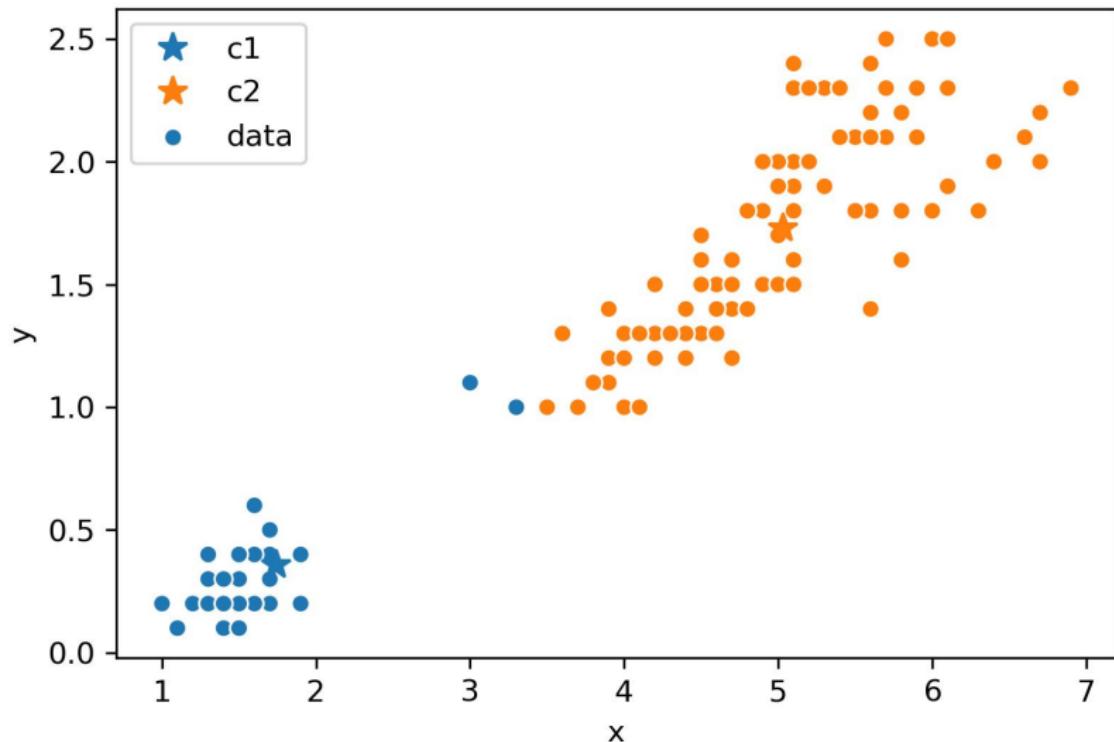
k-means in Action: Iteration 2, Step 1: Assignment



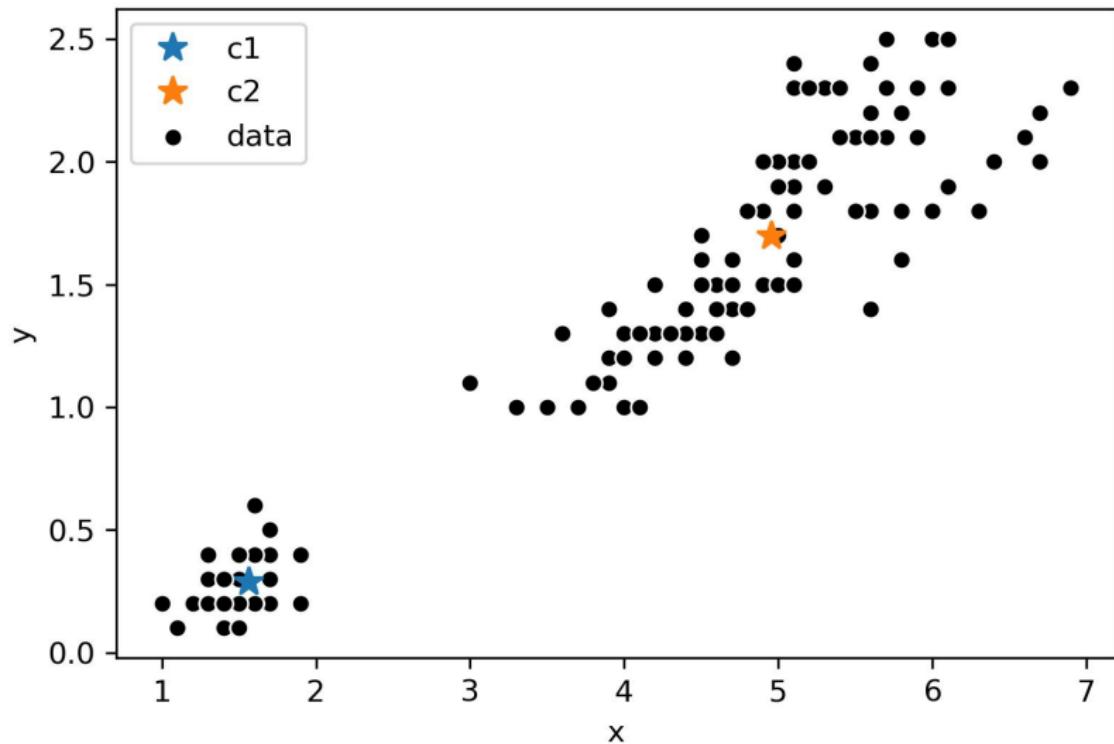
k-means in Action: Iteration 2, Step 2: Recentering



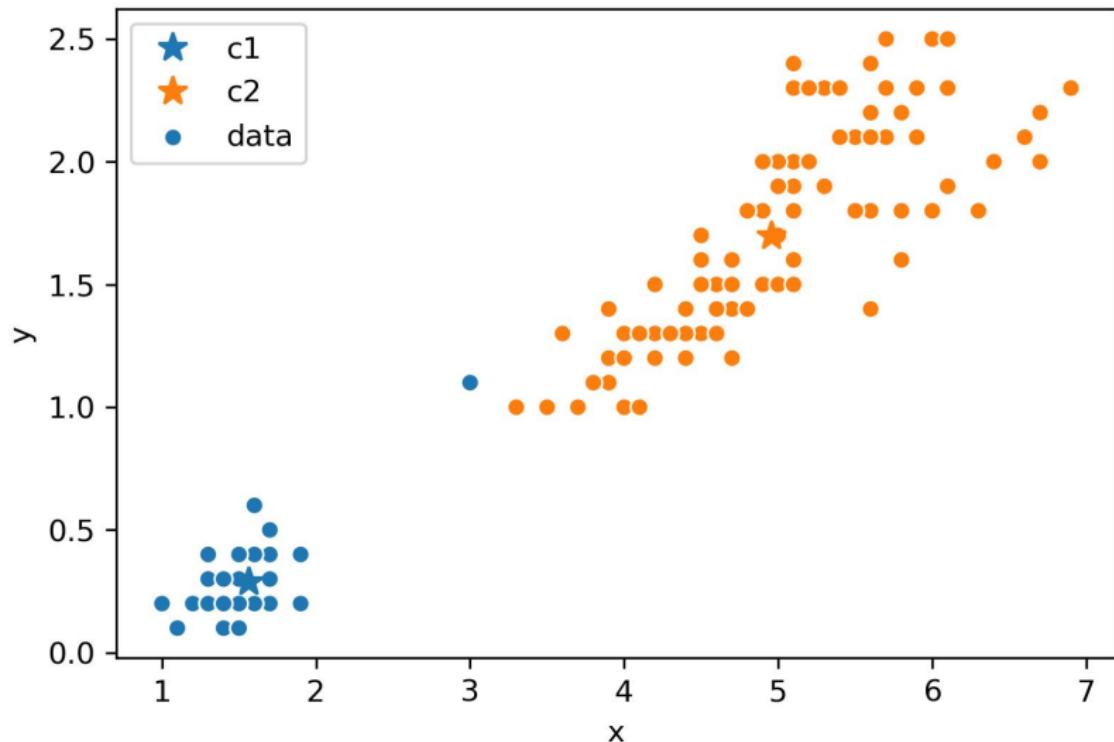
k-means in Action: Iteration 3, Step 1: Assignment



k -means in Action: Iteration 3, Step 2: Recentering

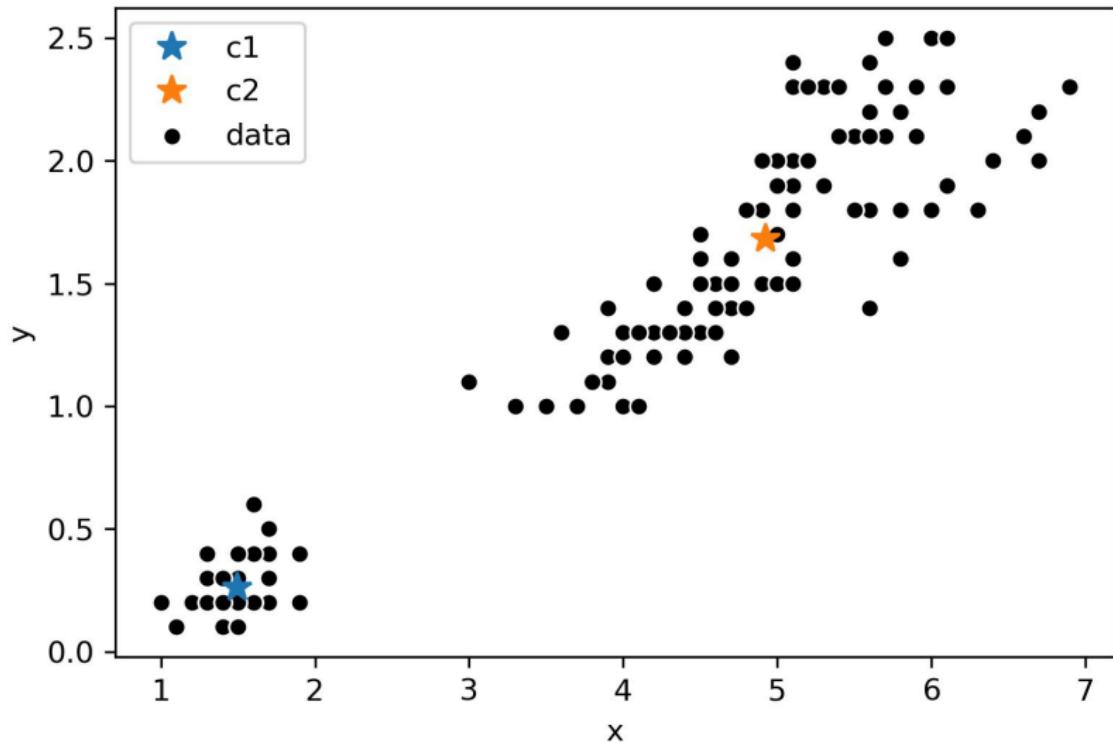


k -means in Action: Iteration 4, Step 1: Assignment

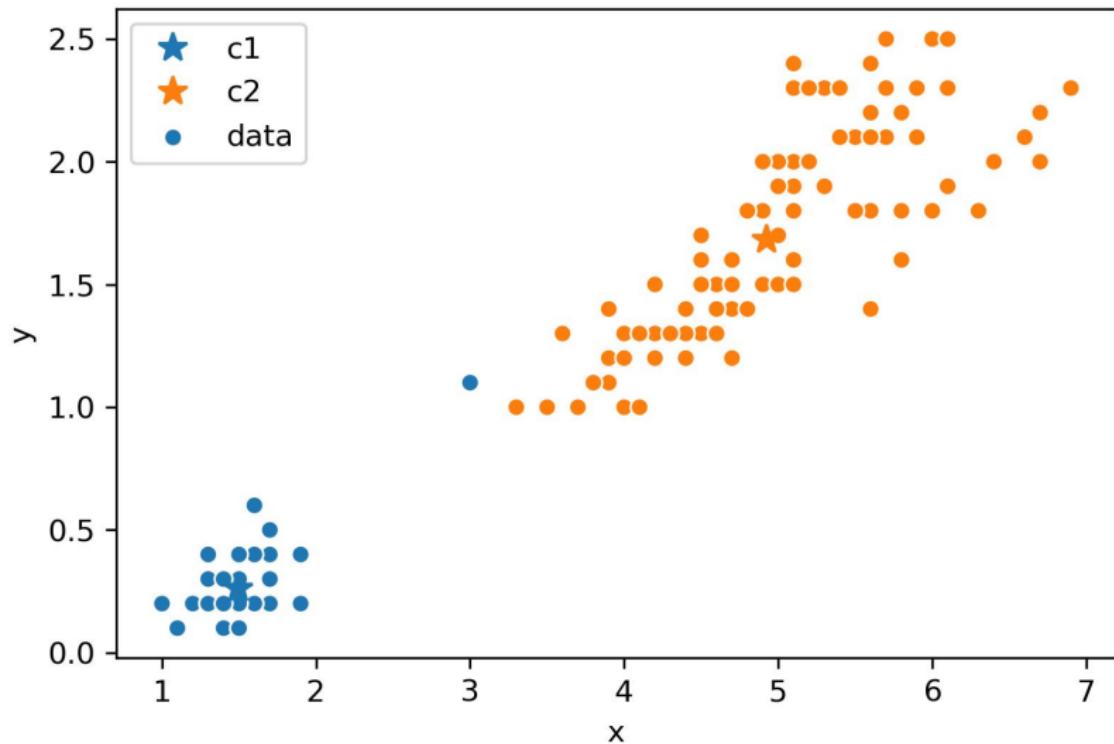


k-means in Action:

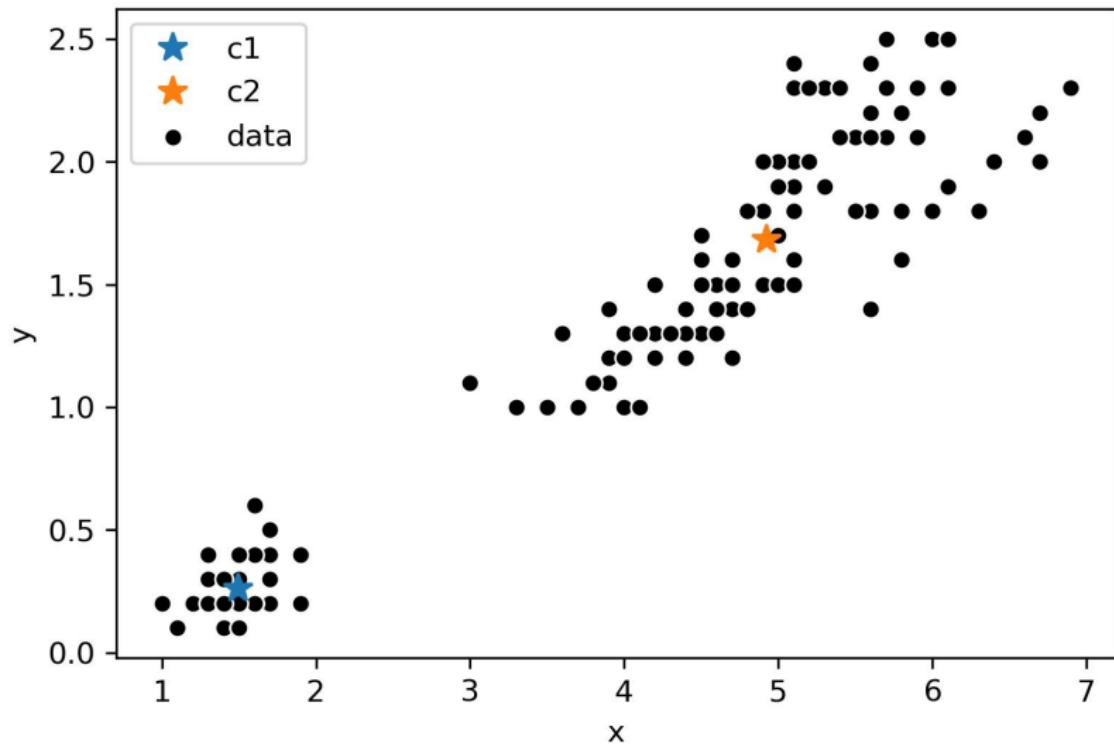
Iteration 4, Step 2: Recentering



k -means in Action: Iteration 5, Step 1: Assignment



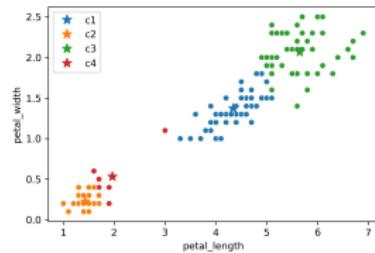
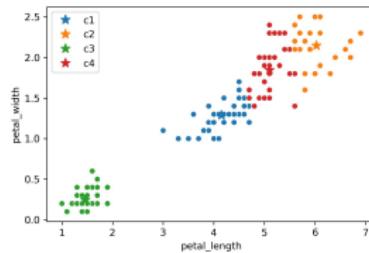
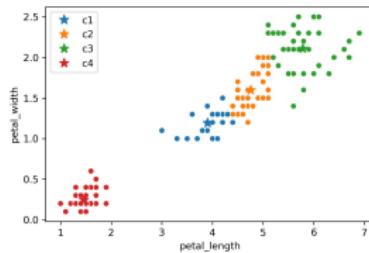
k-means in Action: Iteration 5, Step 2: Recentering



Lecture 10-1c: Additional Notes on k -Means

Properties of Lloyd's Algorithm

- Guaranteed to converge, but **not necessarily** to optimal solution
- Result depends on initialization:



- Choice of k impacts performance
- Sensitive to outliers; doesn't handle nested clusters or long thin clusters well
- Running time per iteration is $O(Npk)$; number of iterations is typically small, but can be large

Initialization of k -means

Some options for choosing initial cluster centroids:

- Select k random points in space
- Select k random points from sample
- Randomly assign the points in the sample to clusters, and compute initial cluster centroids from this
- k -means++ chooses cluster centroids sequentially:
 - ① Set μ_1 to random point from sample, with uniform probability
 - ② For each $j \in \{2, 3, \dots, k\}$:
 - Set μ_j to a random point from the sample, with each point's probability proportional to its minimum squared distance to all other centroids

Use multiple restarts and select the best final solution to try to avoid bad local optima!

Centroids or Central Points

k -means updates each cluster centroid as:

$$\boldsymbol{\mu}_j \leftarrow \frac{1}{|C_j|} \sum_{i \in C_j} \mathbf{x}_i$$

Works well for numerical data, but not categorical data

Instead of averaging points in a cluster together to get a **centroid**, we can choose a centrally located point in the cluster to get a **medoid**!

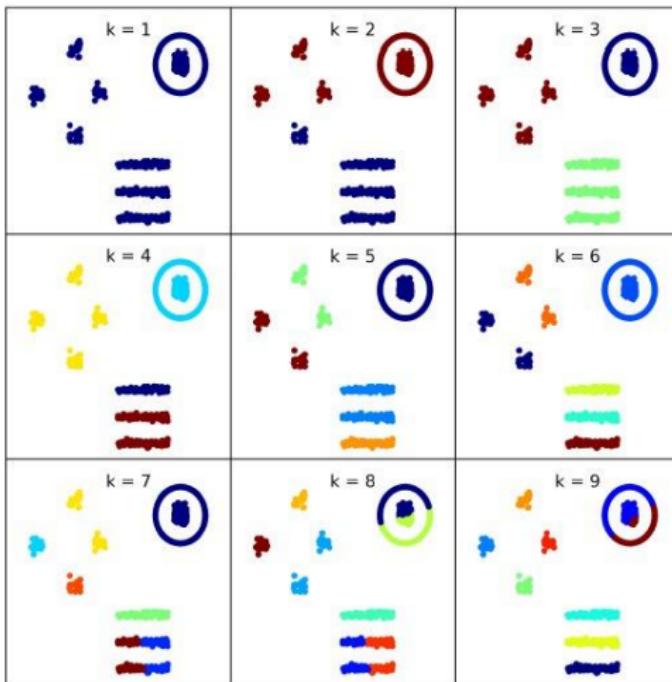
The k -**medoids** algorithm updates clusters using:

- $i^* \leftarrow \arg \min_{i \in C_j} \frac{1}{|C_j|} \sum_{i' \in C_j} \|\mathbf{x}_i - \mathbf{x}_{i'}\|_2^2$
- $\boldsymbol{\mu}_j \leftarrow \mathbf{x}_{i^*}$

Does better than k -means for categorical data, but requires more work.
(k -medians is another variant for dealing with categorical data, as well.)

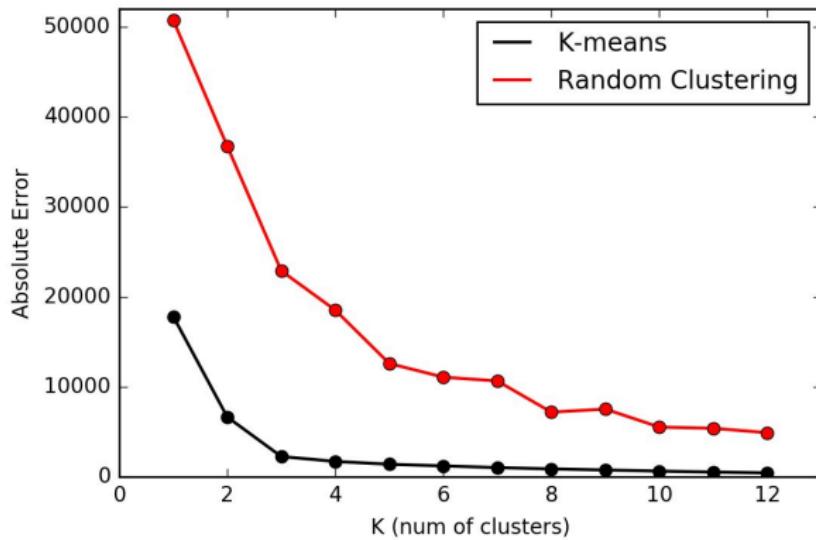
How Many Clusters?

k is usually **unknown** in advance:



Determining the Number of Clusters

Graphical technique known as the “knee” or “elbow” method:

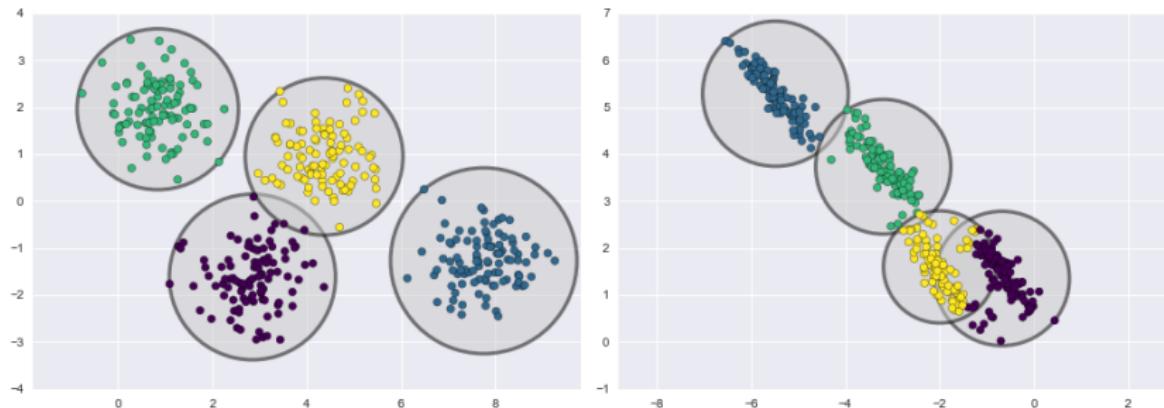


Other algorithms (e.g., mean shift) can be used to pick k automatically

Lecture 10-1d: Clustering with Gaussian Mixture Models

Handling Non-Spherical Clusters

k -Means is only able to construct spherical clusters.



Sometimes this works, but sometimes it doesn't.

Ideally, we want to:

- Increase options for cluster shape
- Relax cluster assignments: probabilistic instead of discrete

Gaussian Mixture Models

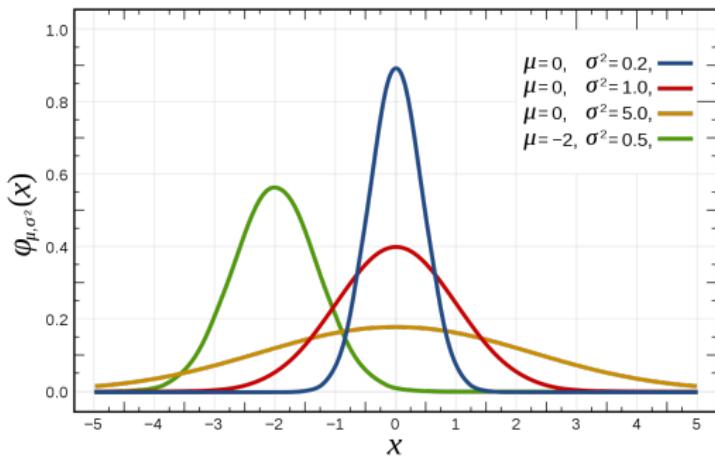
Definition

A **Gaussian mixture model** is a probabilistic model that assumes that the sample is generated from a finite number of underlying Gaussian (normal) distributions with unknown parameters.

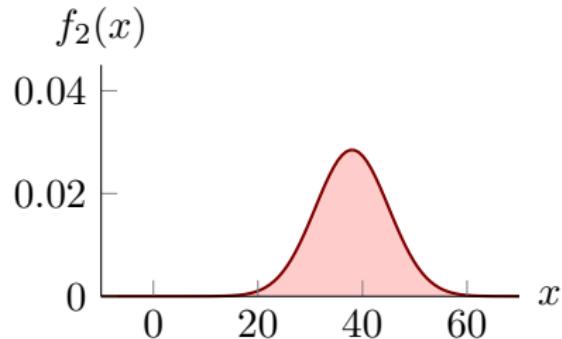
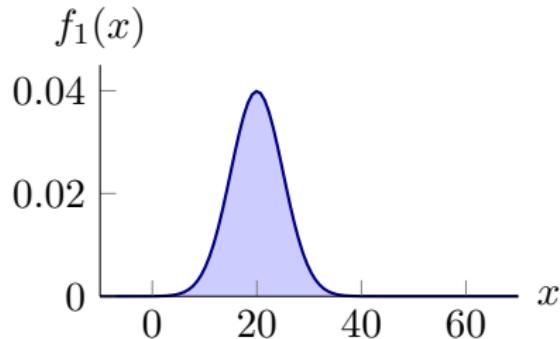
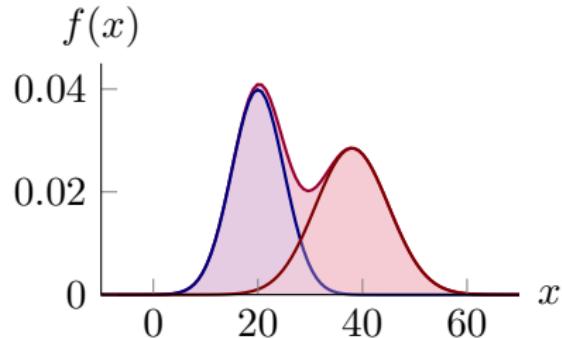
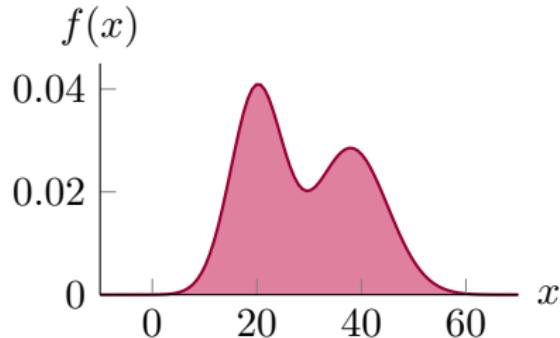
Univariate normal

distribution characterized by two parameters:

- μ : Mean
- σ^2 : Variance



Simple Mixture Model



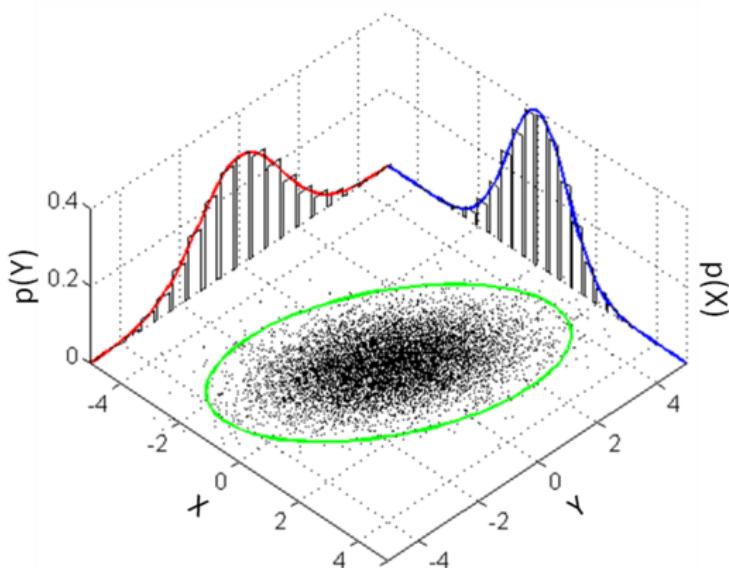
Multivariate Normal Distribution

Multivariate normal distribution in p dimensions characterized by

- μ : p -dimensional mean vector
- Σ : $p \times p$ covariance matrix

Example on right uses

$$\mu = \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \quad \Sigma = \begin{bmatrix} 1 & \frac{3}{5} \\ \frac{3}{5} & 2 \end{bmatrix}$$



Mixture of Multivariate Normal Distributions

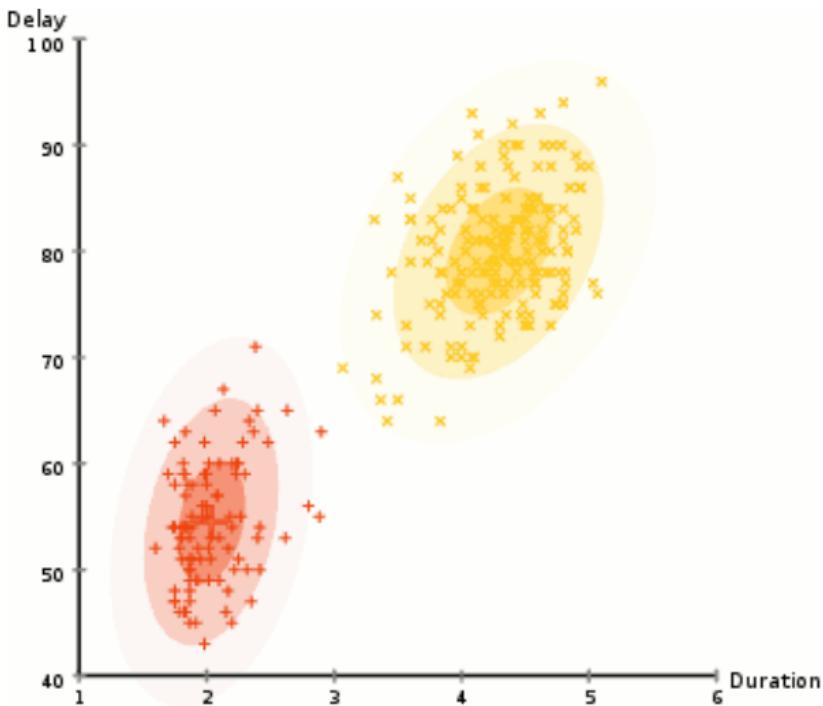


Image from Wikipedia, by Chire, CC BY-SA 3.0

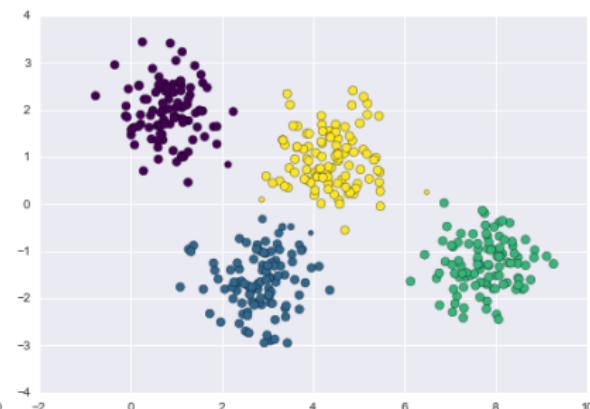
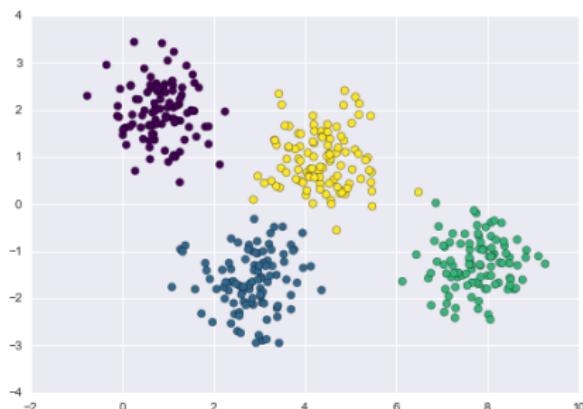
Gaussian Mixture Models

k -means with p -dimensional data:

- Each cluster C_j , $j \in \{1, 2, \dots, k\}$, represented by centroid $\mu_j \in \mathbb{R}^p$

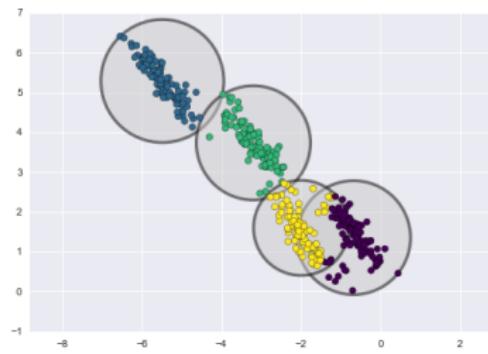
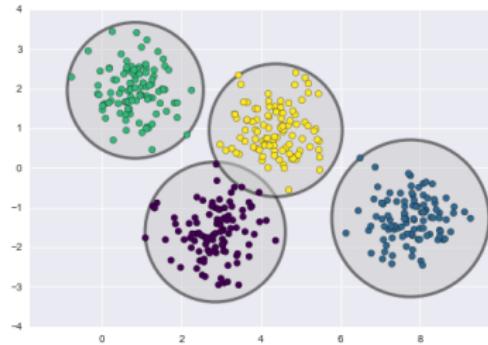
GMM with p -dimensional data:

- Each cluster C_j , $j \in \{1, 2, \dots, k\}$, represented by multivariate normal distribution with mean vector $\mu_j \in \mathbb{R}^p$ and covariance matrix $\Sigma_j \in \mathbb{R}^{p \times p}$

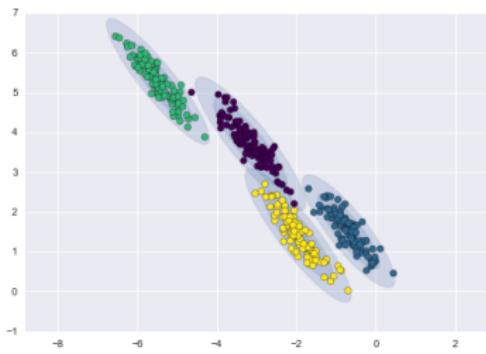
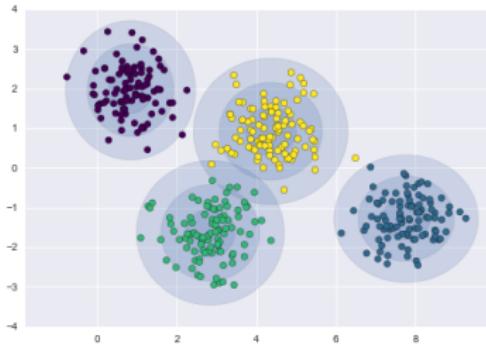


Comparing Cluster Shapes and Boundaries

k-Means



GMM



Fitting a GMM: The Expectation-Maximization Algorithm

Lloyd's algorithm for k -means with p -dimensional data:

- Pick initial guesses for centroids $\mu_1, \mu_2, \dots, \mu_k$
- Iterate:
 - For each point, assign it to cluster with nearest centroid
 - Update each cluster's centroid based on assigned points

Expectation-Maximization (EM) algorithm for GMM with p -dimensional data:

- Pick initial guesses for parameters $\mu_1, \mu_2, \dots, \mu_k$ and $\Sigma_1, \dots, \Sigma_k$
- Iterate:
 - For each point, assign probabilities for cluster membership based on **each** cluster's parameters
 - Update each cluster's parameters based on **all** points, weighted by probability of membership

EM Algorithm in Action

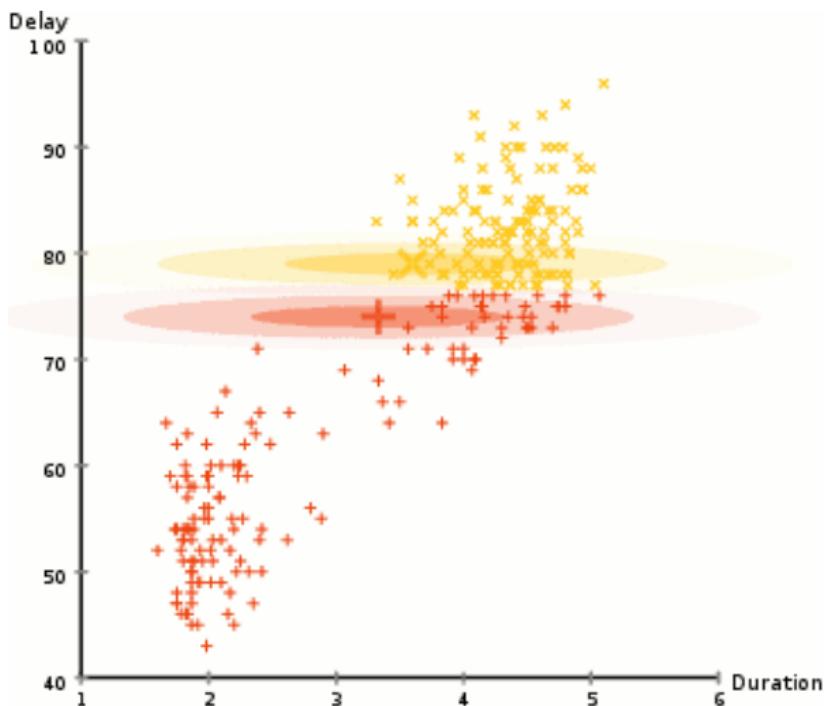


Image from Wikipedia, by Chire, CC BY-SA 3.0

EM Algorithm in Action

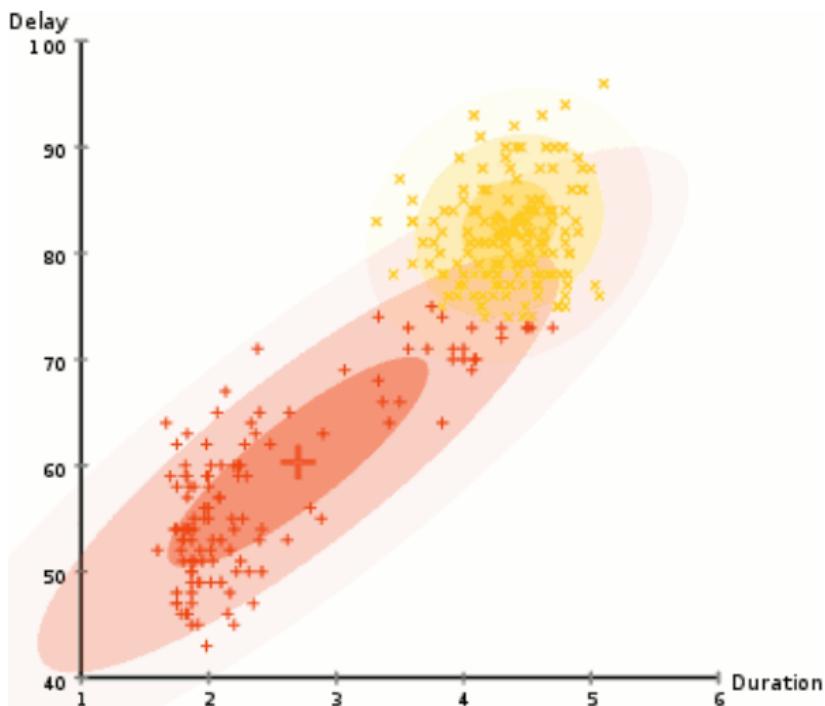


Image from Wikipedia, by Chire, CC BY-SA 3.0

EM Algorithm in Action

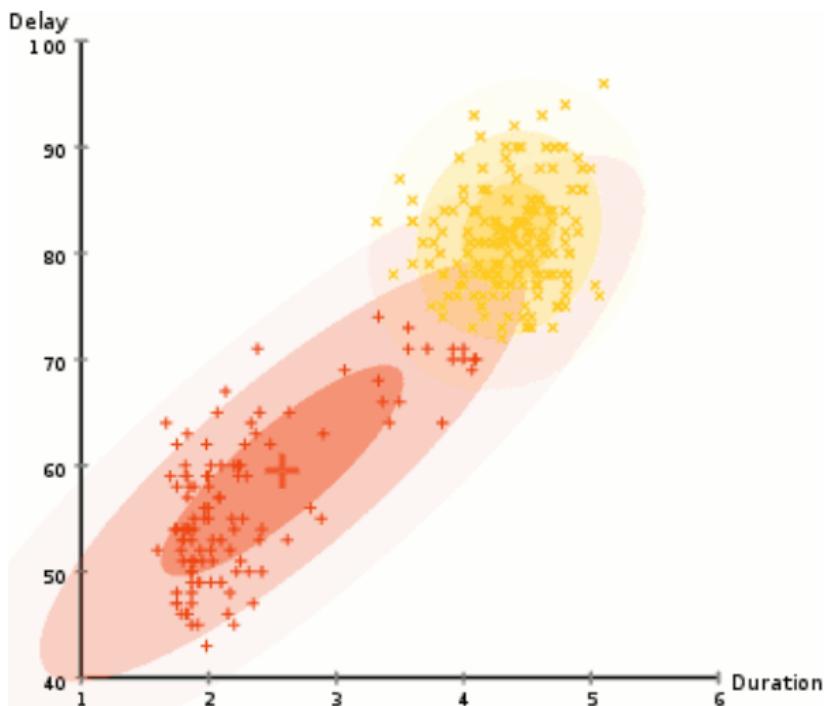


Image from Wikipedia, by Chire, CC BY-SA 3.0

EM Algorithm in Action

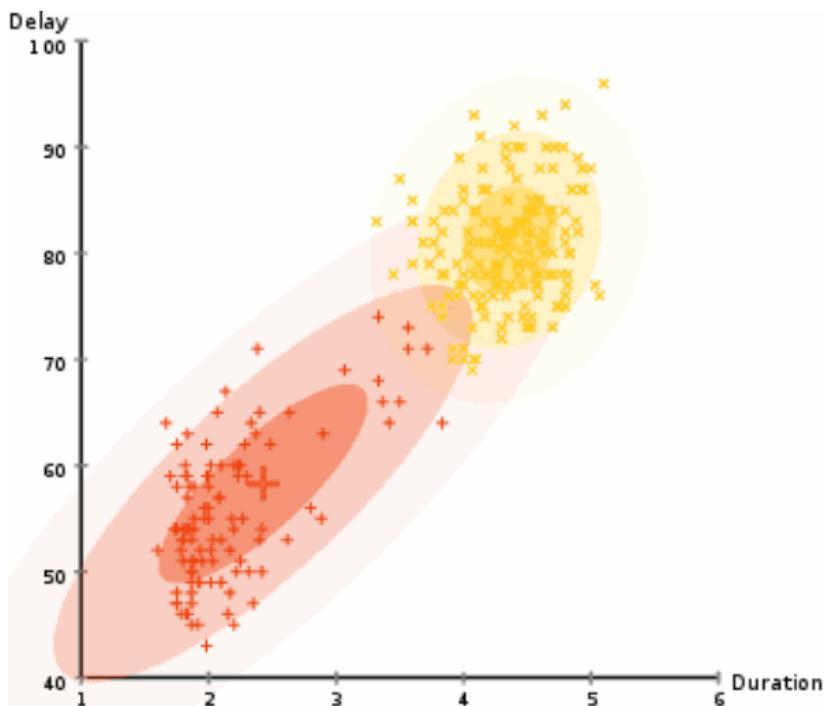


Image from Wikipedia, by Chire, CC BY-SA 3.0

EM Algorithm in Action

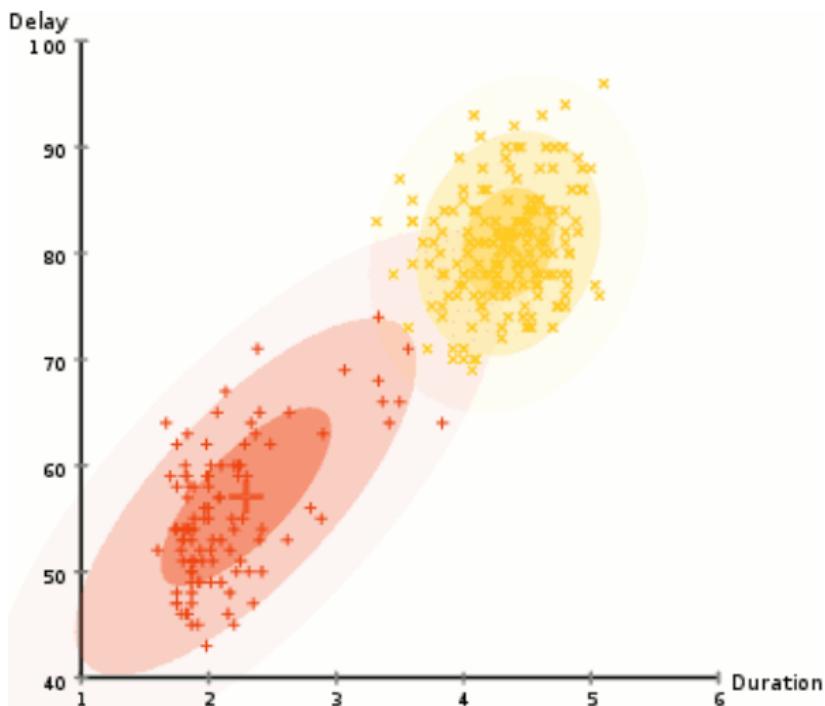


Image from Wikipedia, by Chire, CC BY-SA 3.0

EM Algorithm in Action

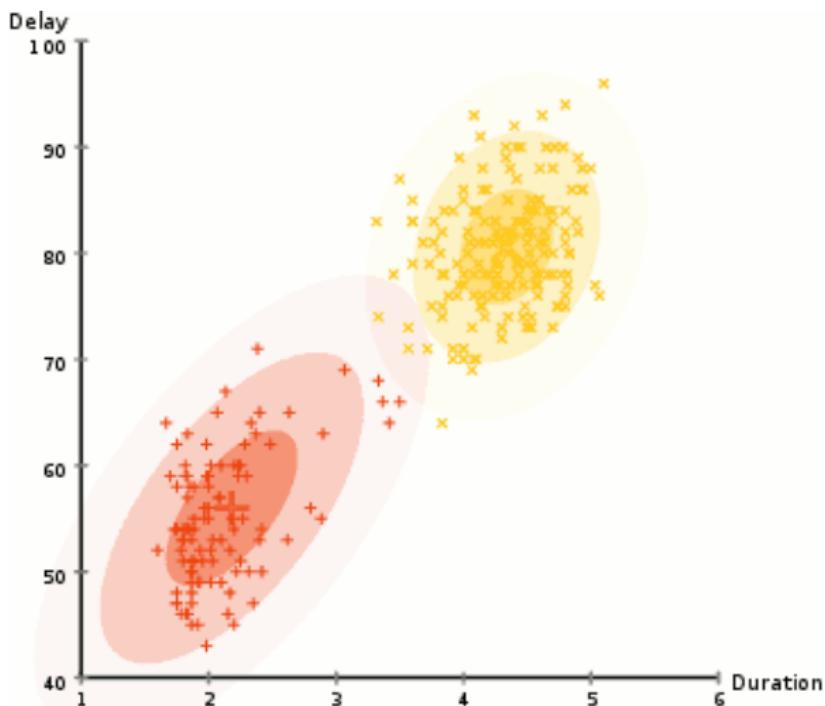


Image from Wikipedia, by Chire, CC BY-SA 3.0

EM Algorithm in Action

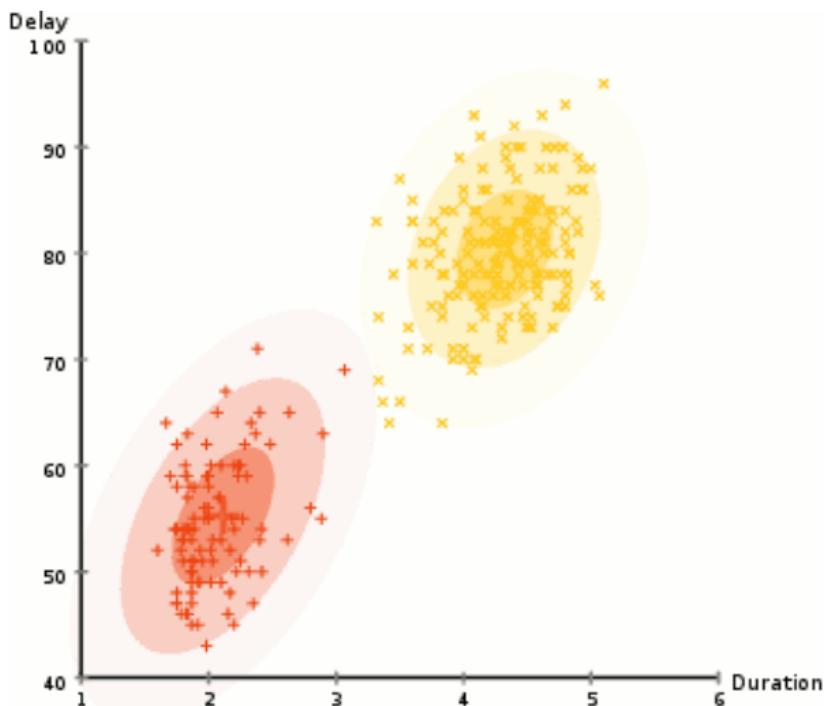


Image from Wikipedia, by Chire, CC BY-SA 3.0

EM Algorithm in Action

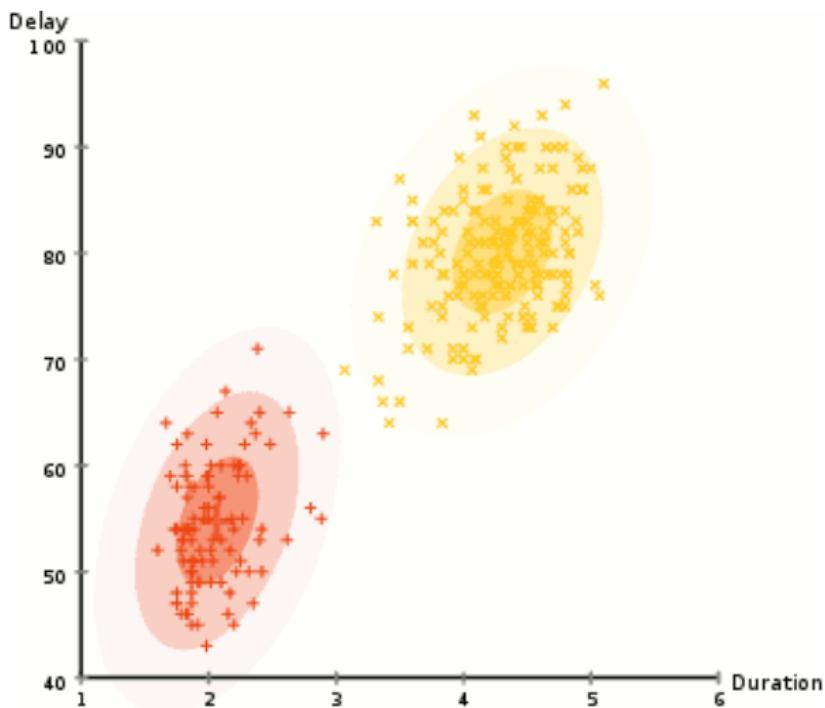


Image from Wikipedia, by Chire, CC BY-SA 3.0

EM Algorithm in Action

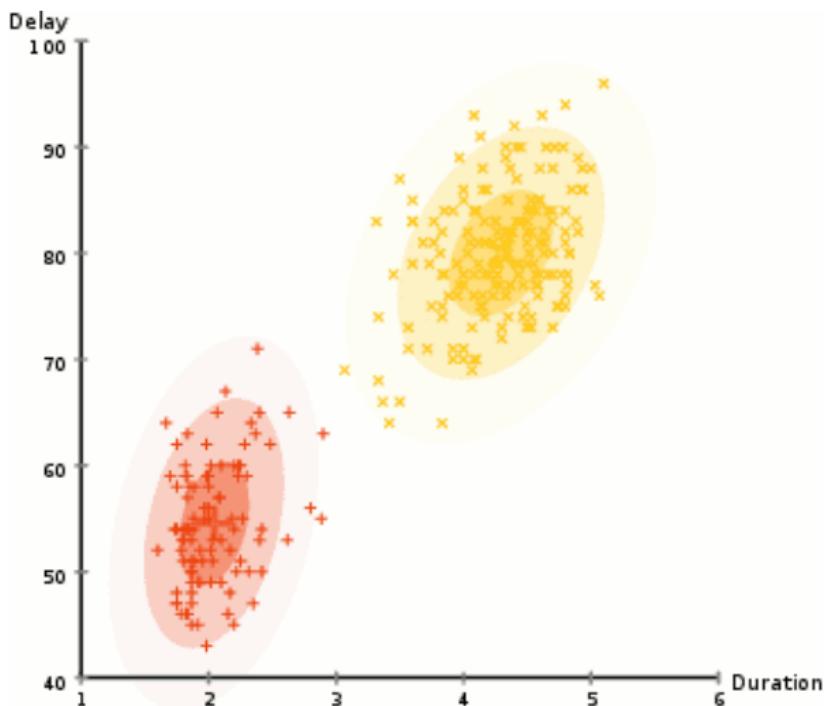


Image from Wikipedia, by Chire, CC BY-SA 3.0

EM Algorithm in Action

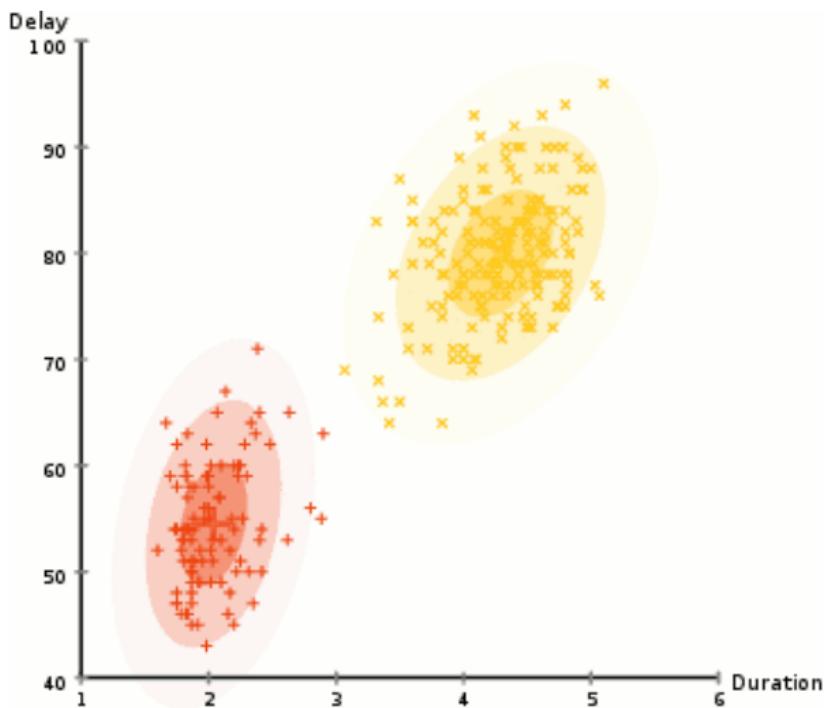


Image from Wikipedia, by Chire, CC BY-SA 3.0

Properties of GMM Clustering

- Converges to local optimum
- Use multiple restarts with different initial solutions
- Still need to know number of clusters in advance
- Need large amount of data points per cluster to obtain good estimates for cluster parameters μ_j and Σ_j

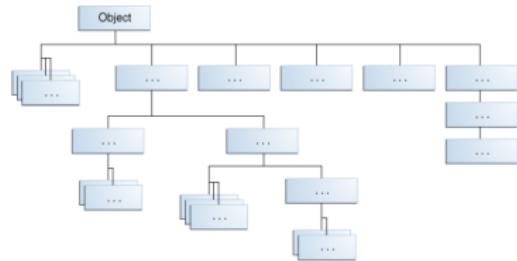
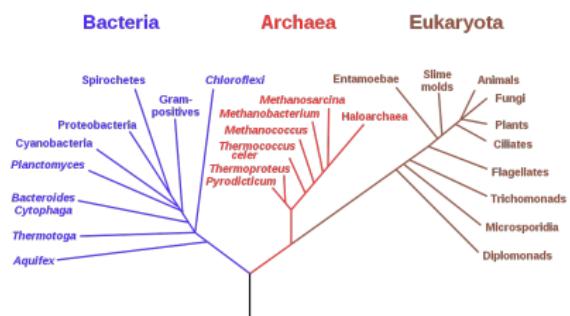
Lecture 10-1e: Hierarchical Clustering

Another Approach to Clustering

k-Means and GMMs **both** require knowing the number of clusters beforehand.

Hierarchical clustering is an approach that avoids this requirement.

A hierarchy organizes items in various levels:



Source: <https://docs.oracle.com/javase/tutorial/java/IandI/subclasses.html>

Hierarchical Clustering

Two main types:

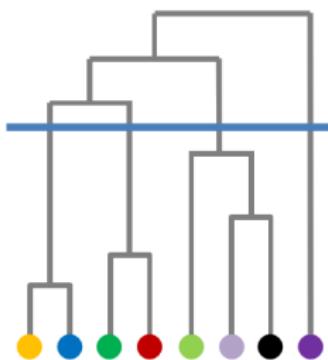
- **Divisive clustering** (top-down approach)
 - All points start in one cluster
 - Clusters get recursively subdivided as we “move down”
- **Agglomerative clustering** (bottom-up approach)
 - All points start in separate clusters
 - Clusters get merged together as we “move up”

Algorithm 1 Hierarchical Agglomerative Clustering Algorithm

```
procedure AGGLOMERATIVECLUSTERING( $\{\mathbf{x}_i\}_{i=1}^N$ )
  for each  $i \in \{1, 2, \dots, N\}$  do
     $C_i \leftarrow \{i\}$ 
  while more than one cluster remains do
    Merge closest two clusters  $C_i$  and  $C_j$ 
```

Dendograms

Hierarchical clustering produces a **dendrogram**:



- Height of a merge indicates dissimilarity between merged clusters
- Items permuted along horizontal axis to allow for adjacent merging
- Cuts at any height produce clusters

Cluster Distance

For two points \mathbf{x}_i and $\mathbf{x}_{i'}$, the dissimilarity $d(\mathbf{x}_i, \mathbf{x}_{i'})$ can be a distance measure (typically Euclidean, but other options are possible).

For two clusters C_j and $C_{j'}$, the dissimilarity $d(C_j, C_{j'})$ is typically defined using one of the following **linkage** options:

- **Single linkage** (nearest neighbor): Minimal intercluster dissimilarity

$$d(C_j, C_{j'}) = \min_{i \in C_j, i' \in C_{j'}} d(\mathbf{x}_i, \mathbf{x}_{i'})$$

- **Average linkage**: Mean intercluster dissimilarity

$$d(C_j, C_{j'}) = \frac{1}{|C_j||C_{j'}|} \sum_{i \in C_j} \sum_{i' \in C_{j'}} d(\mathbf{x}_i, \mathbf{x}_{i'})$$

- **Complete linkage** (farthest neighbor): Maximal intercluster dissimilarity

$$d(C_j, C_{j'}) = \max_{i \in C_j, i' \in C_{j'}} d(\mathbf{x}_i, \mathbf{x}_{i'})$$

Cluster Distance (Continued)

For two clusters C_j and $C_{j'}$, the dissimilarity $d(C_j, C_{j'})$ is typically defined using one of the following **linkage** options:

- **Centroid**: Dissimilarity between cluster centroids

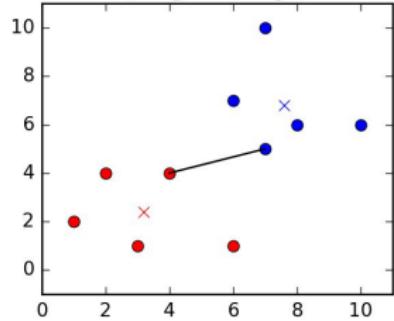
$$d(C_j, C_{j'}) = d \left(\frac{1}{|C_j|} \sum_{i \in C_j} \mathbf{x}_i, \frac{1}{|C_{j'}|} \sum_{i' \in C_{j'}} \mathbf{x}_{i'} \right)$$

- **Ward**: Increase in within-cluster variation after merging

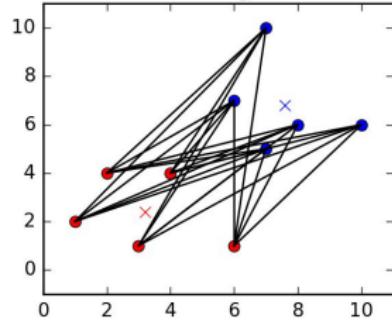
$$\begin{aligned} d(C_j, C_{j'}) = & \sum_{i \in C_j \cup C_{j'}} \|\mathbf{x}_i - \boldsymbol{\mu}_{j''}\|_2^2 \\ & - \sum_{i \in C_j} \|\mathbf{x}_i - \boldsymbol{\mu}_j\|_2^2 - \sum_{i \in C_{j'}} \|\mathbf{x}_i - \boldsymbol{\mu}_{j'}\|_2^2 \end{aligned}$$

Linkage Options

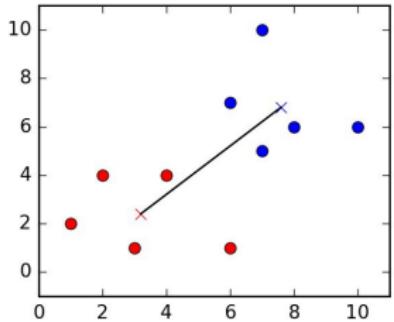
Nearest Neighbor
(Single Linkage)



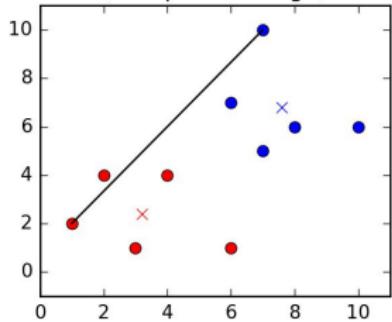
Average



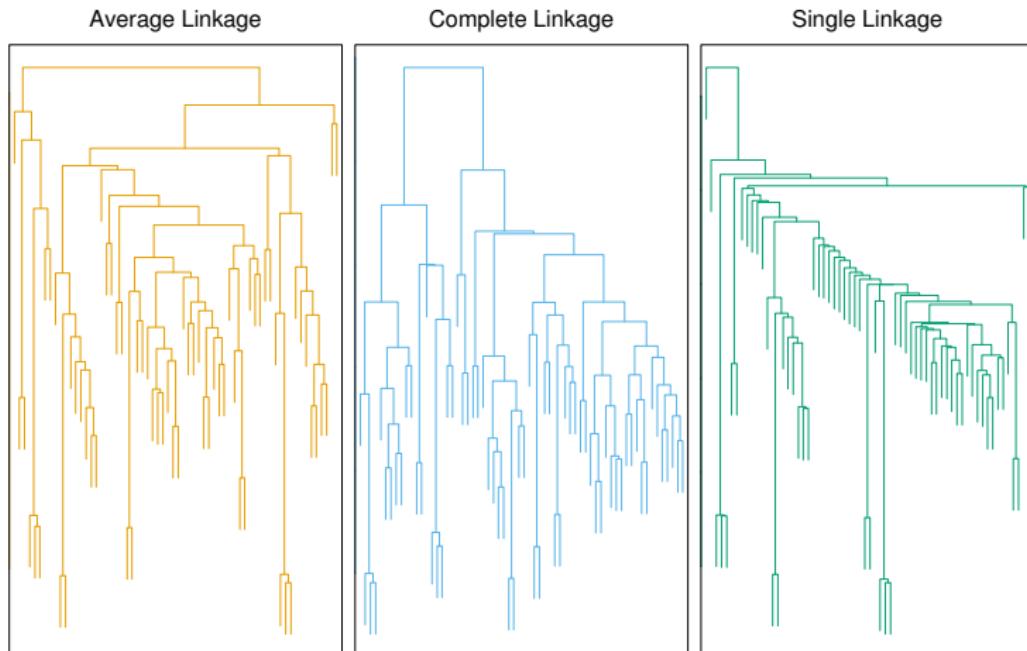
Centroid



Furthest Neighbor
(Complete Linkage)



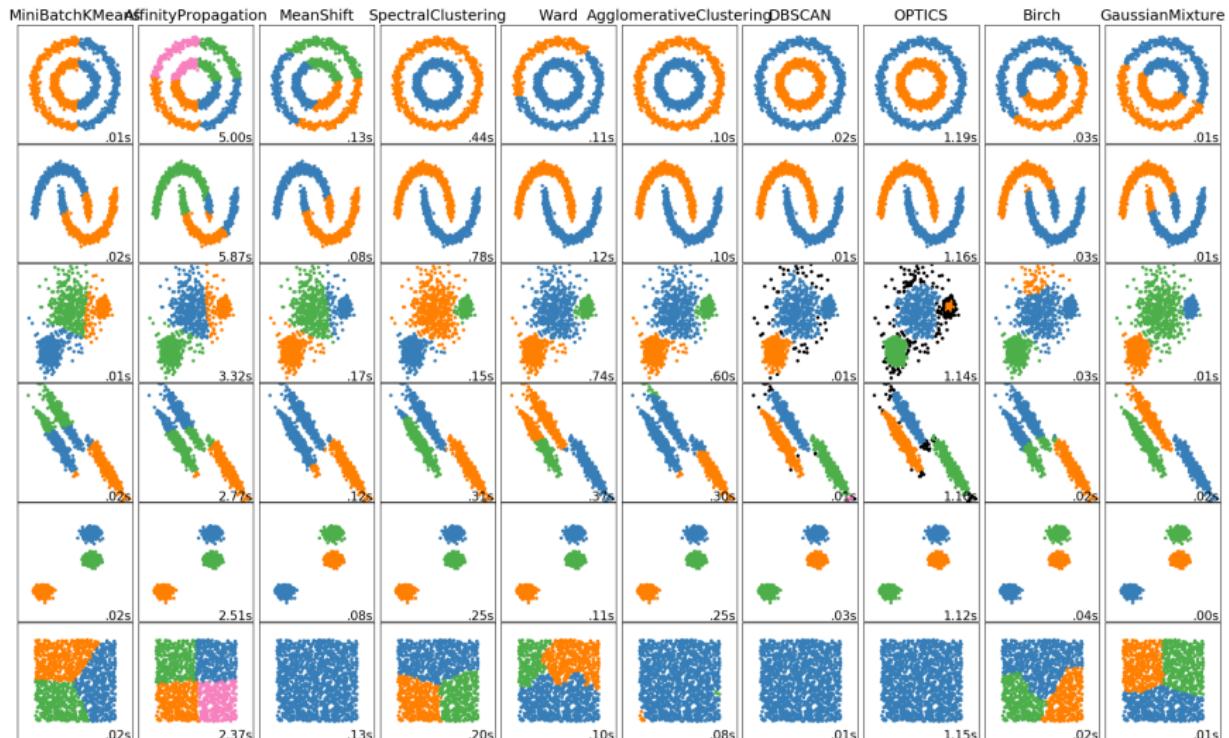
Results from Linkage Options



Taken from "An Introduction to Statistical Learning, with applications in R" (Springer, 2013)
with permission from the authors: G. James, D. Witten, T. Hastie and R. Tibshirani.

Lecture 10-1f: Additional Notes on Clustering

Lots of Additional Clustering Algorithms!



https://scikit-learn.org/stable/auto_examples/cluster/plot_cluster_comparison.html

For Further Reading

k-Means, GMM Clustering, and EM Algorithm

- <https://jakevdp.github.io/PythonDataScienceHandbook/05.11-k-means.html>
- <https://jakevdp.github.io/PythonDataScienceHandbook/05.12-gaussian-mixtures.html>
- <https://scikit-learn.org/stable/modules/mixture.html>
- https://en.wikipedia.org/wiki/Expectation%E2%80%93maximization_algorithm
- https://en.wikipedia.org/wiki/Mixture_model

Hierarchical Clustering

- https://en.wikipedia.org/wiki/Hierarchical_clustering

Other Algorithms

- DBSCAN and HDBSCAN (density-based clustering)
- Spectral clustering

CS 457/557: Machine Learning

Lecture 10-2: Unsupervised Learning: Dimensionality Reduction

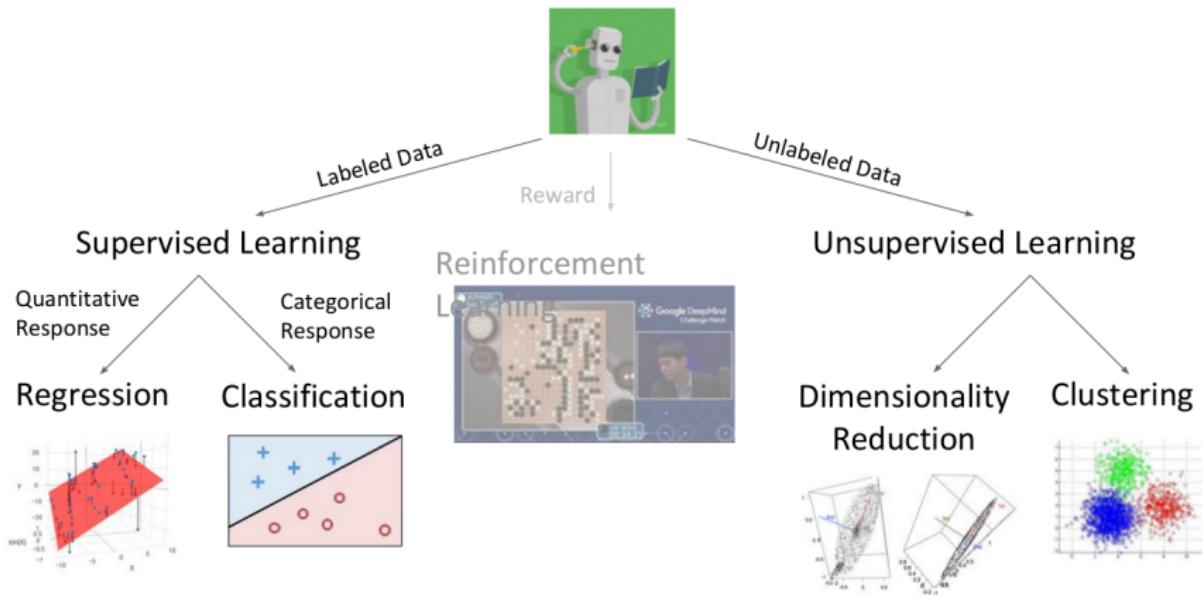
UNIVERSITY *of* WISCONSIN
LA CROSSE

Fall 2020

Prof. Jason Sauppe
jsaupe@uwlax.edu
<https://cs.uwlax.edu/~jsaupe/>

Lecture 10-2a: Dimensionality Reduction

Machine Learning Taxonomy

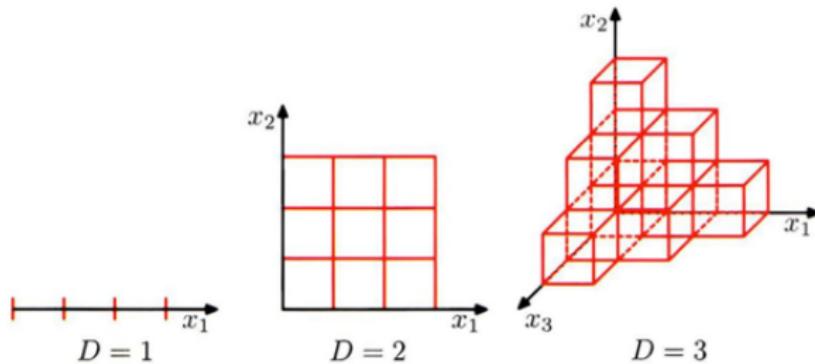


(Image source: DS 100 lecture notes)

Space is Big

Sample $\{\mathbf{x}_i\}_{i=1}^N$: N records, each with p features

- p (alternate: d, D) is the **dimensionality** of our data
- Each record \mathbf{x}_i is a point in \mathbb{R}^p



- Volume grows exponentially with p ; space becomes very sparse
- Methods relying on distance metrics do poorly

High-Dimensional Data Examples

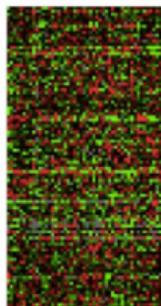


face images

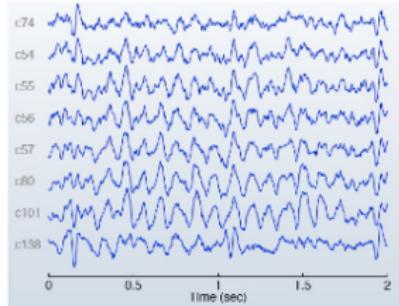
Zambian President Levy Mwanawasa has won a second term in office in an election his challenger Michael Sata accused him of rigging, official results showed on Monday.

According to media reports, a pair of hackers said on Saturday that the Firefox Web browser, commonly perceived as the safer and more customizable alternative to market leader Internet Explorer, is critically flawed. A presentation on the flaw was shown during the ToorCon hacker conference in San Diego.

documents



gene expression data



MEG readings

Dimensionality Reduction

Definition

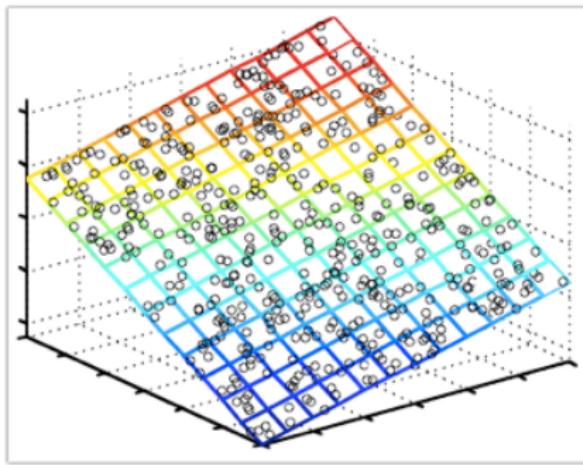
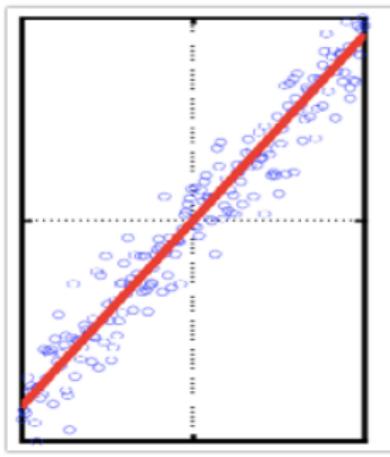
Dimensionality reduction is the process of reducing the number of features in a given data set.

Given $\{\mathbf{x}_i\}_{i=1}^N$ with p features, find a suitable **lower-dimensional** representation of the data!

Some uses:

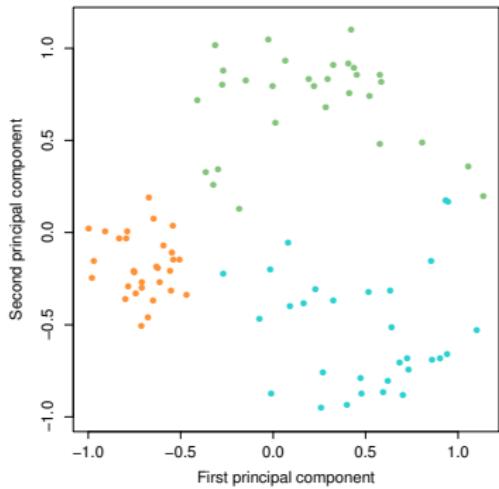
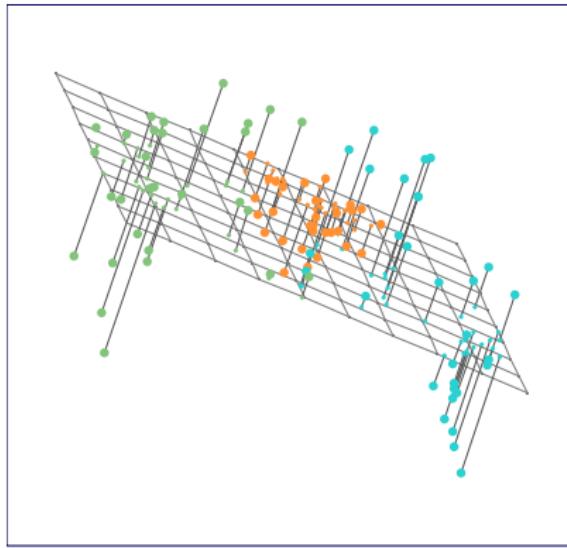
- Visualization
- Speeding up algorithms (e.g., regression, k -NN, k -Means)
- Mitigating the problem of overfitting

Intrinsic Dimensionality



- The *intrinsic* dimensionality of a data set may be lower than the number of features (e.g., multicollinearity)
- **Loss of information** from dimensionality reduction **may be small**

Reducing Dimensionality by Projection



Taken from “An Introduction to Statistical Learning, with applications in R” (Springer, 2013)
with permission from the authors: G. James, D. Witten, T. Hastie and R. Tibshirani.

Lecture 10-2b: Principal Components Analysis

Dimensionality Reduction via Principal Components Analysis

Problem: Given a data set with a large number of correlated features, find a low-dimensional representation of the data set that contains as much variation as possible.

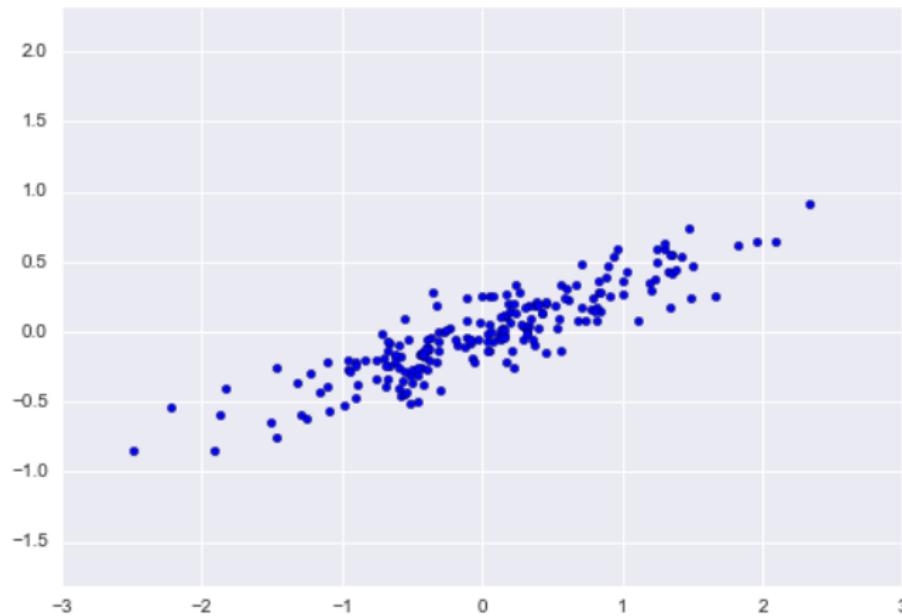
Principal Components Analysis (PCA) finds **successive orthogonal directions** that:

- ① capture the largest amount of variance in the data set
- ② are as close as possible to the data set

With sample $\{\mathbf{x}_i\}_{i=1}^N$ and p features, there are $\min(N - 1, p)$ principal components.

- Each principal component is a **linear combination** of the original features
- Components are **ordered** by amount of variance captured

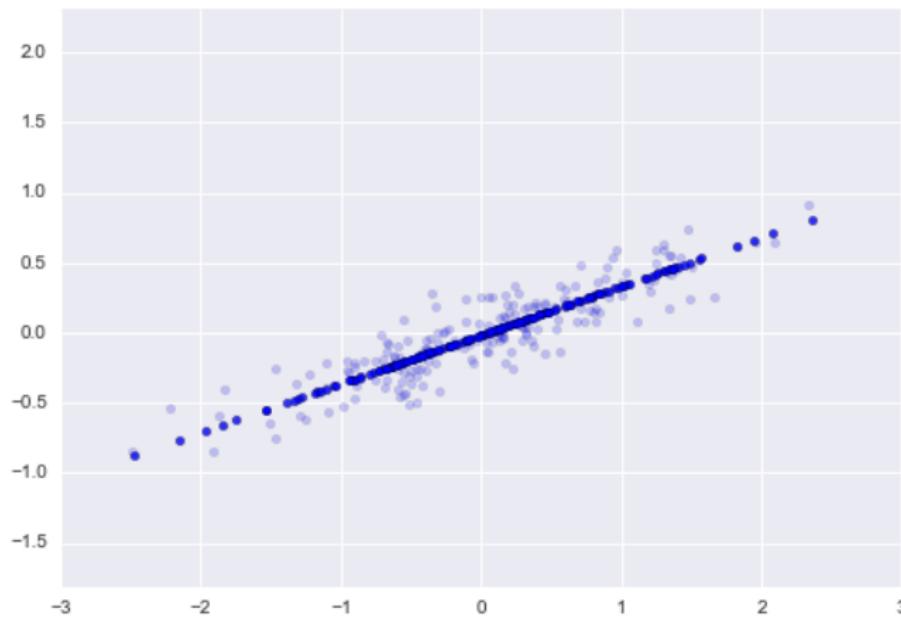
Example: Original Data Set



Example: Principal Component Directions (scaled)



Example: Dimensionality Reduction via Projection



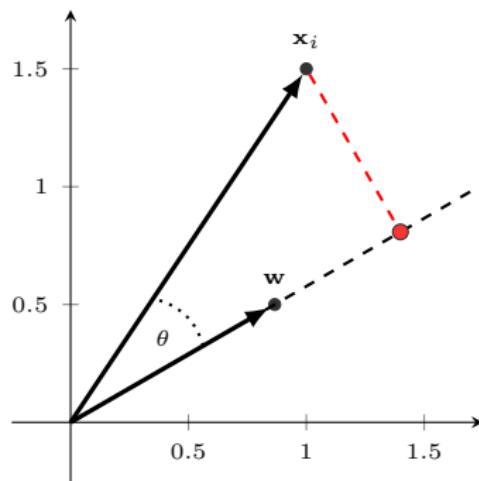
Lecture 10-2c: Finding Principal Component Directions

Finding the First Principal Component Direction

Given: A sample $\{\mathbf{x}_i\}_{i=1}^N$ with p features

Assume: Each feature is normalized to have mean 0 (and typically variance 1; e.g., using Z-scores)

Find: the unit vector $\mathbf{w} \in \mathbb{R}^p$ such that the **projection** of the sample $\{\mathbf{x}_i\}_{i=1}^N$ onto \mathbf{w} has maximum variance.

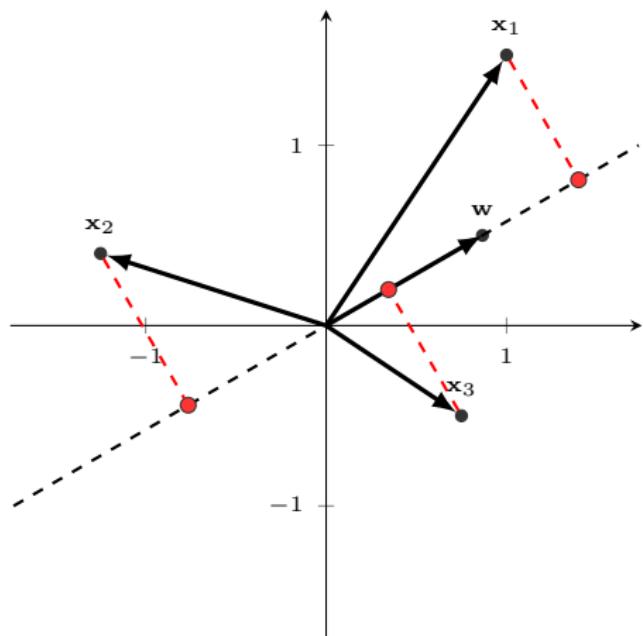


For \mathbf{x}_i , its projection onto \mathbf{w} is \bullet , and is given by $(\mathbf{x}_i \cdot \mathbf{w}) \mathbf{w}$, or $(\mathbf{x}_i^T \mathbf{w}) \mathbf{w}$.

- Length: $\mathbf{x}_i \cdot \mathbf{w} = \|\mathbf{x}_i\| \|\mathbf{w}\| \cos(\theta)$ (scalar)
- Direction: \mathbf{w}

Variance of Projected Points

The projection of the sample $\{\mathbf{x}_i\}_{i=1}^N$ onto \mathbf{w} yields points that all lie along the **one-dimensional** line given by \mathbf{w} !



In the one-dimensional subspace, the sample mean of projected points is:

$$\begin{aligned}\bar{x} &= \frac{1}{N} \sum_{i=1}^N (\mathbf{x}_i^T \mathbf{w}) \\ &= \mathbf{w}^T \left(\frac{1}{N} \sum_{i=1}^N \mathbf{x}_i \right) \\ &= \mathbf{w}^T \mathbf{0} = 0\end{aligned}$$

and the sample variance is:

$$\begin{aligned}s^2 &= \frac{1}{N-1} \sum_{i=1}^N (\mathbf{x}_i^T \mathbf{w} - \bar{x})^2 \\ &= \frac{1}{N-1} \sum_{i=1}^N (\mathbf{x}_i^T \mathbf{w})^2.\end{aligned}$$

An Optimization Problem for Finding the First PC Direction

Vector form:

$$\max \sum_{i=1}^N (\mathbf{x}_i^T \mathbf{w})^2$$

$$\text{s.t. } \|\mathbf{w}\| = 1$$

$$\mathbf{w} \in \mathbb{R}^p.$$

Non-vector form:

$$\max \sum_{i=1}^N \left(\sum_{j=1}^p \mathbf{x}_{ij} w_j \right)^2$$

$$\text{s.t. } \sum_{j=1}^p w_j^2 = 1$$

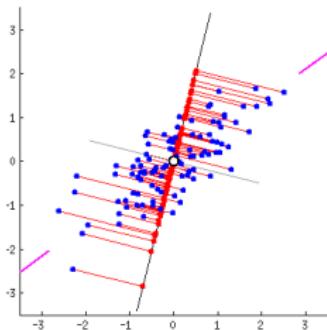
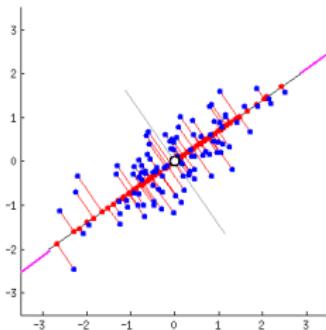
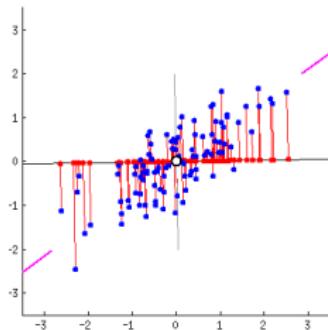
$$w_j \in \mathbb{R}$$

$$\forall j \in \{1, 2, \dots, p\}$$

The optimal solution \mathbf{w}^* defines the **principal component direction (loading vector)**, denoted ϕ_1 , for the **first principal component**.

- The first principal component, Z_1 , is the linear combination of the features: $Z_1 = \phi_{1,1}X_1 + \phi_{1,2}X_2 + \dots + \phi_{1,p}X_p$. This is a **new** feature with maximum variance in the data set.

Cool Visualization



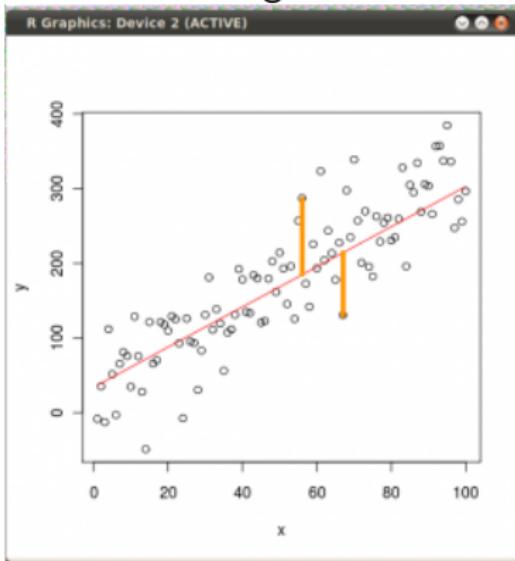
<https://stats.stackexchange.com/a/140579>

The first PC direction **simultaneously**

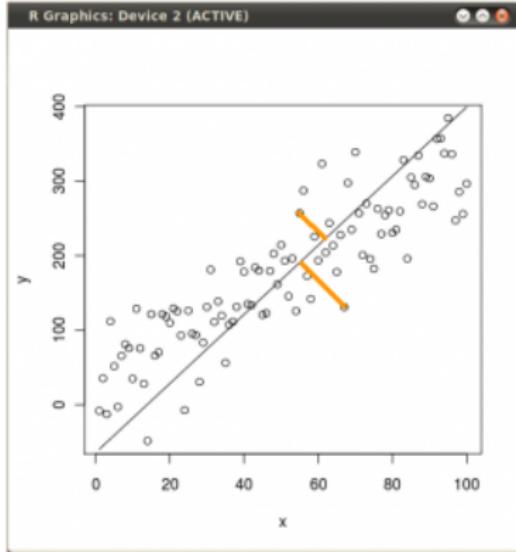
- Maximizes variance in the projected points
- Minimizes errors (squared Euclidean distance) in the projection

PCA vs. Regression

Linear regression:



First PC direction:



<https://stats.stackexchange.com/a/2700>

Finding the Second PC Direction

The PC direction for the **second principal component**, ϕ_2 , is the unit vector such that

- ϕ_2 is **orthogonal** to ϕ_1
- the projection of the sample onto ϕ_2 has maximum variance (subject to the above constraint)

We can find ϕ_2 by solving

$$\begin{aligned} \max & \sum_{i=1}^N (\mathbf{x}_i^T \mathbf{w})^2 \\ \text{s.t. } & \|\mathbf{w}\| = 1 \\ & \mathbf{w}^T \phi_1 = 0 \\ & \mathbf{w} \in \mathbb{R}^p. \end{aligned}$$

Finding the k th PC Direction

More generally, the k th PC direction ϕ_k can be found by solving

$$\begin{aligned} & \max \sum_{i=1}^N (\mathbf{x}_i^T \mathbf{w})^2 \\ \text{s.t. } & \|\mathbf{w}\| = 1 \\ & \mathbf{w}^T \phi_j = 0 \quad \forall j \in \{1, 2, \dots, k-1\} \\ & \mathbf{w} \in \mathbb{R}^p. \end{aligned}$$

These are **non-trivial** optimization problems, but linear algebra can make things much easier!

Lecture 10-2d: Linear Algebra for PCA

Revisiting the Objective Function

With some linear algebra and rewriting, we have

$$\sum_{i=1}^N (\mathbf{x}_i^T \mathbf{w})^2 = \sum_{i=1}^N (\mathbf{w}^T \mathbf{x}_i) (\mathbf{x}_i^T \mathbf{w}) = \mathbf{w}^T \left(\sum_{i=1}^N \mathbf{x}_i \mathbf{x}_i^T \right) \mathbf{w} = \mathbf{w}^T \mathbf{X}^T \mathbf{X} \mathbf{w}.$$

- $\mathbf{X} = \begin{bmatrix} -\mathbf{x}_1^T- \\ -\mathbf{x}_2^T- \\ \vdots \\ -\mathbf{x}_N^T- \end{bmatrix}$ is the $N \times p$ **data matrix** for the **centered** data
- $\mathbf{X}^T \mathbf{X}$ is a $p \times p$ matrix that is proportional to the **sample covariance matrix** of the **centered** data

The principal component directions $\phi_1, \phi_2, \dots, \phi_p$ are **intricately connected** to the structure of the matrix $\mathbf{X}^T \mathbf{X}$!

Eigenvectors of a Matrix

The objective function for finding PC directions is given by $\mathbf{w}^T \mathbf{X}^T \mathbf{X} \mathbf{w}$.

- $\mathbf{X}^T \mathbf{X}$ is a $p \times p$ matrix
- $(\mathbf{X}^T \mathbf{X}) \mathbf{w}$ is a $p \times 1$ vector
- $\mathbf{w}^T ((\mathbf{X}^T \mathbf{X}) \mathbf{w})$ is the dot product of two $p \times 1$ vectors, \mathbf{w} and the result of transforming \mathbf{w} by $\mathbf{X}^T \mathbf{X}$
- The dot product of two vectors is maximized when both vectors point in the **same direction**
- That means we want $(\mathbf{X}^T \mathbf{X}) \mathbf{w} = \lambda \mathbf{w}$ for some scalar λ
- For an $m \times m$ matrix \mathbf{A} and an $m \times 1$ vector \mathbf{v} , if $\mathbf{Av} = \lambda \mathbf{v}$, then \mathbf{v} is an **eigenvector** of \mathbf{A} , with associated **eigenvalue** λ
- Hence, the directions \mathbf{w} that maximize $\mathbf{w}^T ((\mathbf{X}^T \mathbf{X}) \mathbf{w})$ are the **eigenvectors** of $\mathbf{X}^T \mathbf{X}$!
- Because \mathbf{w} is constrained to be a unit vector, the **best** eigenvector (i.e., one with the highest objective) is the one with the largest eigenvalue

Some Linear Algebra: Eigendecomposition

The Eigendecomposition of matrix $\mathbf{X}^T \mathbf{X}$ is given by $\mathbf{Q} \Lambda \mathbf{Q}^T$, where

$$\mathbf{Q} = \begin{bmatrix} | & | & & | \\ \mathbf{q}_1 & \mathbf{q}_2 & \dots & \mathbf{q}_p \\ | & | & & | \end{bmatrix} \text{ and } \Lambda = \begin{bmatrix} \lambda_1 & 0 & \dots & 0 \\ 0 & \lambda_2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \lambda_p \end{bmatrix}.$$

- \mathbf{Q} is a $p \times p$ orthogonal matrix, where column \mathbf{q}_k is the k th **eigenvector** of $\mathbf{X}^T \mathbf{X}$
- Λ is a $p \times p$ diagonal matrix with entries λ_k corresponding to the k th **eigenvalue** of $\mathbf{X}^T \mathbf{X}$, sorted with $\lambda_1 \geq \lambda_2 \geq \dots \geq \lambda_p$

Proposition

For a sample $\{\mathbf{x}_i\}_{i=1}^N$ with data matrix \mathbf{X} and associated Eigendecomposition $\mathbf{X}^T \mathbf{X} = \mathbf{Q} \Lambda \mathbf{Q}^T$, the k th principal component direction is given by the eigenvector \mathbf{q}_k , with eigenvalue λ_k proportional to the amount of variance captured.

Lecture 10-2e: Data Transformations via PCA

Transforming Data into Principal Components

For any data point \mathbf{x}_i , we construct **principal component scores** (i.e., **values for the new features**) by projecting \mathbf{x}_i onto the PC directions:

- First principal component score is $z_{i1} = \mathbf{x}_i^T \boldsymbol{\phi}_1$
- Second principal component score is $z_{i2} = \mathbf{x}_i^T \boldsymbol{\phi}_2$
- p 'th principal component score is $z_{ip} = \mathbf{x}_i^T \boldsymbol{\phi}_p$

We can combine these scores into a data point in a transformed space:

$$\mathbf{z}_i = \begin{pmatrix} z_{i1} \\ z_{i2} \\ \vdots \\ z_{ip} \end{pmatrix} = \begin{pmatrix} \mathbf{x}_i^T \boldsymbol{\phi}_1 \\ \mathbf{x}_i^T \boldsymbol{\phi}_2 \\ \vdots \\ \mathbf{x}_i^T \boldsymbol{\phi}_p \end{pmatrix} = \begin{pmatrix} \boldsymbol{\phi}_1^T \mathbf{x}_i \\ \boldsymbol{\phi}_2^T \mathbf{x}_i \\ \vdots \\ \boldsymbol{\phi}_p^T \mathbf{x}_i \end{pmatrix} = \mathbf{Q}^T \mathbf{x}_i$$

This is a **representation** of \mathbf{x}_i in terms of the **new features** or principal components. **No information is lost here**, because \mathbf{x}_i can be fully recovered from \mathbf{z}_i as $\mathbf{x}_i = \mathbf{Q}\mathbf{z}_i = \mathbf{QQ}^T \mathbf{x}_i = \mathbf{I}\mathbf{x}_i = \mathbf{x}_i$.

Dimensionality Reduction via PCA

Using **all** principal components in our transformation just gives us a **change of basis** for our original data. To actually **reduce** the dimensionality, we can **drop** some of the **new features**!

Full transformation:

$$\mathbf{z}_i = \begin{pmatrix} \phi_1^T \mathbf{x}_i \\ \phi_2^T \mathbf{x}_i \\ \vdots \\ \phi_k^T \mathbf{x}_i \\ \vdots \\ \phi_p^T \mathbf{x}_i \end{pmatrix}$$

Truncated transformation, with $k < p$:

$$\mathbf{z}_i^{(k)} = \begin{pmatrix} \phi_1^T \mathbf{x}_i \\ \phi_2^T \mathbf{x}_i \\ \vdots \\ \phi_k^T \mathbf{x}_i \end{pmatrix}$$

- Dropping features entails a **loss of information**
- By construction, **new features** are ordered from most to least variance, so keeping first k new features retains **as much information** as possible

Transformation via Matrix Multiplication

The original and transformed data points:

$$\mathbf{x}_i = \begin{pmatrix} x_{i1} \\ x_{i2} \\ \vdots \\ x_{ip} \end{pmatrix}, \quad \mathbf{z}_i = \begin{pmatrix} z_{i1} \\ z_{i2} \\ \vdots \\ z_{ip} \end{pmatrix} = \begin{pmatrix} \mathbf{x}_i^T \phi_1 \\ \mathbf{x}_i^T \phi_2 \\ \vdots \\ \mathbf{x}_i^T \phi_p \end{pmatrix} = \begin{pmatrix} \phi_1^T \mathbf{x}_i \\ \phi_2^T \mathbf{x}_i \\ \vdots \\ \phi_p^T \mathbf{x}_i \end{pmatrix} = \mathbf{Q}^T \mathbf{x}_i, \quad \mathbf{z}_i^T = (\mathbf{Q}^T \mathbf{x}_i)^T = \mathbf{x}_i^T \mathbf{Q}$$

The original and transformed data matrices, plus the eigenvector matrix:

$$\mathbf{X} = \begin{bmatrix} -\mathbf{x}_1^T- \\ -\mathbf{x}_2^T- \\ \vdots \\ -\mathbf{x}_N^T- \end{bmatrix}, \quad \mathbf{Q} = \begin{bmatrix} | & & | \\ \phi_1 & \dots & \phi_p \\ | & & | \end{bmatrix}, \quad \mathbf{Z} = \begin{bmatrix} -\mathbf{z}_1^T- \\ -\mathbf{z}_2^T- \\ \vdots \\ -\mathbf{z}_N^T- \end{bmatrix} = \begin{bmatrix} -\mathbf{x}_1^T \mathbf{Q}- \\ -\mathbf{x}_2^T \mathbf{Q}- \\ \vdots \\ -\mathbf{x}_N^T \mathbf{Q}- \end{bmatrix} = \mathbf{XQ}$$

For any $k \in \{1, 2, \dots, p\}$, we can transform the data using:

$$\mathbf{Q}^{(k)} = \begin{bmatrix} | & & | \\ \phi_1 & \dots & \phi_k \\ | & & | \end{bmatrix}, \quad \mathbf{Z}^{(k)} = \mathbf{XQ}^{(k)} = \begin{bmatrix} -\mathbf{x}_1^T \mathbf{Q}^{(k)}- \\ -\mathbf{x}_2^T \mathbf{Q}^{(k)}- \\ \vdots \\ -\mathbf{x}_N^T \mathbf{Q}^{(k)}- \end{bmatrix} = \begin{bmatrix} z_{1,1} & \dots & z_{1,k} \\ z_{2,1} & \dots & z_{2,k} \\ \vdots & \ddots & \vdots \\ z_{N,1} & \dots & z_{N,k} \end{bmatrix}$$

Lecture 10-2f: Additional Notes on PCA

PCA Algorithm Summary

Dimensionality Reduction via PCA:

- ① Center the data: for each feature (column), subtract the mean value from the associated entries
- ② (Typically) scale each dimension by its variance (e.g., use a Z-score transformation as part of step 1)
(helps to pay less attention to magnitude of dimensions)
- ③ Find the k eigenvectors of $\mathbf{X}^T \mathbf{X}$ with the largest eigenvalues
- ④ Use these k eigenvectors as the first k principal component directions
- ⑤ To transform original data points to use new features, compute
$$\mathbf{z}_i^{(k)} = \mathbf{Q}^{(k)} \mathbf{x}_i \quad (\text{or all at once: } \mathbf{Z}^{(k)} = \mathbf{X} \mathbf{Q}^{(k)})$$

An Alternative Decomposition

One way to find the principal component directions is by computing the Eigendecomposition of $\mathbf{X}^T \mathbf{X}$.

Another way to find these directions is by computing the **singular value decomposition** of the (centered) data matrix \mathbf{X} .

$$\text{SVD of } \mathbf{X} \text{ yields } \mathbf{X} = \mathbf{U}\Sigma\mathbf{V}^T$$

Properties of SVD:

- Dimensions: $\mathbf{X}: N \times p$, $\mathbf{U}: N \times N$, $\Sigma: N \times p$, $\mathbf{V}: p \times p$
- \mathbf{U} , \mathbf{V} are orthogonal matrices
- Σ is a rectangular diagonal matrix containing the **singular values** of \mathbf{X} , ordered from greatest to least
- Columns of \mathbf{U} are **left singular vectors**
- Columns of \mathbf{V} are **right singular vectors**

Relationship of SVD to PCA

We have now seen **two** matrix decomposition techniques:

- Eigendecomposition: $\mathbf{X}^T \mathbf{X} = \mathbf{Q} \Lambda \mathbf{Q}^T$
PC directions are columns of \mathbf{Q}
- SVD: $\mathbf{X} = \mathbf{U} \Sigma \mathbf{V}^T$
PC directions are columns of \mathbf{V} (equivalently, rows of \mathbf{V}^T)

Consequences:

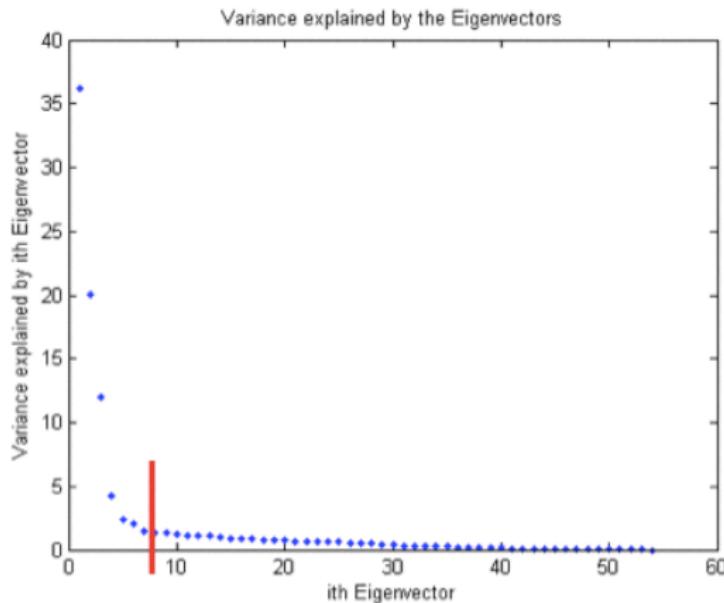
- $\mathbf{Q} = \mathbf{V}$ (i.e., the columns of \mathbf{V} are the eigenvectors of $\mathbf{X}^T \mathbf{X}$)
- $\mathbf{Z} = \mathbf{XQ} = \mathbf{XV} = \mathbf{U}\Sigma$, so $\mathbf{U}\Sigma$ contains the transformed data
- k th singular value σ_k of Σ is proportional to k th eigenvalue λ_k of Λ , both of which indicate the amount of variance captured by ϕ_k
- SVD applied to \mathbf{X} tends to be **more numerically stable** than Eigendecomposition applied to $\mathbf{X}^T \mathbf{X}$

See [here](#), [here](#), [here](#), and [here](#) for more info.

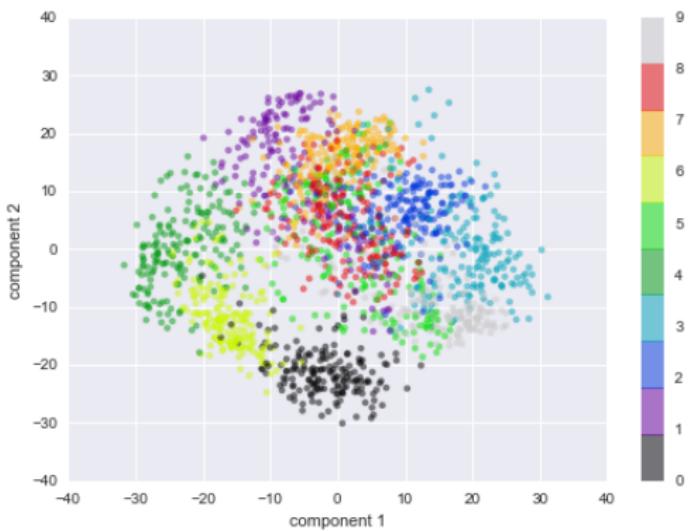
How Many Principal Components?

Use first two principal components for plotting!

For other applications, use “elbow method” with **Skree plot**:



PCA on Digits Data

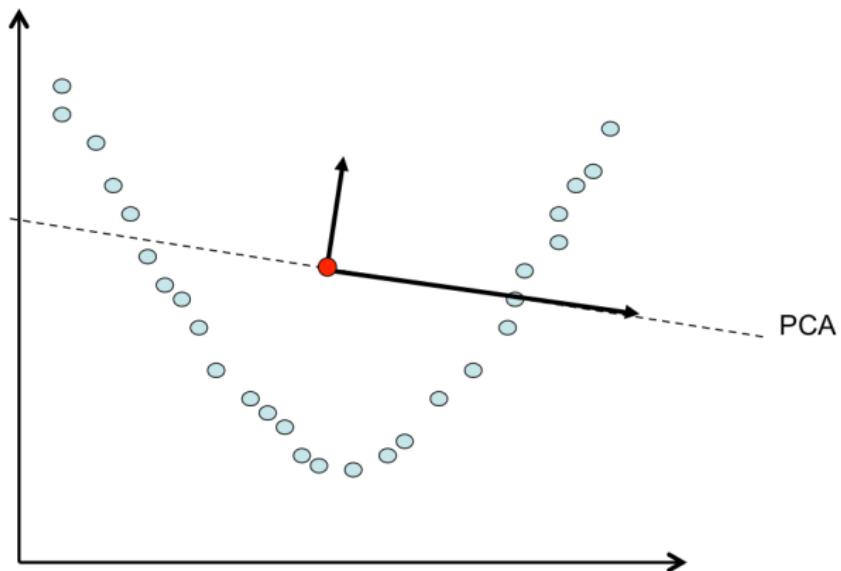


<https://jakevdp.github.io/PythonDataScienceHandbook/05.08-random-forests.html>

<https://jakevdp.github.io/PythonDataScienceHandbook/05.09-principal-component-analysis.html>

Lecture 10-2g: Other Dimensionality Reduction Methods

Limitations of PCA



PCA doesn't handle non-ellipsoidal data very well...

Dimensionality Reduction Methods

Linear Methods:

- Principal Components Analysis (PCA)
- Linear Discriminant Analysis (LDA)

Nonlinear Methods (manifold learning):

- Isomap
- Multidimensional Scaling (MDS)
- Locally Linear Embeddings (LLE)
- t -distributed stochastic neighbor embedding (t -SNE)
- Uniform manifold approximation and projection (UMAP)

Definition

A **manifold** is a lower-dimensional surface embedded in a higher-dimensional space (e.g., a sheet of paper contorted in 3D).

For Further Reading

Dimensionality reduction:

- https://en.wikipedia.org/wiki/Dimensionality_reduction
- https://en.wikipedia.org/wiki/Nonlinear_dimensionality_reduction
- <https://machinelearningmastery.com/dimensionality-reduction-for-machine-learning/>
- <https://towardsdatascience.com/dimensionality-reduction-for-machine-learning-80a46c2ebb7e>

PCA:

- A Tutorial on Principal Component Analysis
- Lecture notes on PCA from UC Berkeley
- <https://jakevdp.github.io/PythonDataScienceHandbook/05.09-principal-component-analysis.html>
- <https://machinelearningmastery.com/calculate-principal-component-analysis-scratch-python/>
- <https://machinelearningmastery.com/introduction-to-matrix-decompositions-for-machine-learning/>
- https://en.wikipedia.org/wiki/Principal_component_analysis
- <https://stats.stackexchange.com/questions/2691/making-sense-of-principal-component-analysis-eigenvectors-eigenvalues>

Manifold learning:

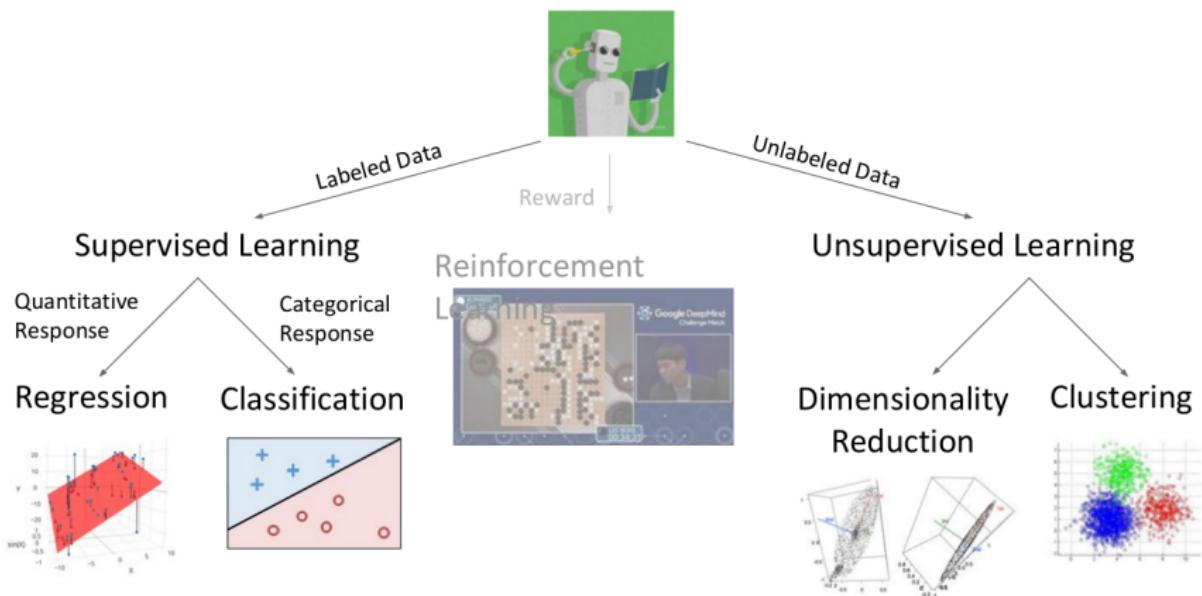
- <https://jakevdp.github.io/PythonDataScienceHandbook/05.10-manifold-learning.html>
- <https://scikit-learn.org/stable/modules/manifold.html>
- https://scikit-learn.org/stable/auto_examples/manifold/plot_compare_methods.html

CS 457/557: Machine Learning

Lecture 10-3: Machine Learning Wrap-Up

Lecture 10-3a: Machine Learning Wrap-Up

Core Machine Learning Taxonomy



(Image source: DS 100 lecture notes)

Topics Covered

A **noncomprehensive** list of topics covered:

- Supervised learning
 - Concepts
 - Prediction types: regression vs. classification
 - Components of ML system
 - Optimization via gradient descent
 - Modeling: Bias-Variance trade-off
 - Validation and Cross-Validation
 - Regularization
 - Feature engineering
 - Methods
 - Decision trees
 - Linear and logistic regression
 - k -Nearest Neighbors
 - Support vector machines
 - Neural networks

Topics Covered (continued)

- Reinforcement learning
 - Markov decision processes
 - Notation: $M = (S, A, P, R, T), \gamma, \pi, V^\pi, Q^\pi$
 - Bellman equations, policy evaluation, & policy iteration
 - Bellman optimality equations and value iteration
 - Temporal difference learning
 - TD updates from one-step experiences
 - Greedy and ϵ -greedy policies
 - SARSA and Q-Learning algorithms
 - Generalization with features
 - TD updates for feature weights
 - Finding features via coarse coding, prototypes
 - Deep reinforcement learning
- Unsupervised learning
 - Clustering
 - Dimensionality reduction

Comic of the Day



<http://xkcd.com/1838/>

An Incomplete List of Uncovered Topics

- Supervised learning
 - Methods: Naive Bayes, Discriminant Analysis
 - Ensemble Methods (e.g., bagging, boosting)
 - Deep neural networks (including CNN, RNN architectures)
 - Advanced optimization algorithms (e.g., Adagrad, Adam)
 - Learning Theory (PAC Learning)
 - Applications: NLP, Vision, Robotics
- Reinforcement learning
 - Monte Carlo methods
 - Policy gradient methods
 - Apprenticeship and Inverse Reinforcement Learning
 - Lots of extensions to basic ideas
- Unsupervised learning
 - Lots of stuff!

For Further Reading

Machine Learning:

- *The Hundred-Page Machine Learning Book* by Burkov
- *Machine Learning Engineering* by Burkov
- *Reinforcement Learning* by Sutton and Barto
- *Machine Learning: An Applied Mathematics Introduction* by Wilmott

Deep Learning:

- *Deep Learning* (The MIT Press Essential Knowledge Series) by Kelleher
- *Deep Learning* by Goodfellow et al.
- *Dive into Deep Learning* by Zhang et al.
- Deep Learning Specialization course on Coursera

Statistical Learning & Predictive Modeling:

- *An Introduction to Statistical Learning* by James et al.
- *Applied Predictive Modeling* by Kuhn and Johnson
- *The Elements of Statistical Learning* by Hastie et al.

Additional Classes

At UWL:

- CS 452/552: Artificial Intelligence
- CS 461/561: Introduction to Data Science
- MTH 309: Linear Algebra and Differential Equations
- MTH 371: Numerical Methods
- STAT 245: Probability and Statistics

Elsewhere:

- CS 229: Machine Learning at Stanford
- CS 230: Deep Learning at Stanford
- CS 189: Introduction to Machine Learning at UC-Berkeley
- CS 285: Deep Reinforcement Learning at UC-Berkeley