Concise Machine Learning

Jonathan Richard Shewchuk
August 14, 2016

Department of Electrical Engineering and Computer Sciences
University of California at Berkeley
Berkeley, California 94720

Abstract

This report contains lecture notes for UC Berkeley's introductory class on Machine Learning. It covers many methods for classification and regression, and several methods for clustering and dimensionality reduction. It is concise because not a word is included that cannot be written or spoken in a single semester's lectures (with whiteboard lectures and almost no Powerpoint slides!) and because the choice of topics is limited to a small selection of particularly useful, popular algorithms.

Supported in part by the National Science Foundation under Award CCF-1423560, in part by the University of California Lab Fees Research Program, and in part by an Alfred P. Sloan Research Fellowship. The claims in this document are those of the author. They are not endorsed by the sponsors or the U.S. Government.

Keywords: machine learning, classification, regression, dimensionality reduction, clustering, perceptrons, support vector machines (SVMs), Gaussian discriminant analysis, linear discriminant analysis (LDA), quadratic discriminant analysis (QDA), logistic regression, decision trees, neural networks, convolutional neural networks (CNNs, ConvNets), nearest neighbor search, least-squares linear regression, logistic regression, polynomial regression, ridge regression, Lasso, principal components analysis (PCA), latent factor analysis, latent semantic indexing, k-means clustering, hierarchical clustering, spectral graph clustering

Lectures

About this Report

This report compiles my lectures notes for UC Berkeley's class CS 189/289A, *Machine Learning*, which is both an undergraduate and introductory graduate course. I hope it will serve as a fast introduction to the subject for readers who are already comfortable with vector calculus, linear algebra, probability, and statistics. Please consult my CS 189/289A web page[1] as an addendum to this report; it includes an extended description of each lecture and additional web links and reading assignments related to the lectures. Consider this report and the web page to be living documents; both will be refined a bit every time I teach the class.

The term "lecture notes" has shifted to include long textbook-style treatments written by professors as supplements to their classes. Not so here. This report compiles the actual notes that I lecture from. I call it *Concise Machine Learning* because I do not include a single word that I do not have time to write or speak during one fourteen-week semester of twice-weekly 80-minute lectures. Words that appear [in brackets] are spoken; everything else is written on the "whiteboard"—in my class, a tablet computer. My whiteboard software permits me to incorporate (and write on) figures, included here. However, I am largely anti-Powerpoint and I resort to prepared slides for just three or four brief segments during the semester.

These notes might be ideal for mathematically sophisticated readers who want to learn the basics of machine learning as quickly as possible. But they're not ideal for everybody. The time limitation necessitates that many details are omitted. I assume that the smartest, most mathematically well-prepared readers will be able to fill in those details themselves. But many readers, including most students who take the class, will want additional readings or discussion sections for greater detail. My class web page lists additional readings for most of the lectures, many of them from two textbooks that have been kindly made available for free on the web: *An Introduction to Statistical Learning with Applications in R*,[2] by Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani, Springer, New York, 2013, ISBN # 978-1-4614-7137-0; and *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*,[3] second edition, by Trevor Hastie, Robert Tibshirani, and Jerome Friedman, Springer, New York, 2008. Wikipedia also has excellent introductions to many machine learning algorithms. Readers wanting the verbose kind of "lecture notes" should consider the fine ones written by Stanford University's Andrew Ng.[4] That web site also includes good primers on linear algebra and probability that can serve as useful reviews before reading this report. I have no interest in duplicating these efforts; instead, I'm aiming for the neglected niche of "shortest introduction." (And perhaps also "best stolen illustrations.")

The other thing that makes this report concise is the choice of topics. CS 189/289A was introduced at UC Berkeley in the spring of 2013 by Prof. Jitendra Malik, and most of his topic choices remain intact here. Jitendra told me that he only taught a machine learning algorithm if he or his collaborators had used it successfully for some application. He told me, "the machine learning course is too important to leave to the machine learning experts"—that is, users of machine learning algorithms often have a more clear-eyed view of their usefulness than inventors of machine learning algorithms.

I thank Peter Bartlett, Alyosha Efros, Isabelle Guyon, and Jitendra Malik—the previous teachers of CS 189/289A—for their lectures and lecture notes, from which I learned the topic myself. While I've given the lectures my own twist and rearranged the material a lot, I am ultimately making incremental improvements (and perhaps incremental worsenings) to a structure they handed down to me. I also thank Carlos Flores for sharing screenshots from my lectures.

[1] http://www.cs.berkeley.edu/~jrs/189/

[2] http://www-bcf.usc.edu/~gareth/ISL/

[3] http://statweb.stanford.edu/~tibs/ElemStatLearn/

[4] http://cs229.stanford.edu/materials.html

1 Introduction

CS 189 / 289A
Machine Learning
Jonathan Shewchuk

Class website: http://www.cs.berkeley.edu/~jrs/189

TAs: Tuomas Haarnoja, Aldo Pacchiano, Rohan Chitnis, Shaun Singh, Marvin Zhang, Brian Hou

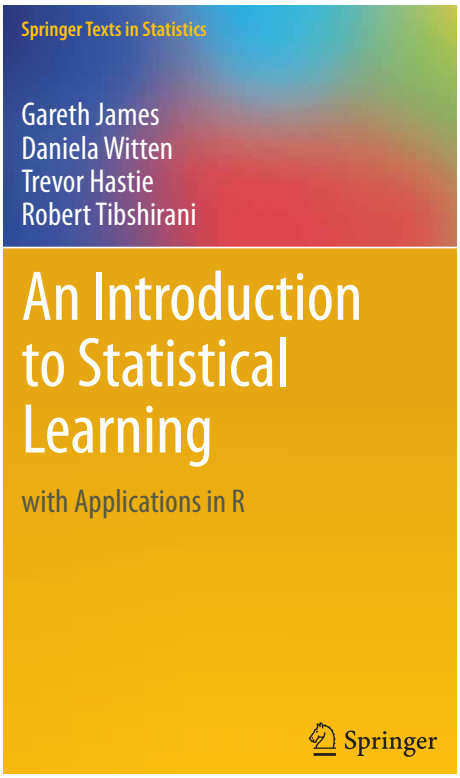
Discussion sections: [Spring 2016]
Go to schedule.berkeley.edu and look up CS 189 (NOT 289A).
You choose your section. If the room is too full, please go to another one.
Sections 102 and 103 are cancelled.
No sections this week.

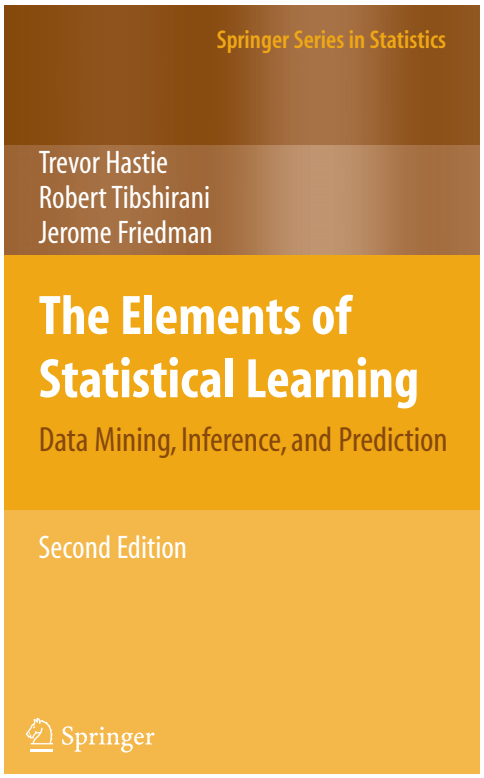
Questions: Please use Piazza, not email. [Piazza has an option for private questions, but please use public for most questions so other people can benefit.]
For personal matters only, jrs@cory.eecs.berkeley.edu

[Enrollment: More than 100 of you were admitted around noon today. Most non-CS grad students won't be admitted.]

[For those of you taking CS 162, I have good news: CS 162 has been moved to exam group 3. There is no longer a final exam conflict between 189 and 162.]

[Textbooks: Available free online. Linked from class web page.]





Prerequisites

Math 53 (vector calculus)
Math 54 or 110 (linear algebra)
CS 70 (discrete math; probability)
CS 188 (AI; probability; decision theory)

Grading: 189

40% 7 Homeworks. Late policy: 5 slip days total.
20% Midterm: Wednesday, March 16, in class (6:30–8 pm)
20% Final Exam: Friday, May 13, 3–6 PM (Exam group 19)

Grading: 289A

40% HW
20% Midterm
20% Final Exam
20% Project

Cheating

- Discussion of HW problems is encouraged.
- All homeworks, including programming, must be written individually.
- We will actively check for plagiarism.
- Typical penalty is a large NEGATIVE score, but I reserve right to give an instant F for even one violation, and will always give an F for two.

[Last time I taught CS 61B, we had to punish roughly 100 people for cheating. It was very painful. Please don't put me through that again.]

CORE MATERIAL

- Finding patterns in data; using them to make predictions.
- Models and statistics help us understand patterns.
- Optimization algorithms "learn" the patterns.

[The most important part of this is the data. Data drives everything else.
You cannot learn much if you don't have enough data.
You cannot learn much if your data sucks.
But it's amazing what you can do if you have lots of good data.
Machine learning has changed a lot in the last decade because the internet has made truly vast quantities of data available. For instance, with a little patience you can download tens of millions of photographs. Then you can build a 3D model of Paris.
Some techniques that had fallen out of favor, like neural nets, have come back big in the last few years because researchers found that they work so much better when you have vast quantities of data.]

EXAMPLE: CLASSIFYING DIGITS

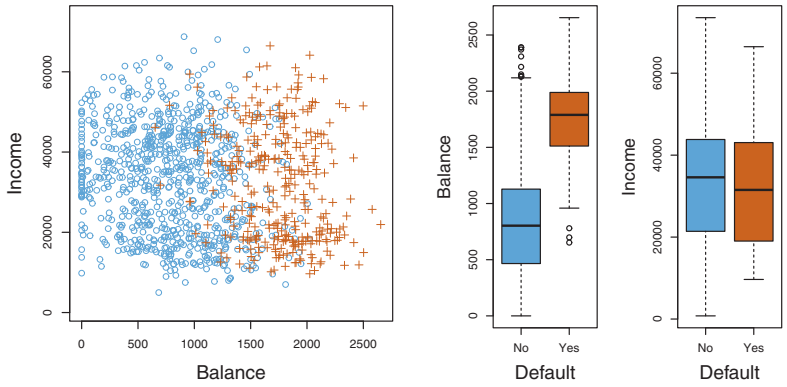
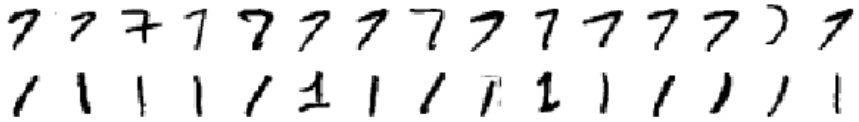


FIGURE 4.1. *The* `Default` *data set.* Left: *The annual incomes and monthly credit card balances of a number of individuals. The individuals who defaulted on their credit card payments are shown in orange, and those who did not are shown in blue.* Center: *Boxplots of* `balance` *as a function of* `default` *status.* Right: *Boxplots of* `income` *as a function of* `default` *status.*

creditcards.pdf (ISL, Figure 4.1) [The problem of classification. We are given data points, each belonging to one of two classes. Then we are given additional points whose class is unknown, and we are asked to predict what class each new point is in. Given the credit card balance and annual income of a cardholder, predict whether they will default on their debt.]

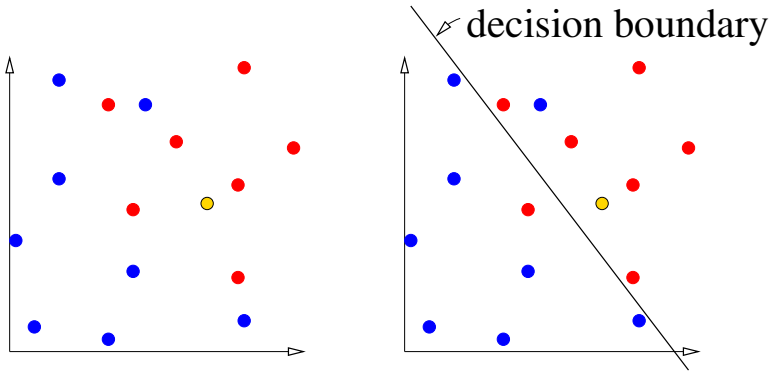


sevensones.pdf [In this simplified digit recognition problem, we are given handwritten 7's and 1's, and we are asked to learn to distinguish the 7's from the 1's.]

– Collect training images: e.g. 7's and digits that are not 7's
– Express these images as vectors

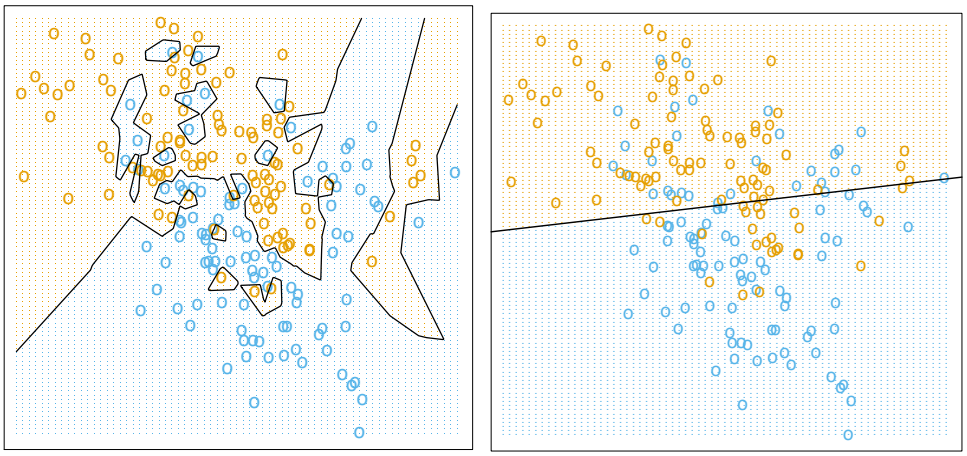
$$\begin{array}{|c|c|c|c|} \hline 3 & 3 & 3 & 3 \\ \hline 0 & 0 & 2 & 3 \\ \hline 0 & 0 & 1 & 3 \\ \hline 3 & 3 & 3 & 3 \\ \hline \end{array} \rightarrow \begin{bmatrix} 3 \\ 3 \\ 3 \\ 3 \\ 0 \\ 0 \\ 2 \\ 3 \\ 0 \\ 0 \\ 1 \\ 3 \\ 3 \\ 3 \\ 3 \\ 3 \end{bmatrix}$$

[I can't draw 16-dimensional space, so let's pretend it's 2-dimensional.]

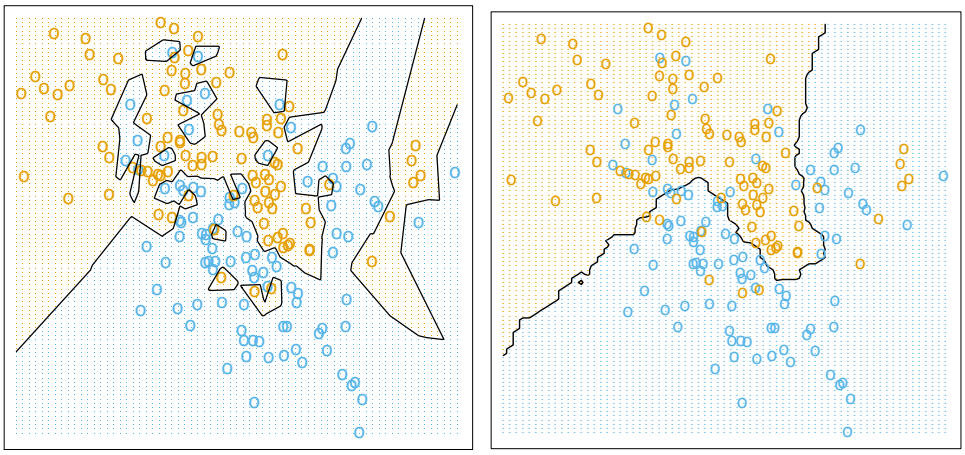


[Draw this figure by hand. classify.pdf]
[Draw 2 colors of dots, almost but not quite linearly separable.]
["How do we classify a new point?" Draw a point in a third color.]
[One possibility: look at its nearest neighbor.]
[Another possibility: draw a linear decision boundary; label it.]

[We'll learn some ways to draw these linear decision boundaries in the next several lectures. But for now, let's compare this method with another method.]



classnear.pdf, classlinear.pdf (ESL, Figures 2.3 & 2.1) [Here are two examples of classifiers for the same data. At left we have a nearest neighbor classifier, which classifies a point by finding the nearest point in the input data, and assigning it the same class. At right we have a linear classifier, which guesses that everything above the line is brown, and everything below the line is blue. The decision boundaries are in black.]



classnear.pdf, classnear15.pdf (ESL, Figures 2.3 & 2.2) [At right we have a 15-nearest neighbor classifier. Instead of looking at the nearest neighbor of a new point, it looks at the 15 nearest neighbors and lets them vote for the correct class. The 1-nearest neighbor classifier at left has a big advantage: it classifies all the training data correctly, whereas the 15-nearest neighbor classifier at right figure does not. But the right figure has an advantage too. Somebody please tell me what.]

[The left figure is an example of what's called overfitting. In the left figure, observe how intricate the decision boundary is that separates the positive examples from the negative examples. It's a bit too intricate to reflect reality. In the right figure, the decision boundary is smoother. Intuitively, that smoothness is probably more likely to correspond to reality.]

Validation

- Train a classifier: it learns to distinguish 7 from not 7
- Test the classifier on NEW images

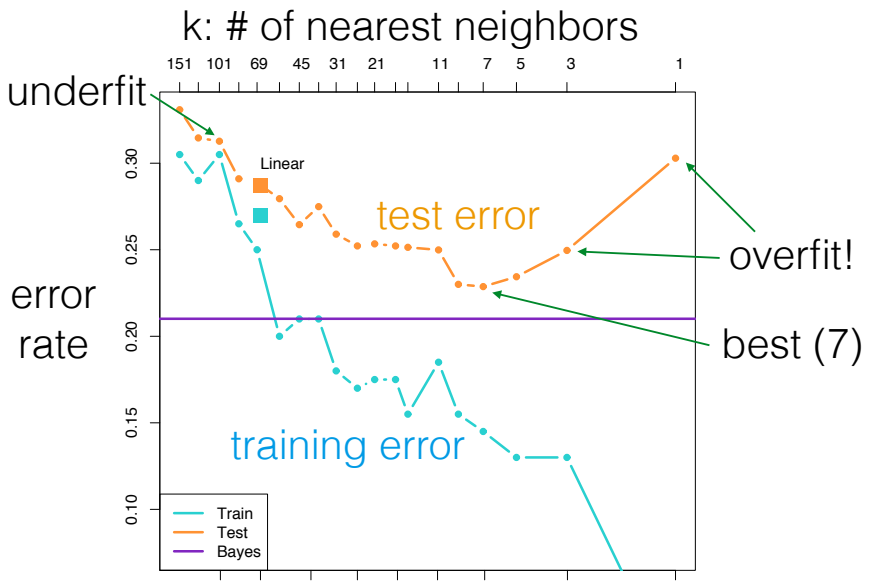
2 kinds of error:

- Training set error: fraction of training images not classified correctly
  [This is zero with the 1-nearest neighbor classifier, but nonzero with the 15-nearest neighbor and linear classifiers we've just seen.]
- Test set error: try out new images, not used during training.

[When I underline a word or phrase, that usually means it's a definition. If you want to do well in this course, my advice to you is to memorize the definitions I cover in class.]

outliers: points whose labels are atypical (e.g. solvent borrower who defaulted anyway).
overfitting: when the test error deteriorates because the classifier becomes too sensitive to outliers or other spurious patterns.

[In machine learning, the goal is to create a classifier that generalizes to new examples we haven't seen yet. Overfitting is counterproductive to that goal. So we're always seeking a compromise between decision boundaries that make fine distinctions and decision boundaries that are downright superstitious.]



overfitlabeled.pdf (modified from ESL, Figure 2.4)

Most ML algorithms have a few hyperparameters that control over/underfitting, e.g. k in k-nearest neighbors. We select them by

validation:
- Hold back a subset of training data, called the validation set.
- Train the classifier multiple times with different hyperparameter settings.
- Choose the settings that work best on validation set.

Now we have 3 sets:
training set used to learn model weights
validation set used to tune hyperparameters, choose among different models
test set used as FINAL evaluation of model. Keep in a vault. Run ONCE, at the very end.
[It's very bad when researchers in medicine or pharmaceuticals peek into the test set prematurely!]

Kaggle.com:
runs ML competitions, including our HWs
we use 2 test sets:
"public" set results available during competition
"private" set revealed only after due date
[If your public results are a lot better than your private results, we will know that you overfitted.]

Techniques [taught in this class, NOT a complete list]

Supervised learning:
- Classification: is this email spam?
- Regression: how likely does this patient have cancer?

Unsupervised learning:
- Clustering: which DNA sequences are similar to each other?
- Dimensionality reduction: what are common features of faces? common differences?

2 Linear Classifiers and Perceptrons

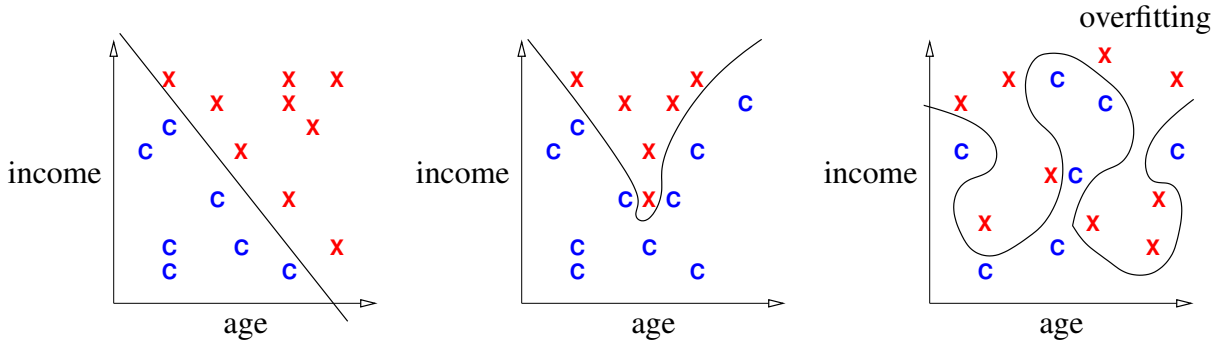
CLASSIFIERS

You are given sample of n observations, each with d features.
Some observations belong to class C; some do not.

Example: Observations are bank loans
Features are income & age ($d = 2$)
Some are in class "defaulted", some are not

Goal: Predict whether future borrowers will default,
based on their income & age.

Represent each observation as a point in d-dimensional space,
called a sample point / a feature vector / predictors / independent variables.



[Draw this by hand; decision boundaries last. classify3.pdf]

[We draw these lines/curves separating C's from X's. Then we use these curves to predict which future borrowers will default. In the last example, though, we're probably overfitting, which could hurt our predictions.]

decision boundary: the boundary chosen by our classifier to separate items in the class from those not.

[By the way, when I underline a word or a short phrase, usually that is a *definition*. If you want to do well in this course, you should *memorize* all the definitions I write down.]

Some (not all) classifiers work by computing a

predictor function: A function $f(x)$ that maps a sample point x to a scalar such that

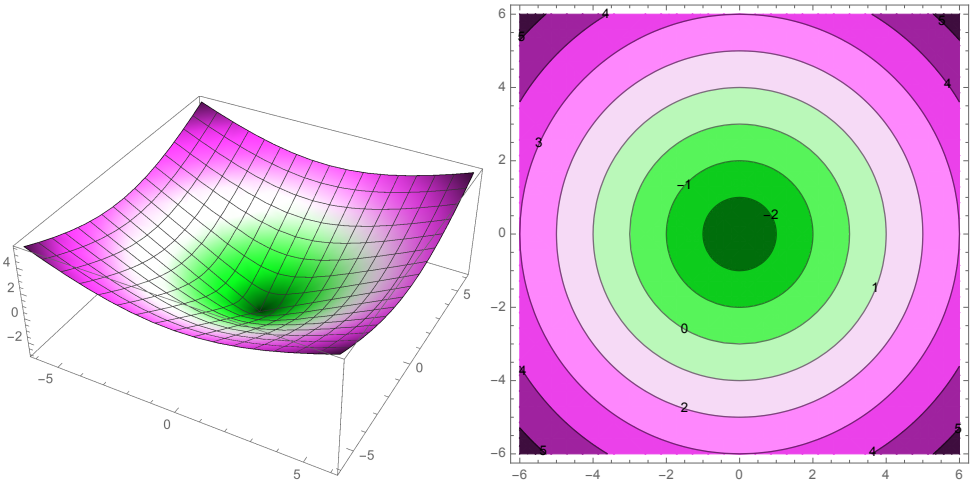
$f(x) > 0$ if $x \in$ class C;
$f(x) \leq 0$ if $x \notin$ class C.

Aka decision function or discriminant function.

For these classifiers, the decision boundary is $\{x \in \mathbb{R}^d : f(x) = 0\}$
[That is, the set of all points where the prediction function is zero.]
Usually, this set is a $(d-1)$-dimensional surface in $\mathbb{R}^d$.

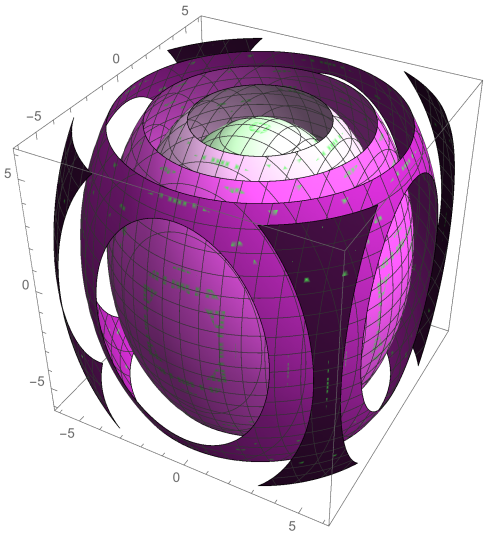
$\{x : f(x) = 0\}$ is also called an isosurface of f for the isovalue 0.

f has other isosurfaces for other isovalues, e.g. $\{x : f(x) = 1\}$.



radiusplot.pdf, radiusiso.pdf [Plot and isocontours of] $f(x, y) = \sqrt{x^2 + y^2} - 3$.

[Imagine a function in $\mathbb{R}^d$, and imagine its $(d-1)$-dimensional isosurfaces.]



radiusiso3d.pdf

linear classifier: The decision boundary is a line/plane.
Usually uses a linear predictor function. [Sometimes no predictor fn.]

overfitting: When sinuous decision boundary fits sample points so well that it doesn't classify future points well.

Math Review

[I will write vectors in matrix notation.]

Vectors: $x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = [x_1 \;\; x_2 \;\; x_3 \;\; x_4 \;\; x_5]^\top$

Think of x as a point in 5-dimensional space.

Conventions (often, but not always):

uppercase roman = matrix, random variable, set	X
lowercase roman = vector	x
Greek = scalar	α
Other scalars:	n = # of sample points
	d = # of features (per point)
	= dimension of sample points
	i j k = indices
function (often scalar)	$f(\,)$, $s(\,)$, ...

inner product (aka dot product): $x \cdot y = x_1 y_1 + x_2 y_2 + ... + x_d y_d$

also written $x^\top y$

Clearly, $f(x) = w \cdot x + \alpha$ is a linear function in x.

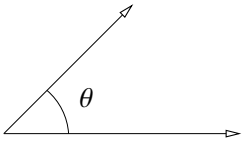
Euclidean norm: $|x| = \sqrt{x \cdot x} = \sqrt{x_1^2 + x_2^2 + ... + x_d^2}$

$|x|$ is the length (aka Euclidean length) of a vector x.

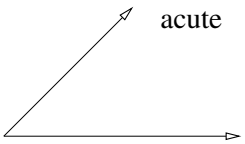
Given a vector x, $\frac{x}{|x|}$ is a unit vector (length 1).

"Normalize a vector x": replace x with $\frac{x}{|x|}$.

Use dot products to compute angles:

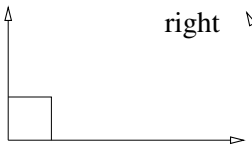


$$\cos\theta = \frac{x \cdot y}{|x|\,|y|} = \underbrace{\frac{x}{|x|}}_{\text{length 1}} \cdot \underbrace{\frac{y}{|y|}}_{\text{length 1}}$$



acute

$\cos\theta > 0$



right

$\cos\theta = 0$

obtuse

$\cos\theta < 0$

Given a linear predictor function $f(x) = w \cdot x + \alpha$, the decision boundary is

$H = \{x : w \cdot x = -\alpha\}$.

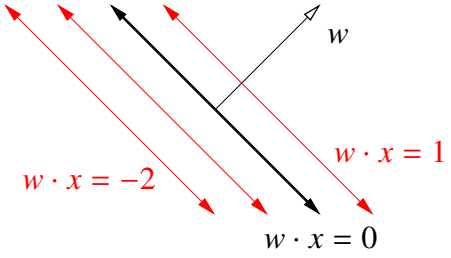
The set H is called a hyperplane. (A line in 2D, a plane in 3D.)

[I want you to understand what a hyperplane is. In 2D, it's a line. In 3D, it's a plane. Now take that concept and generalize it to higher dimensions. In d dimensions, a hyperplane is a flat, infinite thing with dimension $d - 1$. A hyperplane divides the d-dimensional space into two halves.]

Theorem: Let $\vec{xy}$ be a vector that lies on H. Then $w \cdot (y - x) = 0$.

Proof: x and y lie on H. Thus $w \cdot (x - y) = -\alpha - (-\alpha) = 0$.

w is called the normal vector of H,
because (as the theorem shows) w is normal (perpendicular) to H.
(I.e. w is normal to every pair of points in H.)



[Draw black part first, then red parts. hyperplane.pdf]

If w is a unit vector, then $w \cdot x + \alpha$ is the signed distance from x to H.
I.e. it's the distance, but positive on one side of H; negative on other side.

Moreover, the distance from H to the origin is α. [How do we know that?]

Hence $\alpha = 0$ if and only if H passes through origin.

[w does not have to be a unit vector for the classifier to work.
If w is not a unit vector, $w \cdot x + \alpha$ is a multiple of the signed distance.
If you want to fix that, you can rescale the equation
by computing $|w|$ and dividing both w and α by $|w|$.]

The coefficients in w, plus α, are called weights or sometimes regression coefficients.

[That's why I named the vector w; "w" stands for "weights."]

The input data is linearly separable if there exists a hyperplane that separates all the sample points in class C from all the points NOT in class C.

[At the beginning of this lecture, I showed you one plot that's linearly separable and two that are not.]

[We will investigate some linear classifiers that only work for linearly separable data, then we'll move on to more sophisticated linear classifiers that do a decent job with non-separable data. Obviously, if your data is not linearly separable, a linear classifier cannot do a *perfect* job. But we're still happy if we can find a classifier that usually predicts correctly.]

A Simple Classifier

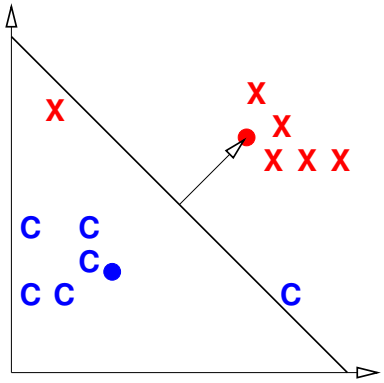
Centroid method: compute mean μ_C of all vectors in class C and mean μ_X of all vectors NOT in C.

We use the predictor function

$$f(x) = \underbrace{(\mu_C - \mu_X)}_{\text{normal vector}} \cdot x - (\mu_C - \mu_X) \cdot \underbrace{\frac{\mu_C + \mu_X}{2}}_{\text{midpoint between } \mu_C, \mu_X}$$

so that decision boundary is the hyperplane that bisects line segment w/endpoints μ_C, μ_X.

[Better yet, we can adjust the right hand side to minimize the number of misclassified points. Same normal vector, but different position.]



[Draw data, then μ_C, μ_X, then line & normal. centroid.pdf]

[In this example, there's clearly a better linear classifier that classifies every sample point correctly.
Note that this is hardly the worst example I could have given.
If you're in the mood for an easy puzzle, pull out a sheet of paper and think of an example, with lots of sample points, where the centroid method misclassifies every sample point but one.]

[Nevertheless, there are cases where this method works well, like when all your positive examples come from one Gaussian distribution, and all your negative examples come from another.]

Perceptron Algorithm (Frank Rosenblatt, 1957)

Slow, but correct for linearly separable points.

Uses a numerical optimization algorithm, namely, gradient descent.

[Poll:
How many of you know what numerical optimization is?
How many of you know what gradient descent is?
How many of you know what Lagrange multipliers are?
How many of you know what linear programming is?
How many of you know what the simplex algorithm for linear programming is?
How many of you know what convex programming is?

We're going to learn what most of these things are. As machine learning people, we will be heavy users of all the optimization methods. Unfortunately, I won't have time to teach you *algorithms* for all these optimization problems, but we'll learn a few.]

Consider n sample points $X_1, X_2, ..., X_n$.

[The reason I'm using capital X here is because we typically store these vectors as rows of a matrix X. So the subscript picks out a row of X, representing a specific sample point.]

For each sample point, let $y_i = \begin{cases} 1 & \text{if } X_i \in \text{class C, and} \\ -1 & \text{if } X_i \notin \text{C.} \end{cases}$

For simplicity, consider only decision boundaries that pass through the origin. (We'll fix this later.)

Goal: find weights w such that

$X_i \cdot w \geq 0$ if $y_i = 1$, and
$X_i \cdot w \leq 0$ if $y_i = -1$. [remember, $X_i \cdot w$ is the signed distance]

Equivalently: $y_i X_i \cdot w \geq 0$. ← inequality called a constraint.

Idea: We define a risk function R that is positive if some constraints are violated. Then we use optimization to choose w that minimizes R.

Define the loss function

$$L(z, y_i) = \begin{cases} 0 & \text{if } y_i z \geq 0, \text{ and} \\ -y_i z & \text{otherwise.} \end{cases}$$

[Here, z is the classifier's prediction, and y_i is the correct answer.]

Idea: if z has the same sign as y_i, the loss function is zero (happiness).
But if z has the wrong sign, the loss function is positive.

[For each sample point, you want to get the loss function down to zero, or as close to zero as possible. It's called the "loss function" because the bigger it is, the bigger a loser you are.]

Define risk function (aka objective function or cost function)

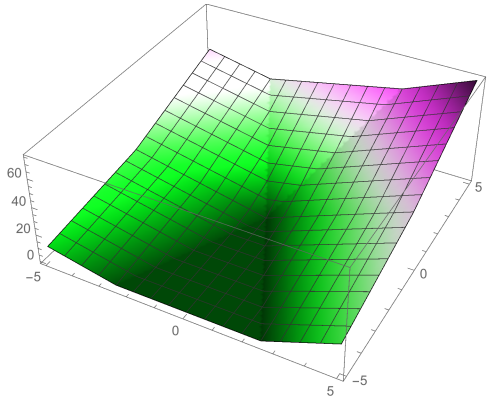
$$\begin{aligned} R(w) &= \sum_{i=1}^{n} L(X_i \cdot w, y_i), \\ &= \sum_{i \in V} -y_i X_i \cdot w \end{aligned}$$

where V is the set of indices i for which $y_i X_i \cdot w < 0$.

If w classifies all $X_1, \ldots, X_n$ correctly, then $R(w) = 0$.
Otherwise, $R(w)$ is positive, and we want to find a better value of w.

Goal: Solve this optimization problem:

Find w that minimizes $R(w)$.



riskplot.pdf [Plot of risk $R(w)$. Every point in the dark green flat spot is a minimum. We'll look at this more next lecture.]

3 Perceptron Learning; Maximum Margin Classifiers

Perceptron Algorithm (cont'd)

Recall:

- linear predictor fn $f(x) = w \cdot x$ (for simplicity, no α)
- decision boundary $\{x : f(x) = 0\}$ (a hyperplane through the origin)
- sample points $X_1, X_2, \ldots, X_n \in \mathbb{R}^d$; classifications $y_1, \ldots, y_n = \pm 1$
- goal: find weights w such that $y_i X_i \cdot w \geq 0$
- goal, rewritten: find w that minimizes $R(w) = \sum_{i \in V} -y_i X_i \cdot w$ [risk function]
  where V is the set of indices i for which $y_i X_i \cdot w < 0$.

[Our original problem was to find a separating hyperplane in one space, which I'll call x-space. But we've transformed this into a problem of finding an optimal point in a different space, which I'll call w-space. It's important to understand transformations like this, where a geometric structure in one space becomes a point in another space.]

Objects in x-space transform to objects in w-space:

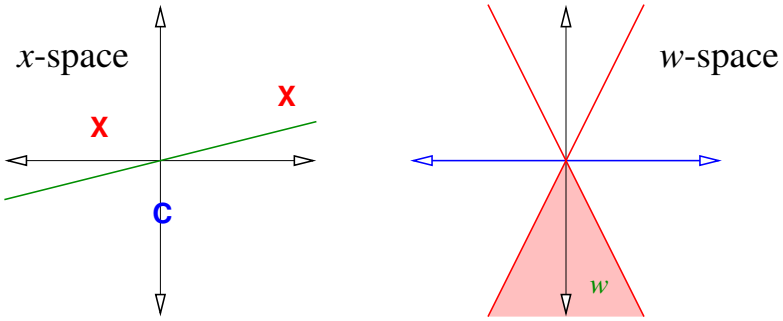
x-space		w-space	
hyperplane:	$\{z : w \cdot z = 0\}$	point:	w
point:	x	hyperplane:	$\{z : x \cdot z = 0\}$

Point x lies on hyperplane $\{z : w \cdot z = 0\} \Leftrightarrow w \cdot x = 0 \Leftrightarrow$ point w lies on hyperplane $\{z : x \cdot z = 0\}$ in w-space.

[You'll notice that in this case, the transformations are symmetric: a hyperplane in x-space transforms to a point in w-space the same way that a hyperplane in w-space transforms to a point in x-space. That won't always be true for the weight spaces we use this semester, but in this case, it makes the transformations easier to remember.]

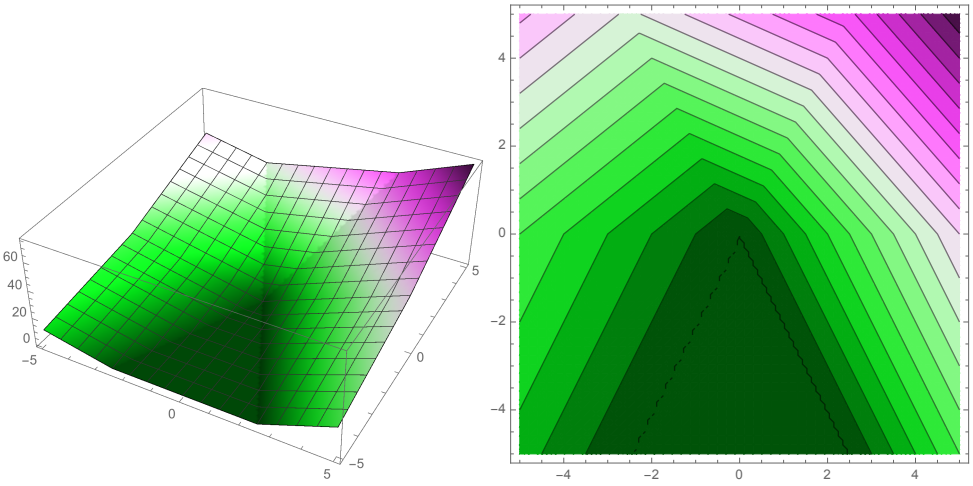
If we want to enforce inequality $x \cdot w \geq 0$, that means

- in x-space, x should be on the same side of $\{z : z \cdot w = 0\}$ as w
- in w-space, w " " " " " " " $\{z : x \cdot z = 0\}$ as x



[Draw this by hand. xwspace.pdf]
[Observe that the x-space points are the normal vectors for the w-space lines. We can choose w to be anywhere in the shaded region.]

[For a sample point x in class C, w and x must be on the *same* side of the hyperplane that x transforms into. For a point x not in class C (call it class X), w and x must be on *opposite* sides of the hyperplane that x transforms into. These rules determine the shaded region above, in which w must lie.]



riskplot.pdf, riskiso.pdf [Plot & isocontours of risk $R(w)$. Note how R's creases match the dual chart above.]

[In this plot, we can choose w to be any point in the bottom pizza slice; all those points minimize R.]

[We have an optimization problem; we need an optimization algorithm to solve it.]

An optimization algorithm: gradient descent on R.

Given a starting point w, find gradient of R with respect to w; this is the direction of steepest ascent. Take a step in the opposite direction. Recall [from your vector calculus class]

$$\nabla R(w) = \begin{bmatrix} \frac{\partial R}{\partial w_1} \\ \frac{\partial R}{\partial w_2} \\ \vdots \\ \frac{\partial R}{\partial w_d} \end{bmatrix} \qquad \text{and} \qquad \nabla(z \cdot w) = \begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_d \end{bmatrix} = z$$

$$\nabla R(w) = \sum_{i \in V} \nabla - y_i X_i \cdot w = -\sum_{i \in V} y_i X_i$$

At any point w, we walk downhill in direction of steepest descent, $-\nabla R(w)$.

$w \leftarrow$ arbitrary nonzero starting point (good choice is any $y_i X_i$)
while $R(w) > 0$
    $V \leftarrow$ set of indices i for which $y_i X_i \cdot w < 0$
    $w \leftarrow w + \epsilon \sum_{i \in V} y_i X_i$
return w

ϵ is the step size aka learning rate, chosen empirically. [Best choice depends on input problem!]

[Show plot of R again. Draw the typical steps of gradient descent.]

Problem: Slow! Each step takes $O(nd)$ time. [Can we improve this?]

Optimization algorithm 2: stochastic gradient descent

Idea: each step, pick **one** misclassified X_i;
do gradient descent on loss fn $L(X_i \cdot w, y_i)$.

Called the perceptron algorithm. Each step takes $O(d)$ time.
[Not counting the time to search for a misclassified X_i.]

while some $y_i X_i \cdot w < 0$
    $w \leftarrow w + \epsilon\, y_i X_i$
return w

[By the way, stochastic gradient descent does not work for every problem that gradient descent works for. The perceptron risk function happens to have special properties that guarantee that stochastic gradient descent will always succeed.]

What if separating hyperplane doesn't pass through origin?
Add a fictitious dimension.

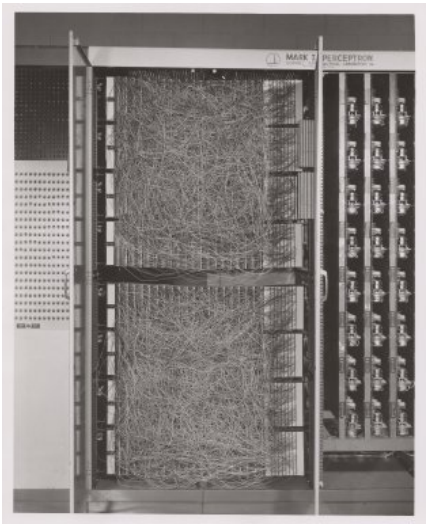
Hyperplane: $w \cdot x + \alpha = 0$

$$[w_1 \;\; w_2 \;\; \alpha] \cdot \begin{bmatrix} x_1 \\ x_2 \\ 1 \end{bmatrix} = 0$$

Now we have points in $\mathbb{R}^{d+1}$, all lying on plane $x_{d+1} = 1$.

Run perceptron algorithm in $(d+1)$-dimensional space.

[The perceptron algorithm was invented in 1957 by Frank Rosenblatt at the Cornell Aeronautical Laboratory. It was originally designed not to be a program, but to be implemented in hardware for image recognition on a 20×20 pixel image. Rosenblatt built a Mark I Perceptron Machine that ran the algorithm, complete with electric motors to do weight updates.]



Mark_I_perceptron.jpg (from Wikipedia, "Perceptron") [The Mark I Perceptron Machine. This is what it took to process a 20×20 image in 1957.]

[Then he held a press conference where he predicted that perceptrons would be "the embryo of an electronic computer that [the Navy] expects will be able to walk, talk, see, write, reproduce itself and be conscious of its existence." We're still waiting on that.]

[One interesting aspect of the perceptron algorithm is that it's an "online algorithm," which means that if new data points come in while the algorithm is already running, you can just throw them into the mix and keep looping.]

Perceptron Convergence Theorem: If data is linearly separable, perceptron algorithm will find a linear classifier that classifies all data correctly in at most $O(R^2/\gamma^2)$ iterations, where $R = \max |X_i|$ is "radius of data" and γ is the "maximum margin."
[I'll define "maximum margin" shortly.]

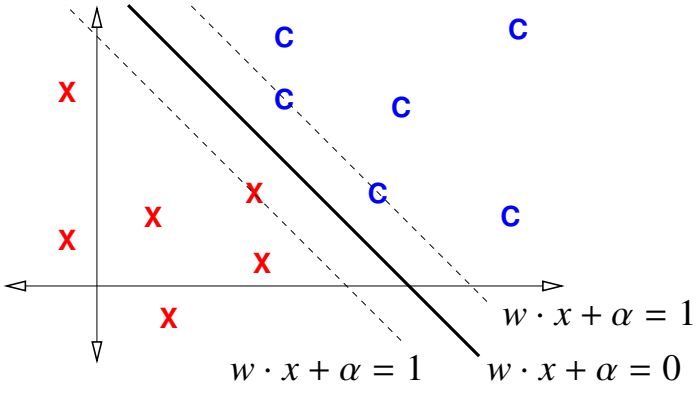
[We're not going to prove this, because perceptrons are obsolete.]

[Although the step size/learning rate doesn't appear in that big-O expression, it does have an effect on the running time, but the effect is hard to characterize. The algorithm gets slower if ϵ is too small because it has to take lots of steps to get down the hill. But it also gets slower if ϵ is too big for a different reason: it jumps right over the region with zero risk and oscillates back and forth for a long time.]

[Although stochastic gradient descent is faster for this problem than gradient descent, the perceptron algorithm is still slow. There's no reliable way to choose a good step size ϵ. Fortunately, optimization algorithms have improved a lot since 1957. You can get rid of the step size by using any decent modern "line search" algorithm. Better yet, you can find a better decision boundary much more quickly by quadratic programming, which is what we'll talk about next.]

MAXIMUM MARGIN CLASSIFIERS

The margin of a linear classifier is the distance from the decision boundary to the nearest sample point. What if we make the margin as wide as possible?



[Draw this by hand. maxmargin.pdf]

We enforce the constraints

$$y_i (w \cdot X_i + \alpha) \geq 1 \quad \text{for } i \in [1, n]$$

[Notice that the right-hand side is a 1, rather than a 0 as it was for the perceptron risk function. It's not obvious, but this a much better way to formulate the problem, partly because it makes it impossible for the weight vector w to get set to zero.]

If $|w| = 1$, the constraints imply the margin is at least 1; [because $w \cdot X_i + \alpha$ is the signed distance]

BUT we allow w to have arbitrary length, so the margin is at least $\frac{1}{|w|}$.

There is a slab of width $\frac{2}{|w|}$ containing no sample points [with the hyperplane running along its middle].

To maximize the margin, minimize $|w|$. Optimization problem:

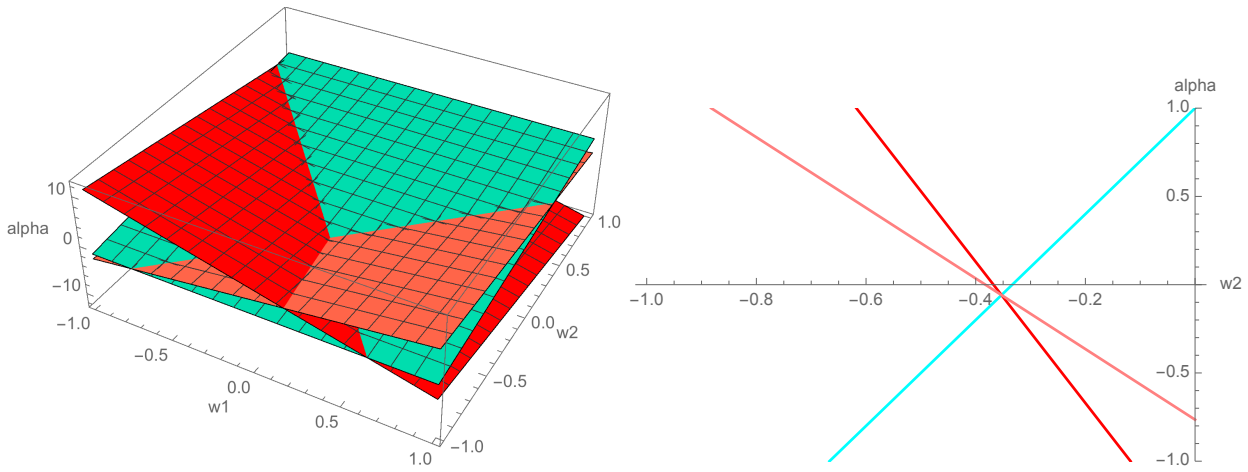
> Find w and α that minimize $|w|^2$
> subject to $y_i(X_i \cdot w + \alpha) \geq 1$ for all $i \in [1, n]$

Called a quadratic program in $d + 1$ dimensions and n constraints.
It has one unique solution!

[A reason we use $|w|^2$ as an objective function, instead of $|w|$, is that the length function $|w|$ is not smooth at zero, whereas $|w|^2$ is smooth everywhere. This makes optimization easier.]

The solution gives us a maximum margin classifier, aka a hard margin support vector machine (SVM).

[Technically, this isn't really a support vector machine yet; it doesn't fully deserve that name until we add features and kernelization, which we'll do in later lectures.]

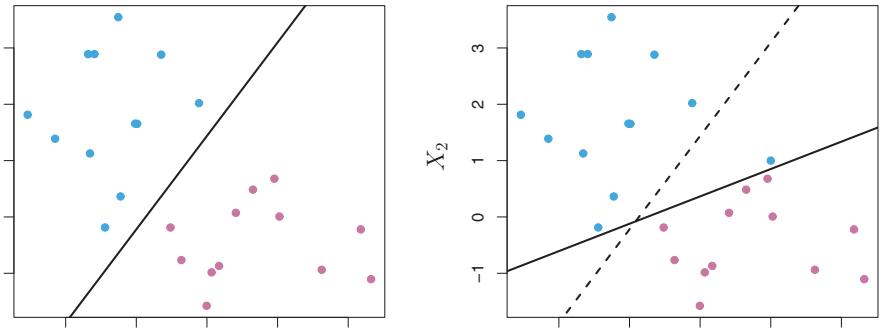


weight3d.pdf, weightcross.pdf [This is an example of what the linear constraints look like in the 3D weight space (w_1, w_2, α) for an SVM with three training points. The SVM is looking for the point nearest the origin that lies above the blue plane (representing an in-class training point) but below the red and pink planes (representing out-of-class training points). In this example, that optimal point lies where the three planes intersect. At right we see a 2D cross-section $w_1 = 1/17$ of the 3D space, because the optimal solution lies in this cross-section. The constraints say that the solution must lie in the leftmost pizza slice, while being as close to the origin as possible, so the optimal solution is where the three lines meet.]

4 Soft-Margin Support Vector Machines; Features

SOFT-MARGIN SUPPORT VECTOR MACHINES (SVMs)

Solves 2 problems:
- Hard-margin SVMs fail if data not linearly separable.
- " " " sensitive to outliers.



sensitive.pdf (ISL, Figure 9.5) [Example where one outlier moves the decision boundary a lot.]

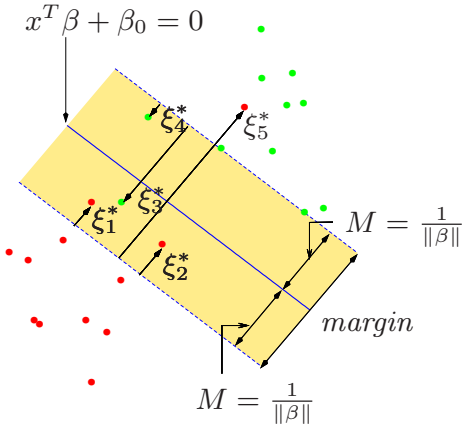
Idea: Allow some points to violate the margin, with slack variables.
Modified constraint for point i:

$$y_i(X_i \cdot w + \alpha) \geq 1 - \xi_i$$

[Observe that the only difference between these constraints and the hard margin constraints we saw last lecture is the extra slack term ξ_i.]
[We also impose new constraints, that the slack variables are never negative.]

$$\xi_i \geq 0$$

[This inequality ensures that all sample points that *don't* violate the margin are treated the same; they all have $\xi_i = 0$. Point i has nonzero ξ_i if and only if it violates the margin.]



slack.pdf (ESL, Figure 12.1) [A margin where some points have slack. For each violating point, the slack distance is $\xi_i^* = \xi_i/|w|$.]

To prevent abuse of slack, we add a loss term to objective fn.

Optimization problem:

Find w, α, and ξ_i that minimize $|w|^2 + C \sum_{i=1}^{n} \xi_i$
subject to $y_i(X_i \cdot w + \alpha) \geq 1 - \xi_i$ for all $i \in [1, n]$
$\xi_i \geq 0$ for all $i \in [1, n]$

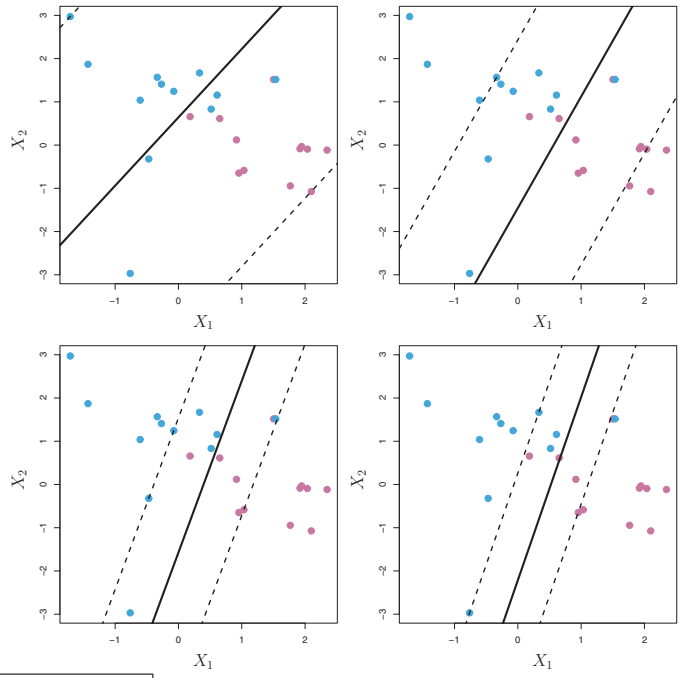
. . . a quadratic program in $d + n + 1$ dimensions and $2n$ constraints.
[It's a quadratic program because its objective function is quadratic and its constraints are linear inequalities.]

$C > 0$ is a scalar regularization hyperparameter that trades off:

	small C	big C
desire	maximize margin $1/\|w\|$	keep most slack variables zero or small
danger	underfitting (misclassifies much training data)	overfitting (awesome training, awful test)
outliers	less sensitive	very sensitive
boundary	more "flat"	more sinuous

[The last row only applies to nonlinear decision boundaries, which we'll discuss next. Obviously, a linear decision boundary can't be sinuous.]

Use validation to choose C.



svmC.pdf (ISL, Figure 9.7) [Examples of how slab varies with C. Smallest C upper left; largest C lower right.]

[One way to think about slack is to pretend that slack is money we can spend to buy permission for a sample point to violate the margin. The further a point penetrates the margin, the bigger the fine you have to pay. We want to make the margin as big as possible, but we also want to spend as little money as possible. If the regularization parameter C is small, it means we're willing to spend lots of money on violations so we can get a bigger margin. If C is big, it means we're cheap and we want to prevent violations, even though we'll get a narrower margin . If C is infinite, we're back to a hard-margin SVM.]

FEATURES

Q: How to do nonlinear decision boundaries?

A: Make nonlinear features that lift points into a higher-dimensional space.
High-d linear classifier → low-d nonlinear classifier.

[Features work with all classifiers, including perceptrons, hard-margin SVMs, and soft-margin SVMs.]

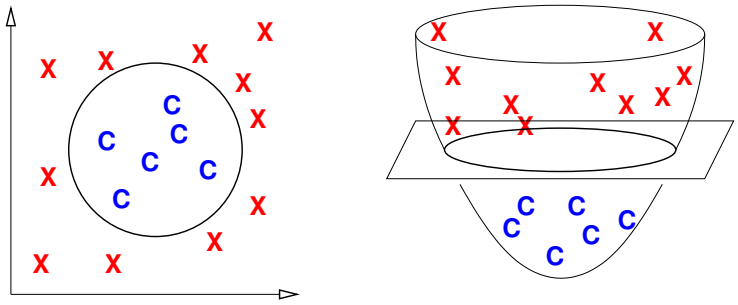
Example 1: The parabolic lifting map

$$\Phi(x) : \mathbb{R}^d \to \mathbb{R}^{d+1}$$

$$\Phi(x) = \begin{bmatrix} x \\ |x|^2 \end{bmatrix} \quad \leftarrow \text{lifts } x \text{ onto paraboloid } x_{d+1} = |x|^2$$

[We've added a new feature, $|x|^2$. Even though the new feature is just a function of other input features, it gives our linear classifier more power.]

Find a linear classifier in Φ-space.
It induces a sphere classifier in x-space.



[Draw this by hand. circledec.pdf]

Theorem: $\Phi(X_1), \ldots, \Phi(X_n)$ are linearly separable iff $X_1, \ldots, X_n$ are separable by a hypersphere.
(Possibly a degenerate hypersphere = hyperplane.)

Proof: Consider hypersphere in $\mathbb{R}^d$ w/center c & radius ρ. Points inside:

$$|x - c|^2 < \rho^2$$
$$|x|^2 - 2c \cdot x + |c|^2 < \rho^2$$
$$\underbrace{[-2c^\top \ \ 1]}_{\text{normal vector}} \underbrace{\begin{bmatrix} x \\ |x|^2 \end{bmatrix}}_{\Phi(x)} < \rho^2 - |c|^2$$

Hence points inside sphere → same side of hyperplane in Φ-space.
(Reverse implication works too.)

[Although the math above doesn't expose it, hyperplane separators are a special case of hypersphere separators, so hypersphere classifiers can do everything linear classifiers can do and more. If you take a sphere and increase its radius to infinity while making it pass through some point, in the limit you get a plane; so you can think of a plane as a degenerate sphere. With the parabolic lifting map, a hyperplane in x-space corresponds to a hyperplane in Φ-space that is parallel to the x_{d+1}-axis.]

Example 2: Axis-aligned ellipsoid/hyperboloid decision boundaries

[Draw examples of axis-aligned ellipses & hyperbola.]

In 3D, these have the formula

$$Ax_1^2 + Bx_2^2 + Cx_3^2 + Dx_1 + Ex_2 + Fx_3 = -G$$

[Here, the capital letters are scalars, not matrices.]

$$\Phi(x) : \mathbb{R}^d \to \mathbb{R}^{2d}$$

$$\Phi(x) = [x_1^2 \quad \dots \quad x_d^2 \quad x_1 \quad \dots \quad x_d]^\top$$

[We've turned d input features into $2d$ features for our linear classifier. If the points are separable by an axis-aligned ellipsoid or hyperboloid, per the formula above, then the points lifted to Φ-space are separable by a hyperplane whose normal vector is (A, B, C, D, E, F).]

Example 3: Ellipsoid/hyperboloid

[Draw example of non-axis-aligned ellipse.]

General formula: [for an ellipsoid or hyperboloid]

$$Ax_1^2 + Bx_2^2 + Cx_3^2 + Dx_1x_2 + Ex_2x_3 + Fx_3x_1 + Gx_1 + Hx_2 + Ix_3 = -J$$

$$\Phi(x) : \mathbb{R}^d \to \mathbb{R}^{(d^2+3d)/2}$$

[The isosurface defined by this equation is called a quadric. In the special case of two dimensions, it's also known as a conic section. So our decision boundary can be an arbitrary conic section.]

[You'll notice that there is a quadratic blowup in the number of features, because every *pair* of input features creates a new feature in Φ-space. If the dimension is large, these feature vectors are getting huge, and that's going to impose a serious computational cost. But it might be worth it to find good classifiers for data that aren't linearly separable.]

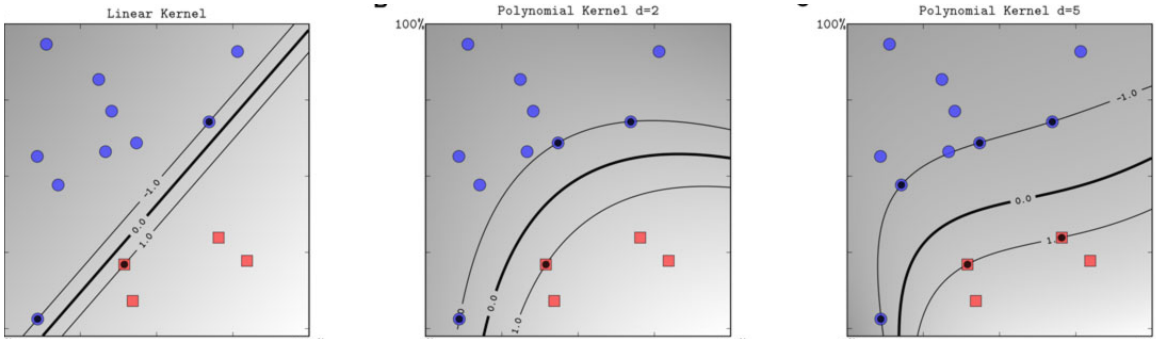
Example 4: Predictor fn is degree-p polynomial

E.g. a cubic in $\mathbb{R}^2$:

$$\Phi(x) = [x_1^3 \quad x_1^2 x_2 \quad x_1 x_2^2 \quad x_2^3 \quad x_1^2 \quad x_1 x_2 \quad x_2^2 \quad x_1 \quad x_2]^\top$$

$$\Phi(x) : \mathbb{R}^d \to \mathbb{R}^{O(d^p)}$$

[Now we're really blowing up the number of features! If you have, say, 100 features per sample point and you want to use degree-4 predictor functions, then each lifted feature vector has a length on the order of 100 million, and your learning algorithm will take approximately forever to run.]
[However, later in the semester we will learn an extremely clever trick that allows us to work with these huge feature vectors very quickly, without ever computing them. It's called "kernelization" or "the kernel trick." So even though it appears now that working with degree-4 polynomials is computationally infeasible, it can actually be done very quickly.]

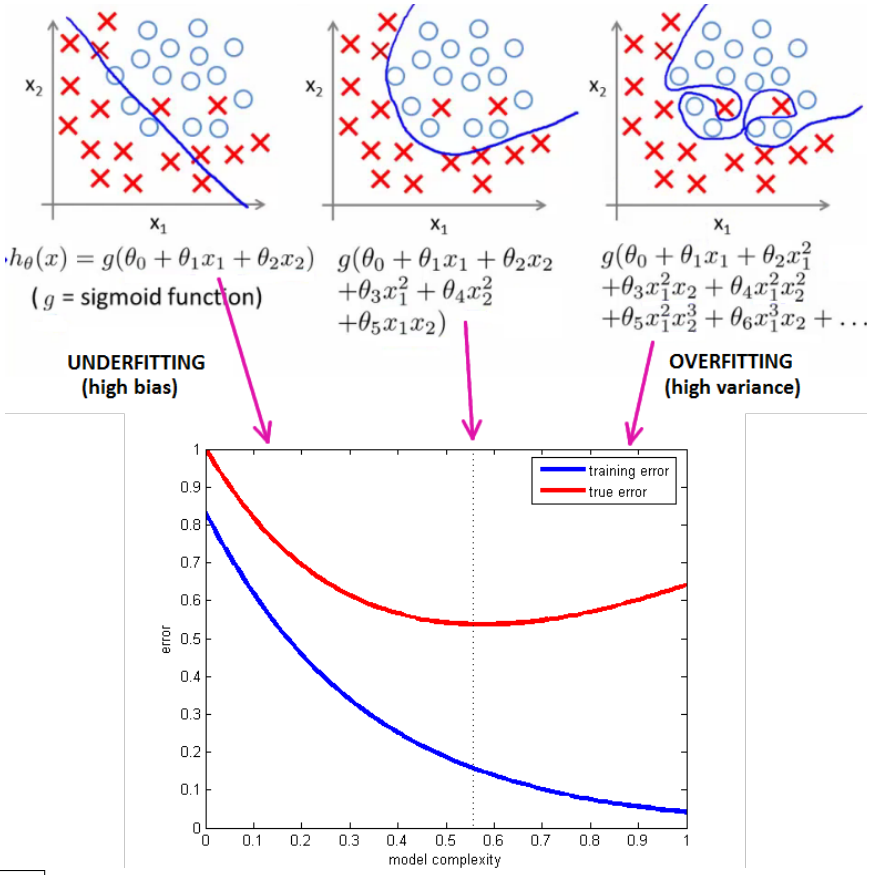


degree5.pdf [SVMs with degree 1/2/5 predictor functions. Observe that the margin tends to get wider as the degree increases.]

[Increasing the degree like this accomplishes two things.

- First, the data might become linearly separable when you lift them to a high enough degree, even if the original data are not linearly separable.
- Second, raising the degree can increase the margin, so you might get a more robust separator.

However, if you raise the degree too high, you will overfit the data.]



overfit.pdf [Training vs. test error for degree 1/2/5 predictor functions. (Artist's conception; these aren't actual calculations, just hand-drawn guesses. Please send me email if you know where to find figures like this with actual data.) In this example, a degree-2 predictor gives the smallest test error.]

[Sometimes you should search for the ideal degree—not too small, not too big. It's a balancing act between underfitting and overfitting. The degree is an example of a *hyperparameter* that can be optimized by validation.]

[If you're using both polynomial features and a soft-margin SVM, now you have two hyperparameters: the degree and C. Generally, the optimal C will be different for every polynomial degree, so when you change degree, you have to run validation again to find the best C for that degree.]

[So far I've talked only about polynomial features. But features can get much more complicated than polynomials, and they can be tailored to fit a specific problem. Let's consider a type of feature you might use if you wanted to implement, say, a handwriting recognition algorithm.]

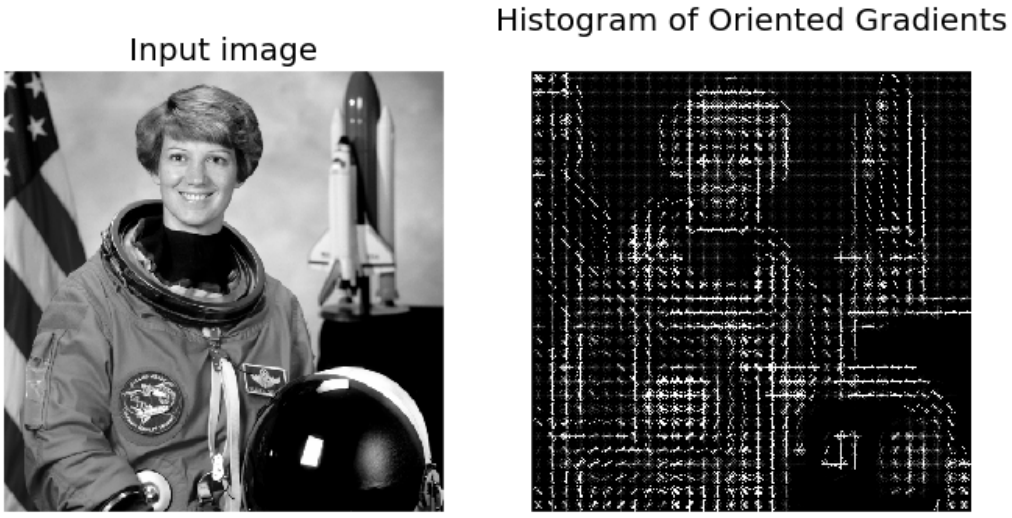
Example 5: Edge detection

Edge detector: algorithm for approximating grayscale/color gradients in image, e.g.

- tap filter
- Sobel filter
- oriented Gaussian derivative filter
  [images are discrete, not continuous fields, so approximation of gradients is necessary.]

[See "Image Derivatives" on Wikipedia.]

Collect line orientations in local histograms (each having 12 orientation bins per region); use histograms as features (*instead* of raw pixels).



orientgrad.png [Image histograms.]

Paper: Maji & Malik, 2009.

[If you want to, optionally, use these features in Homework 1 and try to win the Kaggle competition, this paper is a good online resource.]

[When they use a linear SVM on the raw pixels, Maji & Malik get an error rate of 15.38% on the test set. When they use a linear SVM on the histogram features, the error rate goes down to 2.64%.]

[Many applications can be improved by designing application-specific features. There's no limit but your own creativity and ability to discern the structure hidden in your application.]

5 Machine Learning Abstractions and Numerical Optimization

ML ABSTRACTIONS [some meta comments on machine learning]

[When you write a large computer program, you break it down into subroutines and modules. Many of you know from experience that you need to have the discipline to impose strong abstraction barriers between different modules, or your program will become so complex you can no longer manage nor maintain it.]

[When you learn a new subject, it helps to have mental abstraction barriers, too, so you know when you can replace one approach with a different approach. I want to give you four levels of abstraction that can help you think about machine learning. It's important to make mental distinctions between these four things, and the code you write should have modules that reflect these distinctions as well.]

APPLICATION/DATA

data labeled (classified) or not?
yes: labels categorical (classification) or quantitative (regression)?
no: similarity (clustering) or positioning (dimensionality reduction)?

MODEL [what kinds of hypotheses are permitted?]

e.g.:
– predictor fns: linear, polynomial, logistic, neural net, ...
– nearest neighbors, decision trees
– features
– low vs. high capacity (affects overfitting, underfitting, inference)

OPTIMIZATION PROBLEM

– variables, objective fn, constraints
e.g., unconstrained, convex program, least squares, PCA

OPTIMIZATION ALGORITHM

e.g., gradient descent, simplex, SVD

[In this course, we focus primarily on the middle two levels. As a data scientist, you might be given an application, and your challenge is to turn it into an optimization problem that we know how to solve. We'll talk a bit about optimization algorithms, but usually you'll use an optimization code that's faster and more robust than what you would write yourself.]

[The second level, the model, has a huge effect on the success of your learning algorithm. Sometimes you get a big improvement by tailoring the model or its features to fit the structure of your specific data. The model also has a big effect on whether you overfit or underfit. And if you want a model that you can interpret so you can do *inference*, the model has to be regular, not too complex. Lastly, you have to pick a model that leads to an optimization problem that can be solved. Some optimization problems are just too hard.]

[It's important to understand that when you change something in one level of this diagram, you probably have to change all the levels underneath it. If you switch from a linear classifier to a neural net, your optimization problem changes, and your optimization algorithm probably changes too.]

[Not all machine learning methods fit this four-level decomposition. Nevertheless, for everything you learn in this class, think about where it fits in this hierarchy. If you don't distinguish which math is part of the model and which math is part of the optimization algorithm, this course will be very confusing for you.]

OPTIMIZATION PROBLEMS

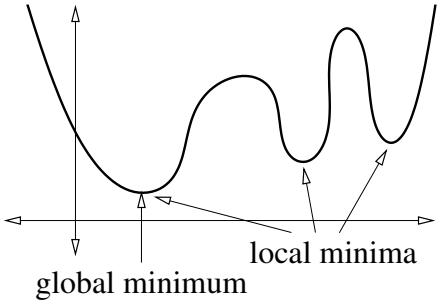
[I want to familiarize you with some types of optimization problems that can be solved reliably and efficiently, and the names of some of the optimization algorithms used to solve them. An important skill for you to develop is to be able to go from an application to a well-defined optimization problem.]

Unconstrained

Goal: Find w that minimizes (or maximizes) a continuous fn $f(w)$.

f is smooth if its gradient is continuous too.

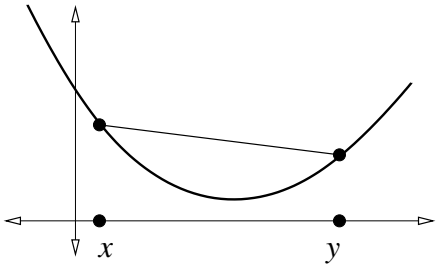
A global minimum of f is a value w such that $f(w) \leq f(v)$ for every v.
A local minimum " " " " " " " " " "
for every v in a tiny ball centered at w.
[In other words, you cannot walk downhill from w.]



[Draw this by hand. minima.pdf]

Usually, finding a local minimum is easy;
finding the global minimum is hard. [or impossible]

Exception: A function is convex if for every $x, y \in \mathbb{R}^d$,
the line connecting $(x, f(x))$ to $(y, f(y))$ does not go below $f(x)$.



[Draw this by hand. convex.pdf]

E.g. perceptron risk fn is convex and nonsmooth.

[When you sum together convex functions, you always get a convex function. The perceptron risk function is a sum of convex loss functions.]

A [continuous] convex function [on a closed, convex domain] has either
- no minimum (goes to $-\infty$), or
- just one local minimum, or
- a connected set of local minima that are all global minima with equal f.

[The perceptron risk function has the last of these three.]
[In the last two cases, if you walk downhill, you eventually converge to a global minimum.]
[However, there are many applications where you don't have a convex objective function, and your machine learning algorithm has to settle for finding a local minimum. For example, neural nets try to optimize an objective function that has *lots* of local minima; they almost never find a global minimum.]

Algs for smooth f:
- Steepest descent:
  - blind [with learning rate] repeat: $w \leftarrow w - \epsilon \nabla f(w)$
  - with line search:
    - Secant method
    - Newton–Raphson (may need Hessian matrix of f)
  - stochastic (blind) [trains on one point per iteration, or a small batch]
- Nonlinear conjugate gradient [uses the same line search methods]
- Newton's method (needs Hessian matrix)

Algs for nonsmooth f:
- Steepest descent
  - blind
  - with direct line search (e.g. golden section search)

These algs find a local minimum. [They don't reliably find a global minimum, because that's hard.]

line search: finds a local minimum along the search direction by solving an optimization problem in 1D.
[...instead of using a blind step size like the perceptron algorithm does. Solving a 1D problem is much easier than solving a higher-dimensional one.]

[Neural nets are unconstrained optimization problems with many, many local minima. They sometimes benefit from the more sophisticated optimization algorithms, but when the input data set is very large, researchers often favor the dumb, blind, stochastic versions of gradient descent.]

[If you're optimizing over a d-dimensional space, the Hessian matrix is a $d \times d$ matrix and it's usually dense, so most methods that use the Hessian are computationally infeasible when d is large.]

Constrained Optimization (smooth equality constraints)

Goal: Find w that minimizes (maximizes) $f(w)$
subject to $g(w) = 0$ [← observe that this is an isosurface]

where g is a smooth fn [g may be a vector, encoding multiple constraints]

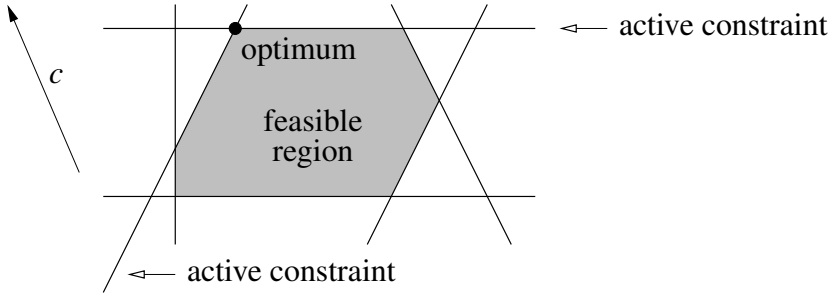
Alg: Use Lagrange multipliers.

Linear Program

Linear objective fn + linear **inequality** constraints.

Goal: Find w that maximizes (or minimizes) $c \cdot w$
subject to $Aw \leq b$

where A is $n \times d$ matrix, $b \in \mathbb{R}^n$, expressing n linear constraints:
$A_i w \leq b_i, \qquad i \in [1, n]$



[Draw this by hand. linprog.pdf]

The set of points that satisfy all constraints is a convex polytope called the feasible region F [shaded].
The optimum is the point in F that is furthest in the direction c. [What does convex mean?]
A point set P is convex if for every $p, q \in P$, the line segment with endpoints p, q lies entirely in P.

[A polytope is just a polyhedron, generalized to higher dimensions.]

The optimum achieves equality for some constraints (but not most), called the active constraints of the optimum. [In the figure above, there are two active constraints. In an SVM, active constraints correspond to the sample points that touch or violate the slab, and they're also known as support vectors.]

[Sometimes, there is more than one optimal point. For example, in the figure above, if c pointed straight up, every point on the top horizontal edge would be optimal. The set of optimal points is always convex.]

Example: EVERY feasible point (w, α) gives a linear classifier:

Find w, α that maximizes 0 subject to $y_i(w \cdot X_i + \alpha) \geq 1 \quad$ for all $i \in [1, n]$

IMPORTANT: The data are linearly separable iff the feasible region is not the empty set.
→ Also true for maximum margin classifier (quadratic program)

Algs for linear programming:
- Simplex (George Dantzig, 1947)
  [Indisputably one of the most important algorithms of the 20th century.]
  [Walks along edges of polytope from vertex to vertex until it finds optimum.]
- Interior point methods

[Linear programming is very different from unconstrained optimization; it has a much more combinatorial flavor. If you knew which constraints would be the active constraints once you found the solution, it would be easy; the hard part is figuring out which constraints should be the active ones. There are exponentially many possibilities, so you can't afford to try them all. So linear programming algorithms tend to have a very discrete, computer science feeling to them, like graph algorithms, whereas unconstrained optimization algorithms tend to have a continuous, numerical optimization feeling.]

[Linear programs crop up everywhere in engineering and science, but they're usually in disguise. An extremely useful talent you should develop is to recognize when a problem is a linear program.]

[A linear program solver can find a linear classifier, but it can't find the maximum margin classifier. We need something more powerful.]

Quadratic program

Quadratic, convex objective fn + linear inequality constraints.

Goal: Find w that minimizes $f(w) = w^\top Q w + c^\top w$
subject to $Aw \leq b$

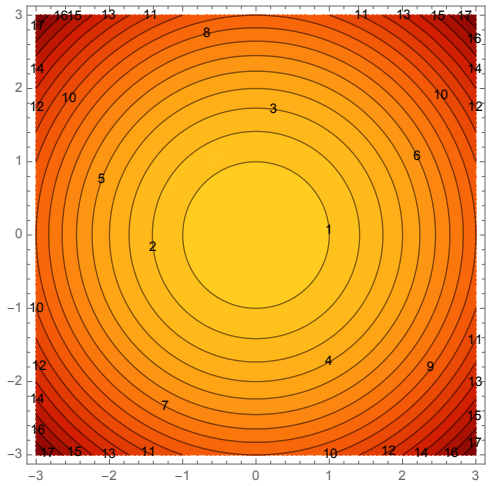
where Q is a symmetric, positive definite matrix.

A matrix is positive definite if $w^\top Q w > 0$ for all $w \neq 0$.

Only one local minimum! [Which is therefore the global minimum.]

[If Q is indefinite, so f is not convex, then the minimum is not always unique and quadratic programming is NP-hard.]

Example: Find maximum margin classifier.



quadratic.pdf [Draw two polygons on these isocontours—one with one active constraint, and one with two—and show the constrained minimum for each polygon. "In an SVM, we are looking for the point in this polygon that's closest to the origin."]

Algs for quadratic programming:

- Simplex-like [commonly used for general-purpose quadratic programs, but not as good for SVMs as the following two algorithms that specifically exploit properties of SVMs]
- Sequential minimal optimization (SMO, used in LIBSVM)
- Coordinate descent (used in LIBLINEAR)

[One clever idea used in SMO is that they do a line search that uses the Hessian, but it's cheap to compute because they don't walk in the direction of steepest descent; instead they walk along just one or a few coordinate axes at a time.]

Convex Program (EE 127/227A/227B)

Convex objective fn + convex inequality constraints.

[What I've given you here is, roughly, a sliding scale of optimization problems of increasing complexity, difficulty, and computation time. But even convex programs are relatively easy to solve. When you're trying to address the needs of real-world applications, it's not uncommon to devise an optimization problem with crazy inequalities and an objective function that's nowhere near convex. These are sometimes very, very hard to solve.]

6 Decision Theory; Generative and Discriminative Models

DECISION THEORY

[Today I'm going to talk about a style of classifier very different from SVMs. The classifiers we'll cover in the next few weeks are based on probability, because sometimes a sample point in feature space doesn't have just one class.]

[Suppose one borrower with income \$30,000 and debt \$15,000 defaults.
another " " " " " " " doesn't default.
So in your feature space, you have two feature vectors at the same point with different classes. Obviously, in that case, you can't draw a decision boundary that classifies all points with 100% accuracy.]

Multiple sample points with different classes could lie at same point:
we want a probabilistic classifier.

Suppose 10% of population has cancer, 90% doesn't.
Probability distributions for calorie intake, $P(X|Y)$: [caps here mean random variables, not matrices.]

calories	(X)	$< 1{,}200$	1,200–1,600	$> 1{,}600$
cancer	$(Y = 1)$	20%	50%	30%
no cancer	$(Y = -1)$	1%	10%	89%

[I made these numbers up. Please don't take them as medical advice.]

Recall: $P(X) = P(X|Y = 1)\,P(Y = 1) + P(X|Y = -1)\,P(Y = -1)$
$P(1{,}200 \le X \le 1{,}600) = 0.5 \times 0.1 + 0.1 \times 0.9 = 0.14$

You meet guy eating $x = 1{,}400$ calories/day. Guess whether he has cancer?

[If you're in a hurry, you might see that 50% of people with cancer eat 1,400 calories, but only 10% of people with no cancer do, and conclude that someone who eats 1,400 calories probably has cancer. But that would be wrong, because that reasoning fails to take the prior probabilities into account.]

Bayes' Theorem:

$$\begin{array}{rcll} & & \downarrow \text{posterior probability} \quad \downarrow \text{prior prob.} & \downarrow \text{for } 1{,}200 \le X \le 1{,}600 \\ P(Y = 1|X) & = & \dfrac{P(X|Y = 1)P(Y = 1)}{P(X)} = \dfrac{0.05}{0.14} & \\ P(Y = -1|X) & = & \dfrac{P(X|Y = -1)P(Y = -1)}{P(X)} = \dfrac{0.09}{0.14} & \text{[These two probs always sum to 1.]} \end{array}$$

$P(\text{cancer} \mid X = 1{,}400 \text{ cals}) = 5/14 \approx 36\%$.

[So we probably shouldn't diagnose cancer.]

[However, we've been assuming that we want to maximize the chance of a correct prediction. But that's not always the right assumption. If you're developing a cheap screening test for cancer, you'd rather have more false positives and fewer false negatives. A false negative might mean somebody misses an early diagnosis and dies of a cancer that could have been treated if caught early. A false positive just means that you spend more money on more accurate tests.]

A loss function $L(z, y)$ specifies badness if true class is y, classifier predicts z.

$$\text{E.g., } L(z,y) = \begin{cases} 1 & \text{if } z = 1, y = -1 \quad \text{false positive is bad} \\ 5 & \text{if } z = -1, y = 1 \quad \text{false negative is BAAAAAD} \\ 0 & \text{if } z = y \end{cases}$$

A 36% probability of loss 5 is worse than a 64% prob. of loss 1,
so we recommend further cancer screening.

Defs: loss fn above is asymmetrical.
The 0-1 loss function is 1 for incorrect predictions, [symmetrical]
0 for correct.

[Another example where you want a very asymmetrical loss function is for spam detection. Putting a good email in the spam folder is much worse than putting spam in your inbox.]

Let $r : \mathbb{R}^d \to \pm 1$ be a decision rule, aka classifier:
a fn that maps a feature vector x to 1 ("in class") or -1 ("not in class").

The risk for r is the expected loss over all values of x, y:

$$\begin{aligned} R(r) &= \mathrm{E}[L(r(X), Y)] \\ &= \sum_x \Big(L(r(x), 1)\, P(Y = 1 | X = x) + L(r(x), -1)\, P(Y = -1 | X = x) \Big)\, P(X = x) \\ &= P(Y = 1) \sum_x L(r(x), 1)\, P(X = x | Y = 1) + P(Y = -1) \sum_x L(r(x), -1)\, P(X = x | Y = -1) \end{aligned}$$

The Bayes decision rule aka Bayes classifier is the r that minimizes $R(r)$; call it r^*.
Assuming $L(z, y) = 0$ for $z = y$:

$$r^*(x) = \begin{cases} 1 & \text{if } L(-1, 1)\, P(Y = 1 | X = x) > L(1, -1)\, P(Y = -1 | X = x), \\ -1 & \text{otherwise} \end{cases}$$

In cancer example, $r^* = 1$ for intakes $\le$ 1,600; $r^* = -1$ for intakes $>$ 1,600.

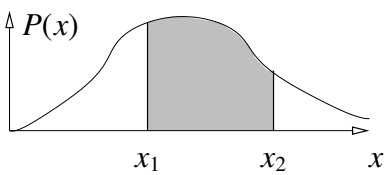
The Bayes risk, aka optimal risk, is the risk of the Bayes classifier.
[In our cancer example, the last expression for risk gives:]

$$R(r^*) = 0.1(5 \times 0.3) + 0.9(1 \times 0.01 + 1 \times 0.1) = 0.249$$

[It is interesting that, if we really know all these probabilities, we really can construct an ideal probabilistic classifier. But in real applications, we rarely know these probabilities; the best we can do is use statistical methods to estimate them.]

Suppose X has a continuous probability density fn (PDF).

Review: [Go back to your CS 70 or stats notes if you don't remember this.]



[Draw this by hand. integrate.pdf]

prob. that random variable $X \in [x_1, x_2] = \int_{x_1}^{x_2} P(x)\, dx$ [shaded area]

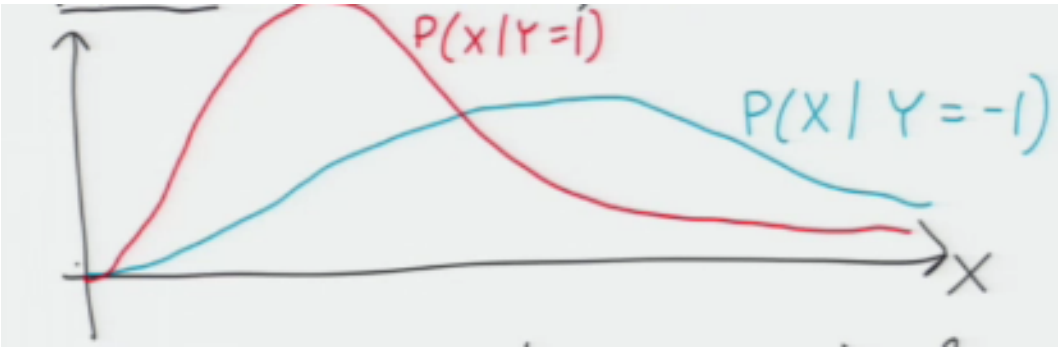
area under whole curve $= 1 = \int_{-\infty}^{\infty} P(x)\,dx$

expected value of $f(x)$: $\mathrm{E}[f(X)] = \int_{-\infty}^{\infty} f(x)P(x)\,dx$

mean $\mu = \mathrm{E}[X] = \int_{-\infty}^{\infty} x\,P(x)\,dx$

variance $\sigma^2 = \mathrm{E}[(X-\mu)^2] = \mathrm{E}[X^2] - \mu^2$

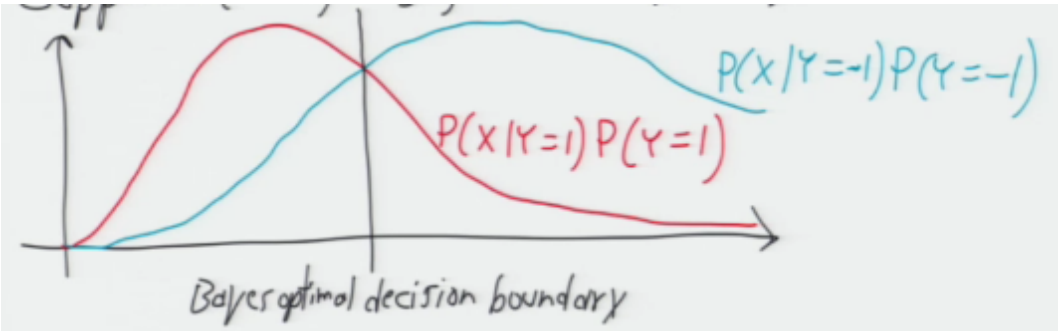
[Perhaps our cancer statistics look like this:]



Draw this figure by hand (cancerconditional.png) [The area under each curve is 1.]

[Let's go back to the 0-1 loss function for a moment. In other words, you want a classifier that maximizes the chance of a correct prediction. The wrong answer would be to look where these two curves cross and make that be the decision boundary. As before, it's wrong because it doesn't take into account the prior probabilities.]

Suppose $P(Y = 1) = 1/3$, $P(Y = -1) = 2/3$, 0-1 loss:



Draw this figure by hand (cancerposterior.png)

[To maximize the chance you'll predict correctly whether somebody has cancer, the Bayes decision rule looks up x on this chart and picks the curve with the highest probability. In this example, that means you pick cancer when x is left of the optimal decision boundary, and no cancer when x is to the right.]

Define risk as before, replacing summations with integrals.

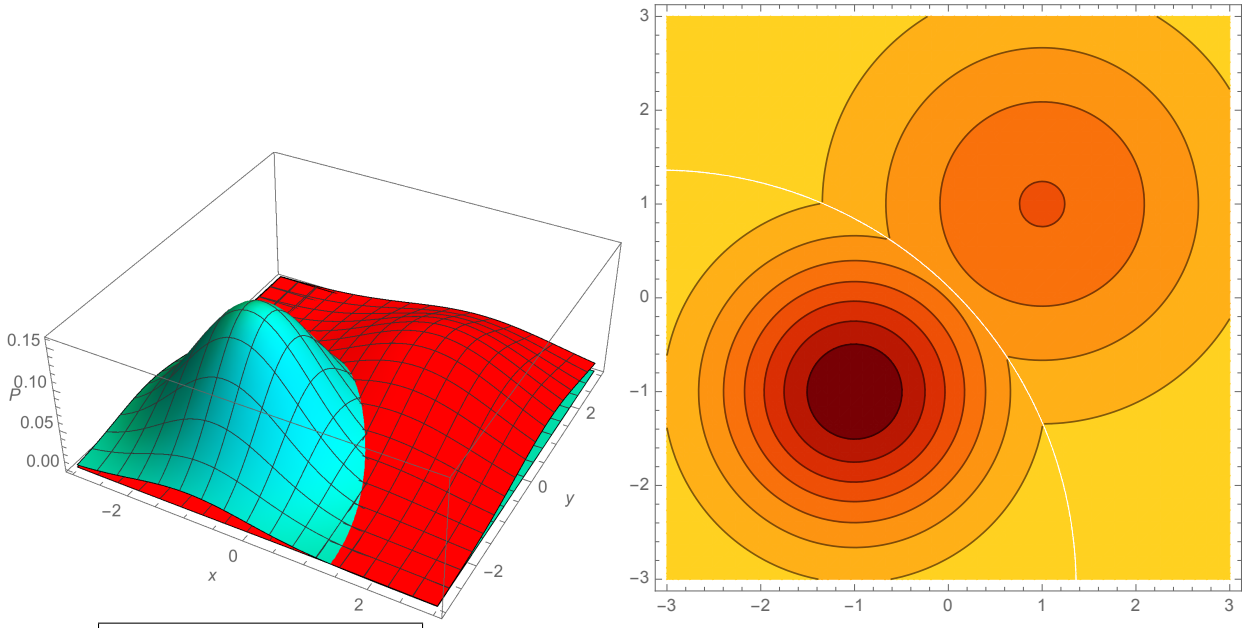
$$\begin{aligned} R(r) &= \mathrm{E}[L(r(X), Y)] \\ &= P(Y = 1) \int L(r(x), 1)\, P(X = x|Y = 1)\, dx + \\ &\quad P(Y = -1) \int L(r(x), -1)\, P(X = x|Y = -1)\, dx \end{aligned}$$

For Bayes decision rule, Bayes Risk is the area under minimum of functions above (shaded). Assuming $L(z, y) = 0$ for $z = y$:

$$R(r^*) = \int \min_{y=\pm 1} L(-y, y)\, P(X = x|Y = y)\, P(Y = y)\, dx$$

[If you want to use an asymmetrical loss function, just scale the vertical reach of each curve accordingly in the figure above.]

If L is 0-1 loss, [then the risk has a particularly nice interpretation:]
$R(r) = P(r(x)$ is wrong) [which makes sense, because R is the expected loss.]
and the Bayes optimal decision boundary is $\{x : \underbrace{P(Y = 1|X = x)}_{\text{predictor fn}} = \underbrace{0.5}_{\text{isovalue}}\}$



qda3d.pdf, qdacontour.pdf [Two different views of the same 2D Gaussians. Note the Bayes optimal decision boundary, which is white at right.]

[Obviously, the accuracy of the probabilities is most important near the decision boundary. Far away from the decision boundary, a bit of error in the probabilities probably wouldn't change the classification.]

[You can also have multi-class classifiers, where each point is in one class among many. The Bayesian approach is a particularly convenient way to generate multi-class classifiers, because you can simply choose whichever class has the greatest posterior probability. Then the decision boundary lies wherever two or more classes are tied for the highest probability.]

3 WAYS TO BUILD CLASSIFIERS

(1) Generative models (e.g. LDA) [We'll learn about LDA next lecture.]
- Assume sample points come from probability distributions, different for each class.
- Guess form of distributions
- For each class C, fit distribution parameters to class C points, giving $P(X|Y = \text{C})$
- For each C, estimate $P(Y = \text{C})$
- Bayes' Theorem gives $P(Y|X)$
- If 0-1 loss, pick class C that maximizes $P(Y = \text{C}|X = x)$ [posterior probability]
  equivalently, maximizes $P(X = x|Y = \text{C})\, P(Y = \text{C})$

(2) Discriminative models (e.g. logistic regression) [We'll learn about logistic regression in a few weeks.]

- Model $P(Y|X)$ directly

(3) Find decision boundary (e.g. SVM)
- Model $r(x)$ directly (no posterior)

Advantage of (1 & 2): $P(Y|X)$ tells you probability your guess is wrong
[This is something SVMs don't do.]
Advantage of (1): you can diagnose outliers: $P(X)$ is very small
Disadvantages of (1): often hard to estimate distributions accurately;
real distributions rarely match standard ones.

[What I've written here doesn't actually define the phrases "generative model" or "discriminative model." The proper definitions accord with the way statisticians think about models. A generative model is a full probabilistic model of all variables, whereas a discriminative model provides a model only for the target variables.]

[It's important to remember that we rarely know precisely the value of any of these probabilities. There is usually error in all of these probabilities, and in a generative model those errors can get compounded when we apply Bayes' Theorem to estimate $P(Y|X)$. In practice, generative models are most popular when you have phenomena that are really well fitted by the normal distribution.]

7 Gaussian Discriminant Analysis (including QDA and LDA)

GAUSSIAN DISCRIMINANT ANALYSIS

Fundamental assumption: each class comes from normal distribution (Gaussian).

$$X \sim N(\mu, \sigma^2) : P(x) = \frac{1}{(\sqrt{2\pi}\sigma)^d} \exp\left(-\frac{|x-\mu|^2}{2\sigma^2}\right)$$

For each class C, suppose we estimate mean μ_C, variance σ_C^2, and prior $\pi_C = P(Y = C)$.

Given x, Bayes decision rule $r^*(x)$ returns class C that maximizes $P(X = x|Y = C)\,\pi_C$.

$\ln z$ is monotonically increasing for $z > 0$, so it is equivalent to maximize

$$Q_C(x) = \ln((\sqrt{2\pi})^d\, P(x)\,\pi_C) = -\frac{|x-\mu_C|^2}{2\sigma_C^2} - d\ln\sigma_C + \ln\pi_C$$

↑ quadratic in x. ↑ normal distribution, estimates $P(X = x|Y = C)$

[In a 2-class problem, you can also incorporate an asymmetrical loss function the same way we incorporate the prior π_C. In a multi-class problem, it gets a bit more complicated, because the penalty for guessing wrong might depend not just on the true class, but also on the wrong guess.]

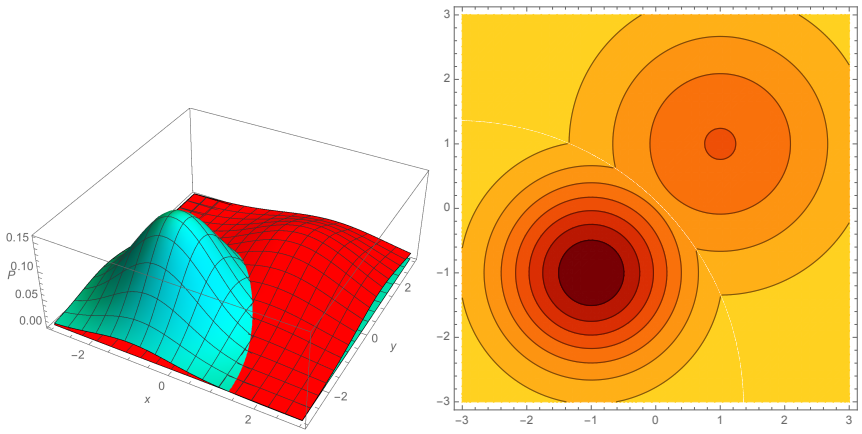
Quadratic Discriminant Analysis (QDA)

Suppose only 2 classes C, D. Then

$$r^*(x) = \begin{cases} C & \text{if } Q_C(x) - Q_D(x) > 0, \\ D & \text{otherwise} \end{cases}$$

Prediction fn is quadratic in x. Bayes decision boundary is $Q_C(x) - Q_D(x) = 0$.

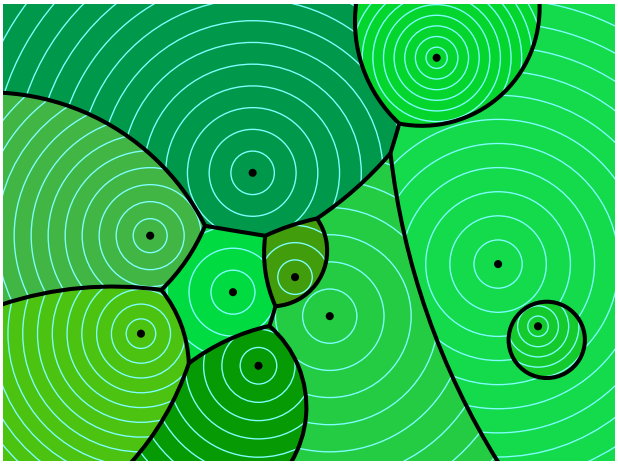
- In 1D, B.d.b. may have 1 or 2 points. [Solution to a quadratic equation]
- In d-D, B.d.b. is a quadric. [In 2D, that's a conic section]



qda3d.pdf, qdacontour.pdf [The same example I showed during the last lecture.]

[The equations I wrote down above can apply to a one-dimensional feature space, or they could apply equally well to a multi-dimensional feature space with isotropic, spherical Gaussians. In the latter case, x and μ are vectors, but the variance σ is a scalar. Next lecture we'll look at anisotropic Gaussians where the variance is different along different directions.]

[QDA works very nicely with more than 2 classes.]



multiplicative.pdf [The feature space gets partitioned into regions. In two or more dimensions, you typically wind up with multiple decision boundaries that adjoin each other at joints. It looks like a sort of Voronoi diagram. In fact, it's a special kind of Voronoi diagram called a multiplicatively weighted Voronoi diagram.]

[You might not be satisfied with just knowing how each point is classified. One of the great things about QDA is that you can also determine the probability that your classification is correct. Let's work that out.]

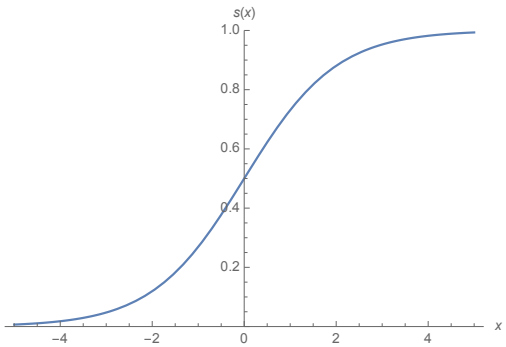
To recover posterior probabilities in 2-class case, use Bayes:

$$P(Y = \mathrm{C}|X) = \frac{P(X|Y = \mathrm{C})\,\pi_\mathrm{C}}{P(X|Y = \mathrm{C})\,\pi_\mathrm{C} + P(X|Y = \mathrm{D})\,\pi_\mathrm{D}}$$

recall $e^{Q_\mathrm{C}(x)} = (\sqrt{2\pi})^d\, P(x)\,\pi_\mathrm{C}$ [by definition of Q_C]

$$\begin{aligned} P(Y = \mathrm{C}|X = x) &= \frac{e^{Q_\mathrm{C}(x)}}{e^{Q_\mathrm{C}(x)} + e^{Q_\mathrm{D}(x)}} = \frac{1}{1 + e^{Q_\mathrm{D}(x) - Q_\mathrm{C}(x)}} \\ &= s(Q_\mathrm{C}(x) - Q_\mathrm{D}(x)), \quad \text{where} \end{aligned}$$

$$s(\gamma) = \frac{1}{1 + e^{-\gamma}} \quad \Leftarrow \text{logistic fn aka sigmoid fn}$$



logistic.pdf [The logistic function. Write beside it:] $s(0) = \frac{1}{2}$, $s(\infty) \to 1$, $s(-\infty) \to 0$, monotonically increasing.

[We interpret $s(0) = \frac{1}{2}$ as saying that on the decision boundary, there's a 50% chance of class C and a 50% chance of class D.]

Linear Discriminant Analysis (LDA)

[Less likely to overfit than QDA.]

Fundamental assumption: all the Gaussians have same variance σ.

[The equations simplify nicely in this case.]

$$Q_{\mathrm{C}}(x) - Q_{\mathrm{D}}(x) = \underbrace{\frac{(\mu_{\mathrm{C}} - \mu_{\mathrm{D}}) \cdot x}{\sigma^2}}_{w \cdot x} \underbrace{- \frac{\mu_{\mathrm{C}}^2 - \mu_{\mathrm{D}}^2}{2\sigma^2} + \ln \pi_{\mathrm{C}} - \ln \pi_{\mathrm{D}}}_{+\alpha}$$

[The quadratic terms in Q_{C} and Q_{D} cancelled each other out!]

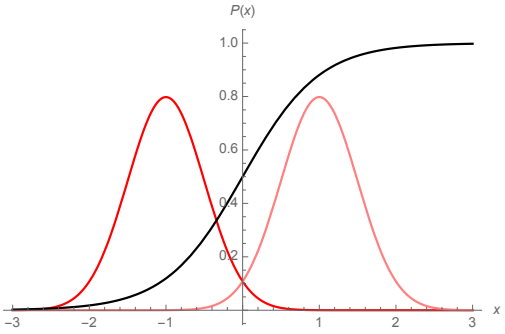
Now it's a linear classifier! Choose C that maximizes linear discriminant fn

$$\frac{\mu_{\mathrm{C}} \cdot x}{\sigma^2} - \frac{\mu_{\mathrm{C}}^2}{2\sigma^2} + \ln \pi_{\mathrm{C}} \qquad \text{[works for any \# of classes]}$$

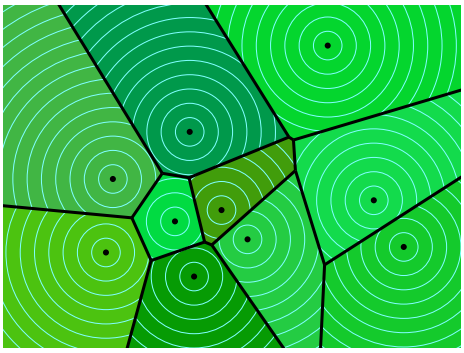
In 2-class case: decision boundary is $w \cdot x + \alpha = 0$

Bayes posterior is $P(Y = \mathrm{C} | X = x) = s(w \cdot x + \alpha)$

[The effect of "$w \cdot x + \alpha$" is to scale and translate the logistic fn in x-space. It's a linear transformation.]



lda1d.pdf [Two 1D Gaussians (red) and the logistic function (black). The logistic function arises as the right Gaussian divided by the sum of the Gaussians.]



voronoi.pdf [When you have many classes, their LDA decision boundaries form a classical Voronoi diagram. All the Gaussians here have the same width.]

$$\text{If } \pi_{\mathrm{C}} = \pi_{\mathrm{D}} = \frac{1}{2} \quad \Rightarrow \quad (\mu_{\mathrm{C}} - \mu_{\mathrm{D}}) \cdot x - (\mu_{\mathrm{C}} - \mu_{\mathrm{D}}) \cdot \left(\frac{\mu_{\mathrm{C}} + \mu_{\mathrm{D}}}{2} \right) = 0$$

This is the centroid method!

MAXIMUM LIKELIHOOD ESTIMATION OF PARAMETERS (Ronald Fisher, circa 1912)

[To use Gaussian discriminant analysis, we must first fit Gaussians to the data. Before I talk about fitting Gaussians, I want to start with a simpler example. It's easier to understand maximum likelihood if we consider a discrete distribution first; then a continuous distribution later.]

Let's flip biased coins! Heads with probability p; tails w/prob. $1 - p$.

10 flips, 8 heads, 2 tails. What is most likely value of p?

Binomial distribution: $X \sim B(n, p)$

$$P[X = x] = \binom{n}{x} p^x (1-p)^{n-x}$$

Our example: $n = 10$,

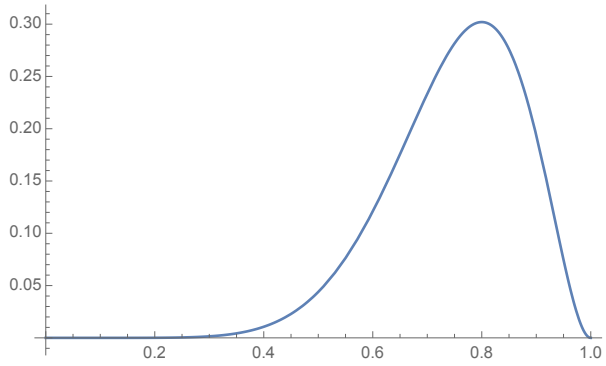
$$P[X = 8] = 45p^8 (1-p)^2 \qquad \stackrel{\text{def}}{=} \mathcal{L}(p)$$

Probability of 8 heads in 10 flips:
written as a fn $\mathcal{L}(p)$ of distribution parameter(s), this is the likelihood fn.

Maximum likelihood estimation (MLE): A method of estimating the parameters of a statistical model by picking the params that maximize the likelihood fn.

[Let's phrase it as an optimization problem.]

Find p that maximizes $\mathcal{L}(p)$.



binomlikelihood.pdf [Graph of $\mathcal{L}(p)$ for this example.]

Solve this example by setting derivative = 0:

$$\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}p} = 360p^7(1-p)^2 - 90p^8(1-p) = 0$$

$$\Rightarrow \quad 4(1-p) - p = 0 \quad \Rightarrow \quad p = 0.8$$

[It shouldn't seem surprising that a coin that comes up heads 80% of the time is the coin most likely to produce 8 heads in 10 flips.]

[Note: $\frac{\mathrm{d}^2\mathcal{L}}{\mathrm{d}p^2} \doteq -18.9 < 0$ at $p = 0.8$, confirming it's a maximum.]

Likelihood of a Gaussian

Given sample points $x_1, x_2, \ldots, x_n$ (scalars or vectors), find best-fit Gaussian.

[Let's do this with a normal distribution instead of a binomial distribution. If you generate a random point from a normal distribution, what is the probability that it will be exactly at the mean of the Gaussian?]

[Zero. So it might seem like we have a bit of a problem here. With a continuous distribution, the probability of generating any particular point is zero. But we're just going to ignore that and do "likelihood" anyway.]

Likelihood of generating these points is

$$\mathcal{L}(\mu, \sigma; x_1, \ldots, x_n) = P(x_1)P(x_2)\cdots P(x_n)$$ [How do we maximize this?]

The log likelihood $\ell(\cdot)$ is the ln of the likelihood $\mathcal{L}(\cdot)$.
Maximizing likelihood $\Leftrightarrow$ maximizing log-likelihood.

$$\ell(\mu, \sigma; x_1, ..., x_n) = \ln P(x_1) + \ln P(x_2) + ... + \ln P(x_n)$$

Want to set $\nabla_\mu \ell = 0$, $\dfrac{\partial \ell}{\partial \sigma} = 0$

$$\ln P(x) = -\frac{|x-\mu|^2}{2\sigma^2} - d \ln \sqrt{2\pi} - d \ln \sigma$$ [ln of normal distribution]

$$\nabla_\mu \ell = \sum_{i=1}^{n} \frac{x_i - \mu}{\sigma^2} = 0 \quad \Rightarrow \quad \hat{\mu} = \frac{1}{n}\sum_{i=1}^{n} x_i$$ [The hats ˆ mean "estimated"]

$$\frac{\partial \ell}{\partial \sigma} = \sum_{i=1}^{n} \frac{|x_i - \mu|^2 - d\sigma^2}{\sigma^3} = 0 \quad \Rightarrow \quad \hat{\sigma}^2 = \frac{1}{dn}\sum_{i=1}^{n} |x_i - \mu|^2$$

We don't know μ exactly, so substitute $\hat{\mu}$ for μ to compute $\hat{\sigma}$.

In short, we use mean & variance of points in class C to estimate mean & variance of Gaussian for class C.

For QDA: estimate mean & variance of each class as above
& estimate the priors (for each class C):

$$\hat{\pi}_\mathrm{C} = \frac{n_\mathrm{C}}{\sum_\mathrm{D} n_\mathrm{D}} \quad \Leftarrow \quad \text{total sample points in all classes}$$ [$\hat{\pi}_\mathrm{C}$ is the coin flip parameter!]

For LDA: same mean & priors; one variance for all classes:

$$\hat{\sigma}^2 = \frac{1}{dn}\sum_{\mathrm{C}} \sum_{\{i:y_i=\mathrm{C}\}} |x_i - \mu_\mathrm{C}|^2$$

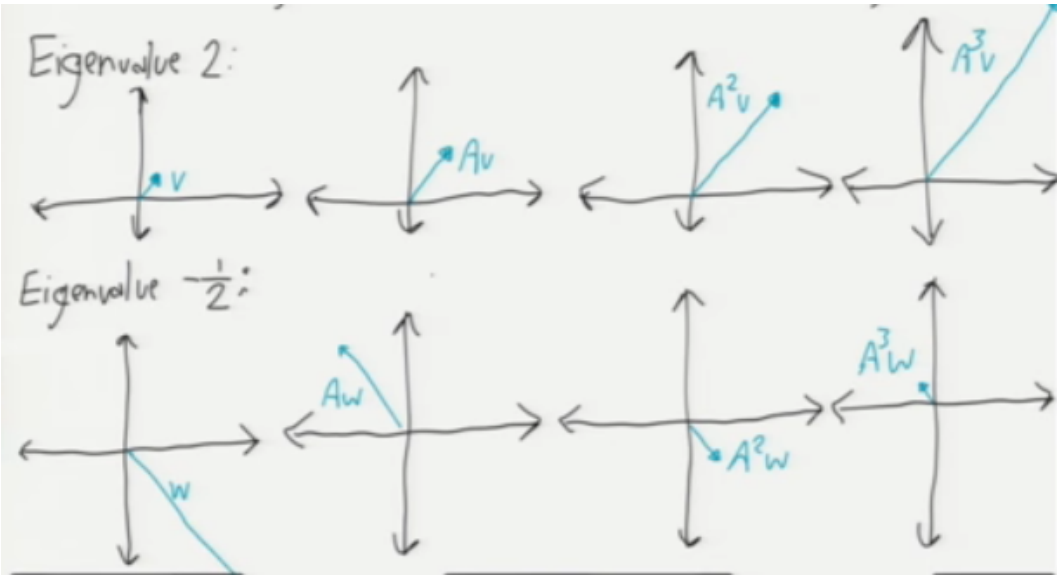
8 Eigenvectors and the Anisotropic Gaussian Distribution

EIGENVECTORS

[I don't know if you were properly taught about eigenvectors here at Berkeley, but I sure don't like the way they're taught in most linear algebra books. So I'll start with a review. You all know the definition of an eigenvector:]

Given square matrix A, if $Av = \lambda v$ for some vector $v \neq 0$, scalar λ, then
v is an eigenvector of A and λ is the associated eigenvalue of A.

[But what does that mean? It means that v is a magical vector that, after being multiplied by A, still points in the *same direction*, or in exactly the *opposite direction*.]



Draw this figure by hand (eigenvectors.png)

[For most matrices, most vectors don't have this property. So the ones that do are special, and we call them eigenvectors.]
[Clearly, when you scale an eigenvector, it's still an eigenvector. Only the direction matters, not the length. Let's look at a few consequences.]

Theorem: if v is eigenvector of A w/eigenvalue λ,
then v is eigenvector of A^k w/eigenvalue λ^k [we will use this later]

Proof: $A^2v = A(\lambda v) = \lambda^2 v$, etc.

Theorem: moreover, if A is invertible,
then v is eigenvector of A^{-1} w/eigenvalue $1/\lambda$

Proof: $A^{-1}v = \frac{1}{\lambda}A^{-1}Av = \frac{1}{\lambda}v$ [look at the figures above, but go from right to left.]

[Stated simply: When you invert a symmetric matrix, the eigenvectors don't change, but the eigenvalues get inverted. When you square a symmetric matrix, the eigenvectors don't change, but the eigenvalues get squared.]

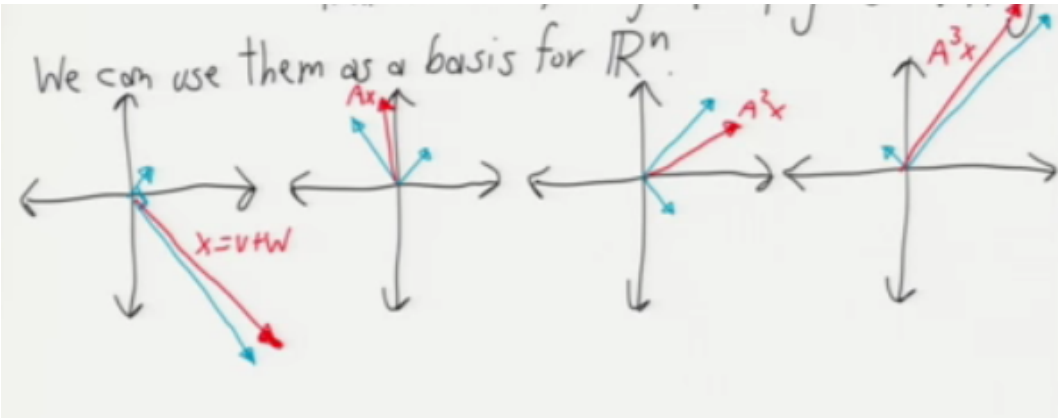
[Those theorems are pretty obvious. The next theorem is not obvious at all. But it's going to be very useful for understanding the effect of a symmetric matrix on a vector that is *not* an eigenvector.]

Spectral Theorem: every symmetric $n \times n$ matrix has n eigenvectors that are mutually orthogonal, i.e., $v_i^\top v_j = 0$ for all $i \neq j$

[This takes about a page of math to prove.
One minor detail is that a matrix can have more than n eigenvector directions. If two eigenvectors happen to have the same eigenvalue, then every linear combination of those eigenvectors is also an eigenvector. Then you have infinitely many eigenvector directions, but they all span the same plane. So you just arbitrarily pick two vectors in that plane that are orthogonal to each other. By contrast, the set of eigenvalues is always uniquely determined by a matrix, including the multiplicity of the eigenvalues.]

We can use them as a basis for $\mathbb{R}^n$.

[Now we can ask, what happens to a vector that *isn't* an eigenvector when you apply a symmetric matrix to it? Express that vector as a linear combination of eigenvectors, and look at each eigenvector separately.]



Draw this figure by hand (basis.png)

[This vector, x, is the sum of the two eigenvectors I've already shown you. Every time we apply A to this vector, it changes direction. But we can understand it by writing it as a sum of components that don't change direction.]

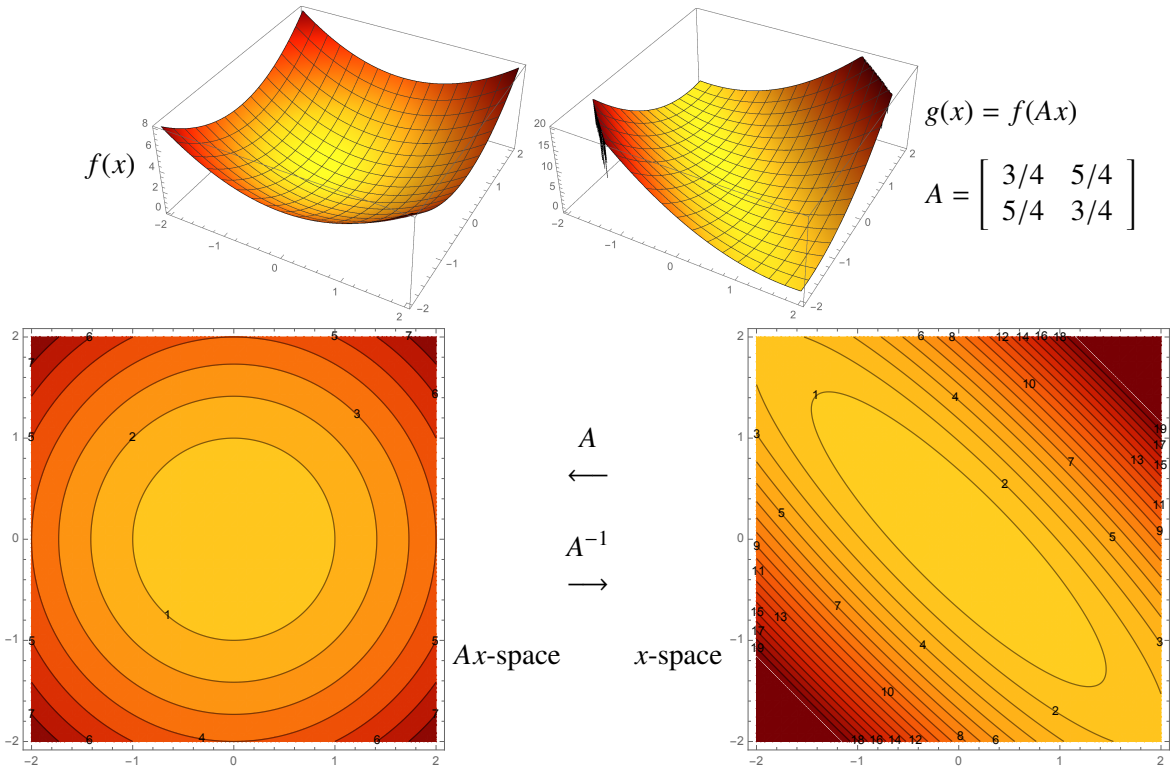
Write x as linear combo of eigenvectors:

$$\begin{aligned} x &= \alpha v + \beta w \\ A^k x &= \lambda_v^k \alpha v + \lambda_w^k \beta w \end{aligned}$$

Ellipsoids

[Now, let's look what happens to a quadratic function when we apply a symmetric matrix to the space, with these two eigenvectors and eigenvalues.]

$$\begin{aligned} f(x) &= x^\top x && \Leftarrow \text{quadratic; isotropic; isosurfaces are spheres} \\ g(x) &= f(Ax) && \Leftarrow A \text{ symmetric} \\ &= x^\top A^2 x && \Leftarrow A \text{ quadratic form of the matrix } A^2 \\ & && \quad \text{anisotropic; isosurfaces are ellipsoids} \end{aligned}$$



circlebowl.pdf, ellipsebowl.pdf, circles.pdf, ellipses.pdf

[Both figures at left graph $f(x) = |x|^2$, and both figures at right graph $g(x) = |Ax|^2$.
(Draw the stretch direction $(1, 1)$ with eigenvalue 2 and the shrink direction $(1, -1)$ with eigenvalue $-\frac{1}{2}$ on the ellipses at bottom right.)]

[Here's how I think of this: we stretched the plane on the right along the direction with eigenvalue 2, and shrunk the plane along the direction with eigenvalue $-1/2$; then we drew circular isocontours, like on the left; then we undid the stretching and let the plane spring back to its original shape. So the circle turned into an ellipse when the plane sprang back.]

[Looking at the quadratic form is one of the best ways to visually understand symmetric matrices and their eigenvectors and eigenvalues.]

$g(x) = 1$ is an ellipsoid with axes $v_1, v_2, \ldots, v_n$ and
radii $1/\lambda_1, 1/\lambda_2, \ldots, 1/\lambda_n$

because if v_i has length $1/\lambda_i$, $g(v_i) = f(Av_i) = f(\lambda_i v_i) = 1$
$\Rightarrow$ v_i lies on the ellipsoid

[The reason the ellipsoid radii are the reciprocals of the eigenvalues is that we're stretching the plane by the eigenvalues, then drawing the spheres, then letting the plane spring back to its original shape. When the plane springs back, each axis of the spheres gets scaled by 1/eigenvalue.]

bigger eigenvalue $\Leftrightarrow$ steeper slope $\Leftrightarrow$ shorter ellipsoid radius
[$\uparrow$ actually bigger curvature; the slope varies along the axis]

Alternate interpretation: ellipsoids are spheres in distance metric A^2

Call $M = A^2$ a metric tensor because
the [Euclidean] distance between points x & z in stretched space is

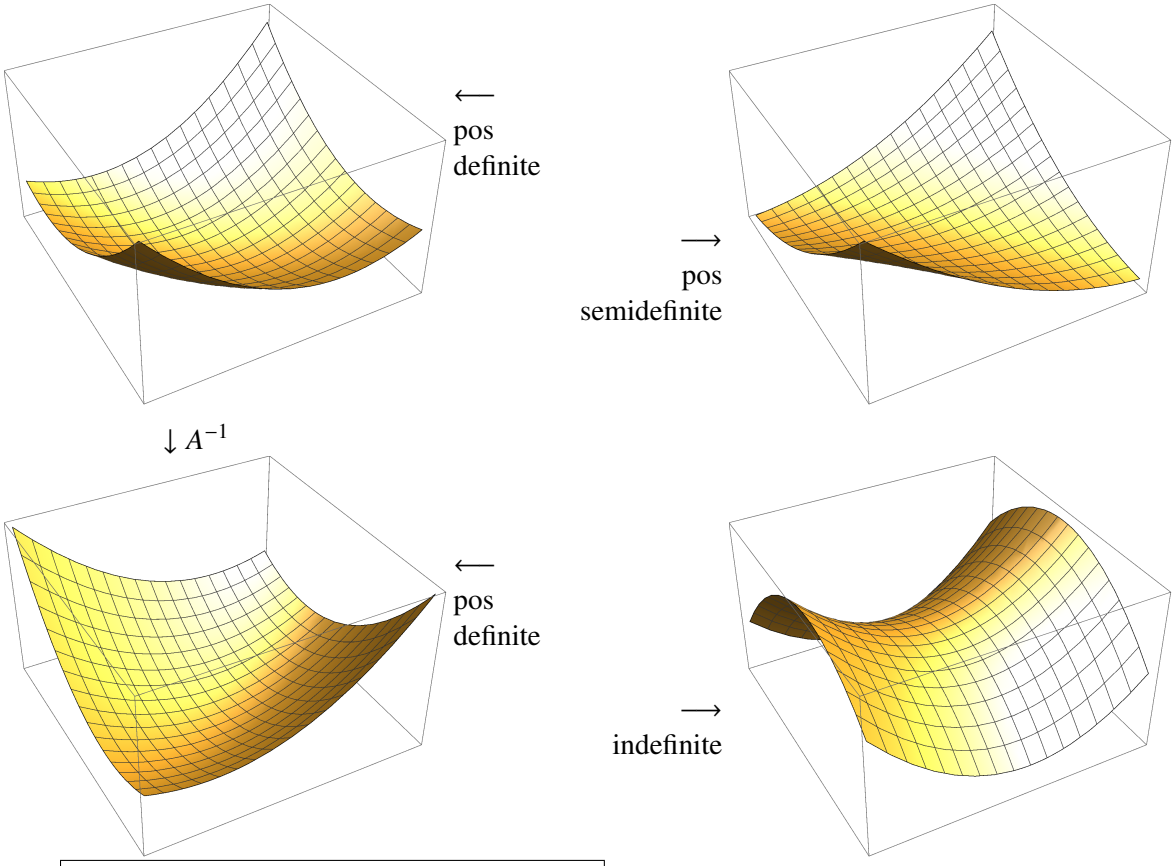
$$d(x, z) = |Ax - Az| = \sqrt{(x - z)^\top M(x - z)}$$

[This is the Euclidean distance in the stretched space, but let's think of it as an alternative metric for measuring distances in the original space. It's a kind of distance from x to z that's different from the Euclidean distance.]

[I'm calling M a "tensor" because that's standard usage in Riemannian geometry, but don't worry about what "tensor" means. For our purposes, it's a matrix.]

Ellipsoids are "spheres" in this metric: $\{x : d(x, \text{center}) = \text{isovalue}\}$

A square matrix B is		
	positive definite	if $w^\top Bw > 0$ for all $w \neq 0$.
		$\Leftrightarrow$ all eigenvalues positive
	positive semidefinite	if $w^\top Bw \geq 0$ for all w.
		$\Leftrightarrow$ all eigenvalues nonnegative
	indefinite	if +ve eigenvalue & −ve eigenvalue
	invertible	if no zero eigenvalue



posdef.pdf, possemi.pdf, invert.pdf, indef.pdf

[(Show this figure on a separate "whiteboard.") Examples of quadratic forms for positive definite, positive semidefinite, and indefinite matrices. Positive eigenvalues correspond to axes where the curvature goes up; negative eigenvalues correspond to axes where the curvature goes down. (Draw the eigenvector directions, and draw the flat trough in the positive semidefinite bowl.) The lower left quadratic form shows what happens that when you invert a symmetric matrix: you effectively replace the eigenvalues with their reciprocals.]

Metric tensors must be symmetric +ve definite (SPD).

[Remember that $M = A^2$, so M's eigenvalues are the squares of the eigenvalues of A, so the eigenvalues must be nonnegative and M is positive semidefinite. But if M has a zero eigenvalue, its distance function is not a "metric." To have a metric, you must have a strictly positive definite M. If you have eigenvalues of zero, the isosurfaces are cylinders instead of ellipsoids.]

Special case: M & A are diagonal $\Leftrightarrow$ eigenvectors are coordinate axes
$\Leftrightarrow$ ellipsoids are axis-aligned

[Draw axis-aligned isocontours for a diagonal metric.]

Building a Quadratic w/Specified Eigenvectors/values

[I, personally, find the process of going from eigenvectors and eigenvalues to a matrix and some ellipsoids to be more intuitive than the reverse. So let's do that. Suppose you know which ellipsoid axes you want to use, and you know what ellipsoid radius or stretch factor you want to use along each axis.]

Choose n mutually orthogonal **unit** n-vectors $v_1, \ldots, v_n$ [so they specify an orthonormal coordinate system]
Let $V = [v_1 \quad v_2 \quad \ldots \quad v_n]$ $\Leftarrow n \times n$ matrix
Observe: $V^\top V = I$ [off-diagonal 0's because the vectors are orthogonal]
[diagonal 1's because they're unit vectors]

$\Rightarrow \quad V^\top = V^{-1} \quad \Rightarrow \quad VV^\top = I$
V is orthonormal matrix: acts like rotation (or reflection)

Choose some inverse radii λ_i:

$$\text{Let } \Lambda = \begin{bmatrix} \lambda_1 & 0 & \ldots & 0 \\ 0 & \lambda_2 & & 0 \\ \vdots & & \ddots & \vdots \\ 0 & 0 & \ldots & \lambda_n \end{bmatrix} \qquad \text{[diagonal matrix of eigenvalues]}$$

Theorem: $A = V\Lambda V^\top = \sum_{i=1}^{n} \lambda_i \underbrace{v_i v_i^\top}$ has chosen eigenvectors/values [Clearly, A is symmetric]
outer product: $n \times n$ matrix, rank 1

Proof: Equivalent to $AV = V\Lambda$ $\Leftarrow$ definition of eigenvectors! (in matrix form)

[Draw arrow to "$A = V\Lambda V^\top$"] This is a matrix factorization called the eigendecomposition.
Λ is the diagonalized version of A.
$V^\top$ rotates the ellipsoid to be axis-aligned.

This is also a recipe for building quadratics with axes v_i, radii $1/\lambda_i$

Observe: $M = A^2 = V\Lambda V^\top V\Lambda V^\top = V\Lambda^2 V^\top$ [so M has same eigenvectors as A, squared eigenvalues]

Given SPD metric tensor M, we can find a symmetric square root $A = M^{1/2}$:

compute eigenvectors/values of M
take square roots of M's eigenvalues
reassemble matrix A

[The first step—breaking a matrix down to its eigenvectors and eigenvalues—is much harder than the last step—building up a new matrix from its eigenvectors and eigenvalues. But I think that the latter process helps take a lot of the mystery out of eigenvectors.]

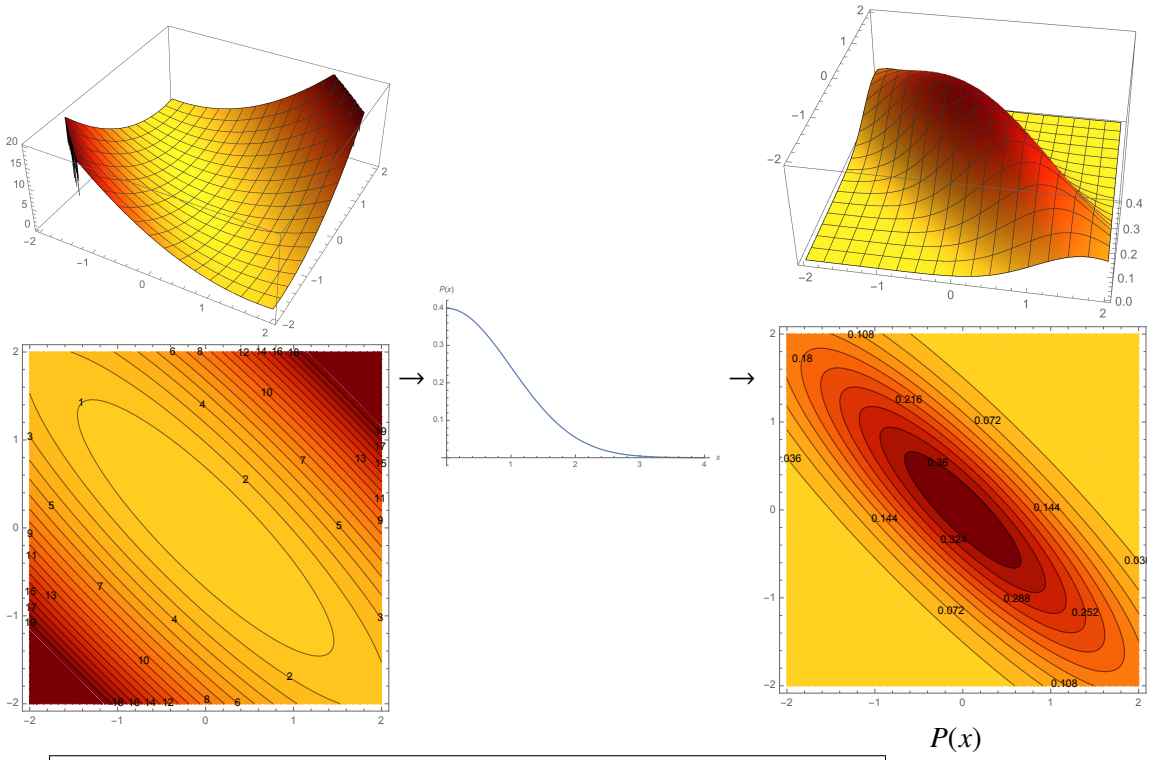
ANISOTROPIC GAUSSIANS

$$X \sim N(\mu, \Sigma)$$

$$P(x) = \frac{1}{(\sqrt{2\pi})^d \sqrt{|\Sigma|}} \exp\left(-\frac{1}{2}(x-\mu)^\top \Sigma^{-1} (x-\mu)\right)$$

↑ determinant of Σ

Σ is the $d \times d$ SPD covariance matrix.
Σ^{-1} is the $d \times d$ SPD precision matrix; serves as metric tensor.



$P(x)$

ellipsebowl.pdf, ellipses.pdf, gauss.pdf, gauss3d.pdf, gausscontour.pdf

[(Show this figure on a separate "whiteboard" for easy reuse next lecture.) A paraboloid (left) becomes a bivariate Gaussian (right) after you compose it with the univariate Gaussian (center).]

9 The Anisotropic Gaussian Distribution, QDA, and LDA

ANISOTROPIC GAUSSIANS

$X \sim N(\mu, \Sigma)$ $\Leftarrow X$ is a random d-vector with mean μ

$$P(x) = \frac{1}{(\sqrt{2\pi})^d \sqrt{|\Sigma|}} \exp\left(-\frac{1}{2}(x-\mu)^\top \Sigma^{-1} (x-\mu)\right)$$

$\uparrow$ determinant of Σ

Σ is the $d \times d$ SPD covariance matrix.
Σ^{-1} is the $d \times d$ SPD precision matrix; serves as metric tensor.

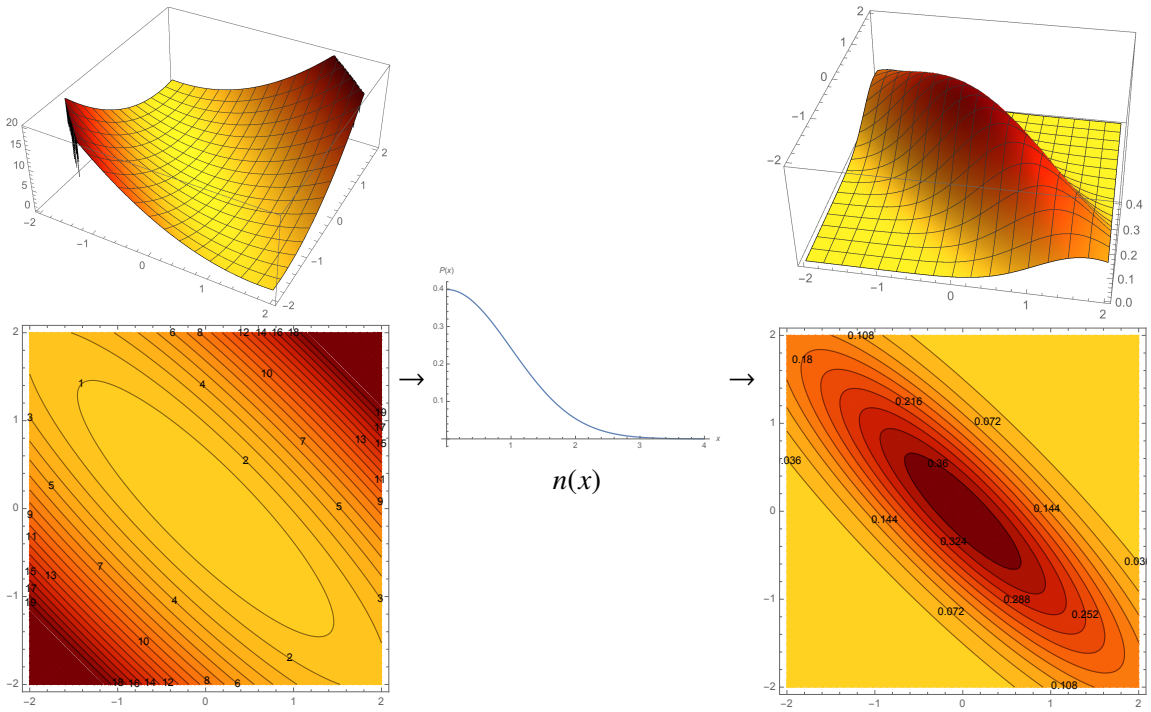
Write $P(x) = n(q(x))$, where $q(x) = (x-\mu)^\top \Sigma^{-1} (x-\mu)$

$\uparrow$ $\uparrow$

$\mathbb{R} \to \mathbb{R}$, exponential $\mathbb{R}^d \to \mathbb{R}$, quadratic

[Now $q(x)$ is a function we understand—it's just a quadratic bowl centered at μ whose curvature is represented by the metric tensor Σ^{-1}. $q(x)$ is the squared distance from μ to x under this metric. The other function $n(\cdot)$ is like a 1D Gaussian with a different normalization factor. It is helpful to understand that this mapping $n(\cdot)$ does not change the isosurfaces.]

Principle: given $n : \mathbb{R} \to \mathbb{R}$, isosurfaces of $n(q(x))$ are same as $q(x)$ (different isovalues), except that some might be "combined" [if n maps them to the same value]



ellipsebowl.pdf, ellipses.pdf, gauss.pdf, gauss3d.pdf, gausscontour.pdf

[A paraboloid (left) becomes a bivariate Gaussian (right) after you compose it with the univariate Gaussian (center).]

covariance: $\mathrm{Cov}(X, Y) = \mathrm{E}[(X - \mathrm{E}[X])\,(Y - \mathrm{E}[Y])^\top] = \mathrm{E}[XY^\top] - \mu_X \mu_Y^\top$
$\mathrm{Var}(X) = \mathrm{Cov}(X, X)$
[These two definitions hold for both vectors and scalars.]

For a Gaussian, one can show $\mathrm{Var}(X) = \Sigma$.
[. . . by integrating the expectation in anisotropic spherical coordinates. It's a painful integral.]
Hence

$$\Sigma = \begin{bmatrix} \mathrm{Var}(X_1) & \mathrm{Cov}(X_1, X_2) & \dots & \mathrm{Cov}(X_1, X_d) \\ \mathrm{Cov}(X_2, X_1) & \mathrm{Var}(X_2) & & \mathrm{Cov}(X_2, X_d) \\ \vdots & & \ddots & \vdots \\ \mathrm{Cov}(X_d, X_1) & \mathrm{Cov}(X_d, X_2) & \dots & \mathrm{Var}(X_d) \end{bmatrix}$$

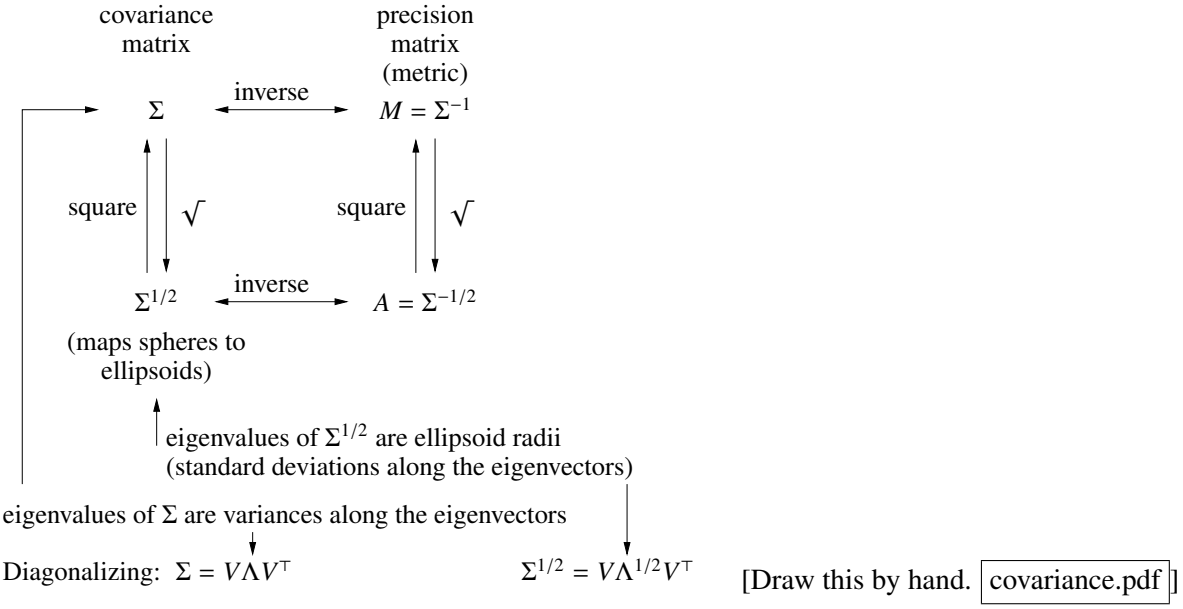
[An important point is that statisticians didn't just arbitrarily decide to call this thing a covariance matrix. Rather, statisticians discovered that if you find the covariance of the normal distribution by integration, it turns out that the covariance matrix happens to be the inverse of the metric tensor. This is a happy discovery; it's rather elegant.]

[Observe that Σ is symmetric, as $\mathrm{Cov}(X_i, X_j) = \mathrm{Cov}(X_j, X_i)$.]

X_i, X_j independent $\quad\Rightarrow\quad \mathrm{Cov}(X_i, X_j) = 0$
$\mathrm{Cov}(X_i, X_j) = 0$ AND joint normal distribution $\quad\Rightarrow\quad X_i, X_j$ independent
all features pairwise independent $\quad\Rightarrow\quad \Sigma$ is diagonal
Σ is diagonal $\quad\Leftrightarrow\quad$ axis-aligned Gaussian; squared radii on the diagonal
$\quad\Leftrightarrow\quad \underbrace{P(X)}_{\text{multivariate}} = \underbrace{P(X_1)P(X_2)\cdots P(X_d)}_{\text{each univariate}}$

[So when the features are independent, you can write the multivariate Gaussian as a product of univariate Gaussians. When they aren't, you can do a change of coordinates to the eigenvector coordinate system, and write it as a product of univariate Gaussians in those coordinates.]

[It's tricky to keep track of the relationships between the matrices, so here's a handy chart.]



[Draw this by hand. covariance.pdf]

[Remember that all four of these matrices have the same eigenvectors V. Remember that when you take the inverse or square or square root of an SPD matrix, you do the same to its eigenvalues. So the ellipsoid radii, being the eigenvalues of $\Sigma^{1/2}$, are the square roots of the eigenvalues of Σ; moreover, they are the inverse square roots of the eigenvalues of the precision matrix (metric).]

[Keep this chart handy when you do Homework 3.]

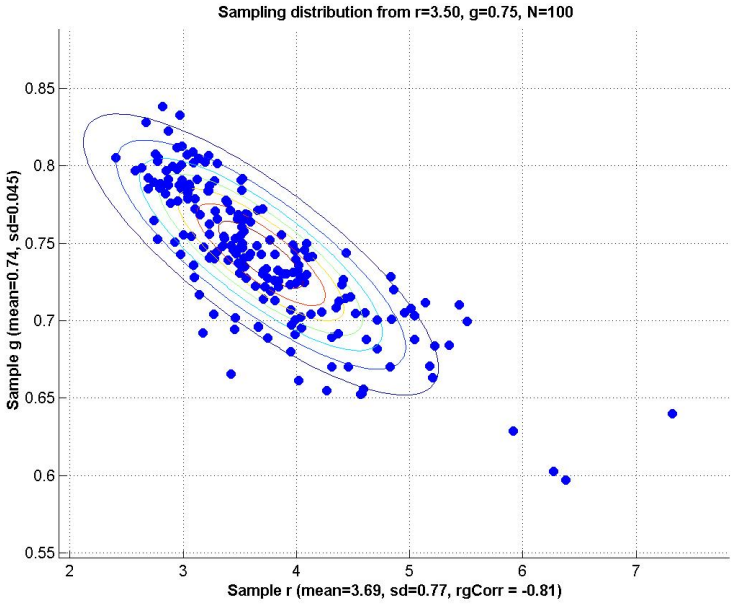
Maximum likelihood estimation for anisotropic Gaussians

Given sample points $x_1, \ldots, x_n$ and classes $y_1, \ldots, y_n$, find best-fit Gaussians.

[Once again, we want to fit the Gaussian that maximizes the likelihood of generating the sample points in a specified class. This time I won't derive the maximum-likelihood Gaussian; I'll just tell you the answer.]

For QDA:

$$\hat{\Sigma}_C = \frac{1}{n_C} \sum_{i:y_i=C} \underbrace{(x_i - \mu_C)(x_i - \mu_C)^\top}_{\text{outer product matrix}} \qquad \Leftarrow \text{ conditional covariance for points in class C}$$



maxlike.jpg Maximum likelihood estimation takes these points and outputs this Gaussian.

Priors π_C, means μ_C: same as before
[Priors are class sample points / total sample points; means are per-class sample point means]
$\hat{\Sigma}_C$ is positive semidefinite, but not always definite!
If some zero eigenvalues, must eliminate the zero-variance dimensions (eigenvectors).
[I'm not going to discuss how to do that today, but it involves projecting the sample points onto a subspace along the eigenvectors with eigenvalue zero.]

For LDA:

$$\hat{\Sigma} = \frac{1}{n} \sum_C \sum_{i:y_i=C} (x_i - \mu_C)(x_i - \mu_C)^\top \qquad \Leftarrow \text{ pooled within-class covariance matrix}$$

[Notice that although we're computing one covariance matrix for all the data, each sample point contributes with respect to *its own class's mean*. This gives a very different result than if you simply compute one covariance matrix for all the points using the global mean! In the former case, the variances are typically smaller.]

[Let's revisit QDA and LDA and see what has changed now that we know anisotropic Gaussians. The short answer is "not much has changed." By the way, capital X is once again a random variable.]

QDA

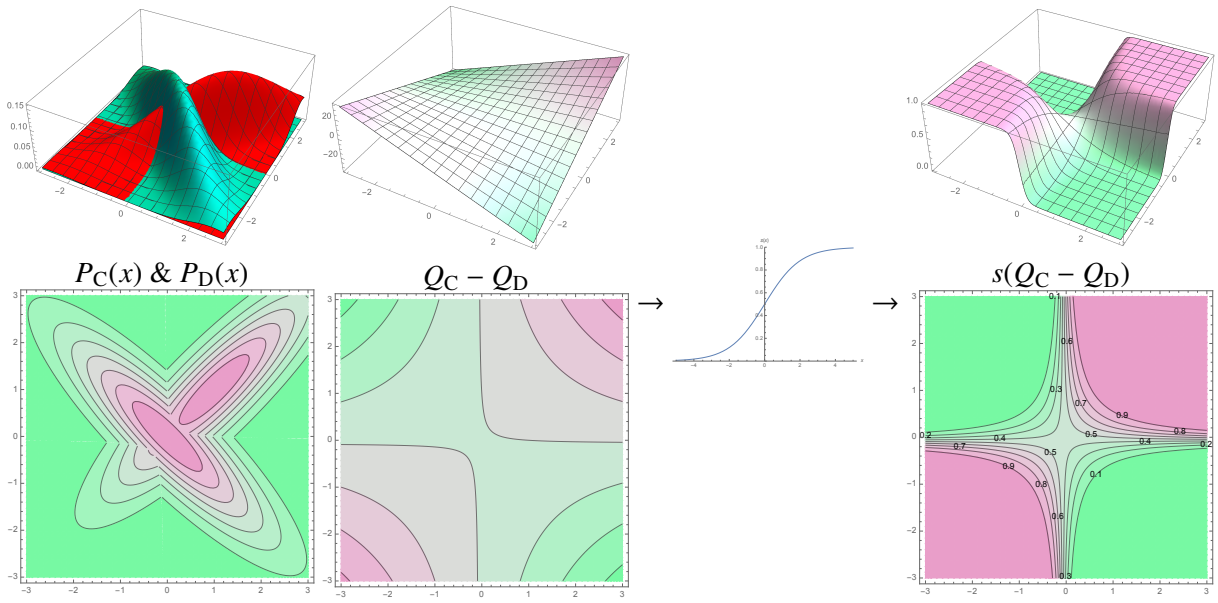
π_C, μ_C, Σ_C may be different for each class C.

Choosing C that maximizes $P(X = x|Y = C)\pi_C$ is equivalent to maximizing the quadratic discriminant fn

$$Q_C(x) = \ln((\sqrt{2\pi})^d P_C(x)\pi_C) = -\frac{1}{2}q_C(x) - \frac{1}{2}\ln|\Sigma_C| + \ln\pi_C \qquad \text{[works for any \# of classes]}$$

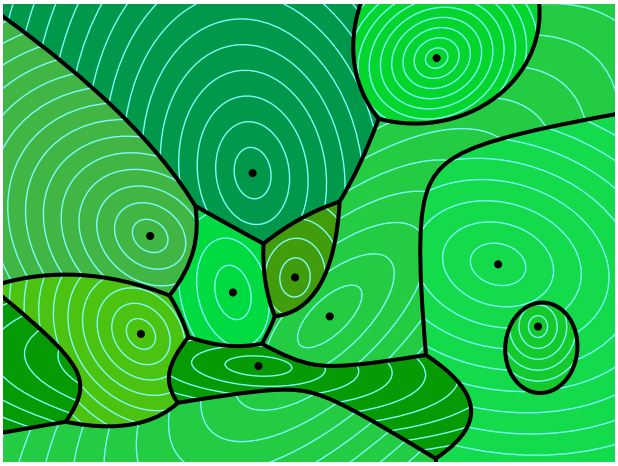
$\uparrow$
Gaussian for C

2 classes: Prediction fn $Q_C(x) - Q_D(x)$ is quadratic, but may be indefinite
$\Rightarrow$ Bayes decision boundary is a quadric.
Posterior is $P(Y = C|X = x) = s(Q_C(x) - Q_D(x))$
where $s(\cdot)$ is logistic fn



qdaaniso3d.pdf, qdaanisocontour.pdf, qdaanisodiff3d.pdf, qdaanisodiffcontour.pdf, logistic.pdf, qdaanisoposterior3d.pdf, qdaanisoposteriorcontour.pdf

[(Show this figure on a separate "whiteboard.") An example where the decision boundary is a hyperbola. At left, two anisotropic Gaussians. Center left, the difference $Q_C - Q_D$. After applying the logistic function to this difference we obtain the posterior probabilities at right. However, observe that we can see the decision boundary in both the contour plot for $Q_C - Q_D$ and the contour plot for $s(Q_C - Q_D)$. We don't need to apply the logistic function to find the decision boundary, but we do need to apply it if we want the posterior probabilities.]



aniso.pdf [When you have many classes, their QDA decision boundaries form an anisotropic Voronoi diagram. Interestingly, a cell of this diagram might not be connected.]

LDA

One Σ for all classes.
[Once again, the quadratic terms cancel each other out so the predictor function is linear and the decision boundary is a hyperplane.]

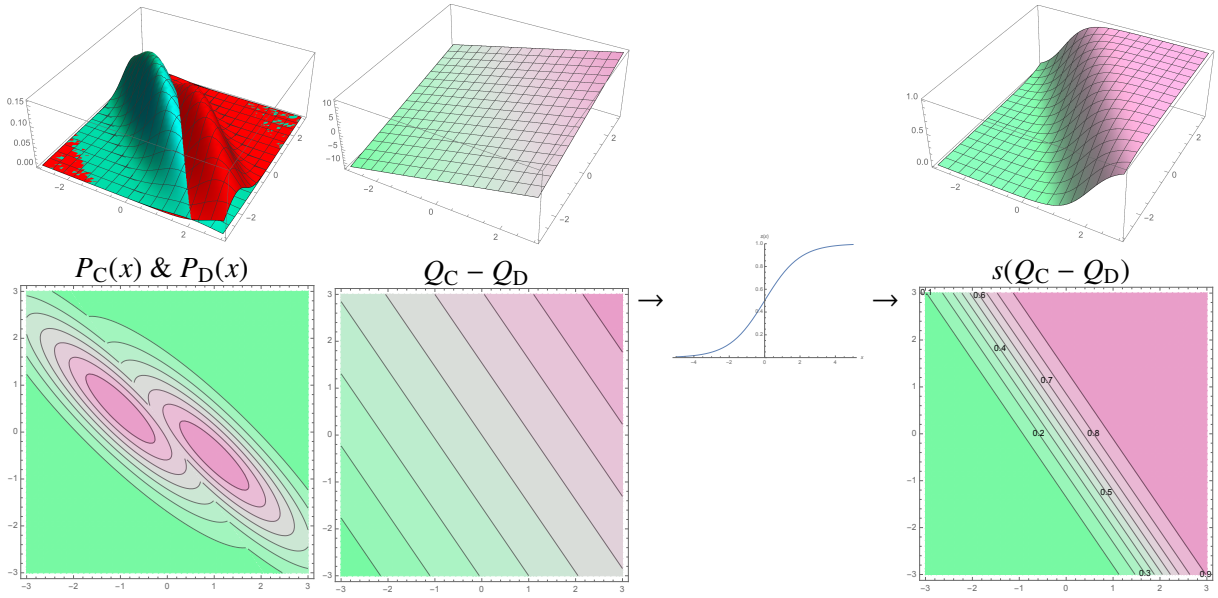
$$Q_C(x) - Q_D(x) =$$

$$\underbrace{(\mu_C - \mu_D)^\top \Sigma^{-1}}_{w^\top} x \underbrace{- \frac{\mu_C^\top \Sigma^{-1} \mu_C - \mu_D^\top \Sigma^{-1} \mu_D}{2} + \ln \pi_C - \ln \pi_D}_{+\alpha}$$

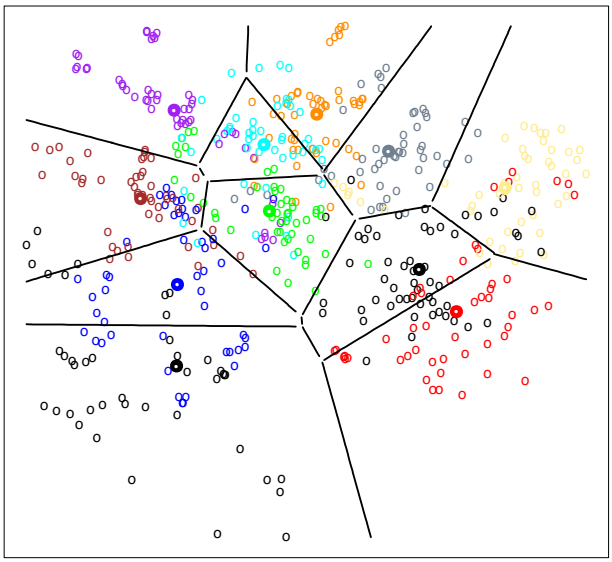
Choose class C that maximizes the linear discriminant fn

$$\mu_C^\top \Sigma^{-1} x - \frac{1}{2} \mu_C^\top \Sigma^{-1} \mu_C + \ln \pi_C \qquad \text{[works for any \# of classes]}$$

2 classes: Decision boundary is $w^\top x + \alpha = 0$
Posterior is $P(Y = C|X = x) = s(w^\top x + \alpha)$



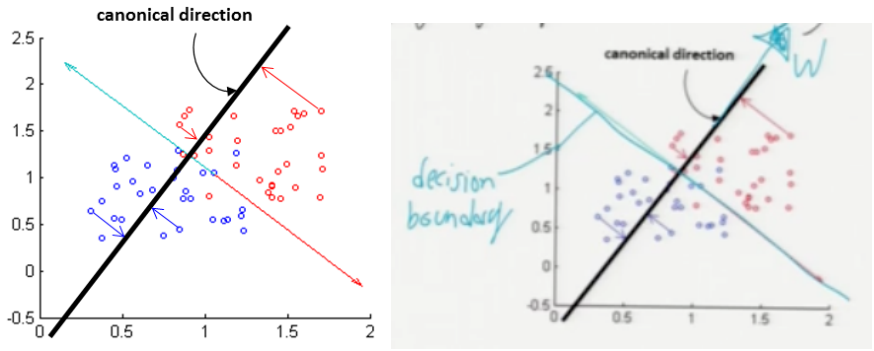
ldaaniso3d.pdf, ldaanisocontour.pdf, ldaanisodiff3d.pdf, ldaanisodiffcontour.pdf, logistic.pdf, ldaanisoposterior3d.pdf, ldaanisoposteriorcontour.pdf
[(Show this figure on a separate "whiteboard.") In LDA, the decision boundary is always a hyperplane.]



LDAdata.pdf (ESL, Figure 4.11) [An example of LDA with messy data. The points are not sampled from perfect Gaussians, but LDA still works reasonably well.]

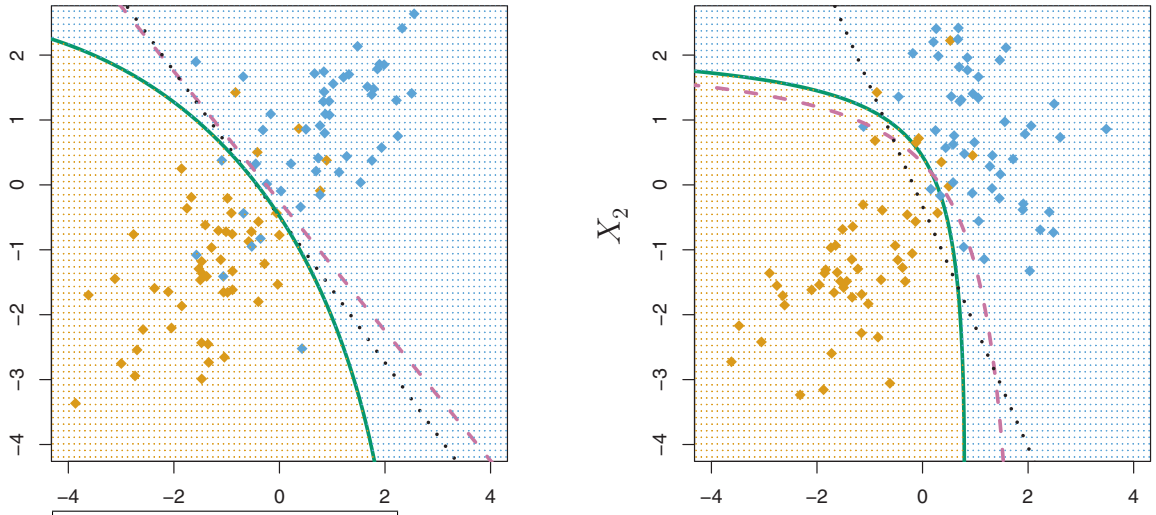
Notes:

- Changing prior π_C (or loss) is easy: adjust α.
- LDA often interpreted as projecting points onto normal vector w; cutting the line in half.



project.png

- For 2 classes,
  - LDA has $d + 1$ parameters (w, α);
  - QDA has $\frac{d(d+3)}{2} + 1$ parameters;
  - QDA more likely to overfit.



ldaqda.pdf (ISL, Figure 4.9) [In these examples, the QDA decision boundary is green, the LDA decision boundary is black (and dotted), and the Bayes optimal decision boundary is purple (and dashed). When the optimal boundary is linear, as at left, LDA gives a more stable fit whereas QDA may overfit. When the optimal boundary is curved, as at right, QDA better fits it.]

- With features, LDA can give nonlinear boundaries; QDA can give nonquadratic.
- We don't get *true* optimum Bayes classifier
  - estimate distributions from finite data
  - real-world data not perfectly Gaussian
- Posterior gives decision boundaries for 10% confidence, 50%, 90%, etc.
  - choosing isovalue = probability p is same as choosing asymmetrical loss p for false positive, $1 - p$ for false negative.

[LDA & QDA are best in practice for many applications. In the STATLOG project, either LDA or QDA were in the top three classifiers for 10 out of 22 datasets. But it's not because all those datasets are Gaussian. LDA & QDA work well when the data can only support simple decision boundaries such as linear or quadratic, because Gaussian models provide stable estimates. See ESL, Section 4.3]

10 Regression, including Least-Squares Linear and Logistic Regression

REGRESSION aka Fitting Curves to Data

Classification: given sample x, predict class (often binary)
Regression: given sample x, predict a numerical value

[Classification gives a discrete prediction, whereas regression gives us a quantitative prediction, usually on a continuous scale.]
[We've already seen an example of regression in Gaussian discriminant analysis. QDA and LDA don't just give us a classifier; they also give us the probability that a particular class label is correct. So QDA and LDA implicitly do regression on probability values.]

- Choose form of regression fn $h(x; p)$ with parameters p (h = hypothesis)
  - like predictor fn in classification [e.g. linear, quadratic, logistic in x]
- Choose a cost fn (objective fn) to optimize
  - usually based on a loss fn; e.g. risk fn = expected loss

Some regression fns:
(1) linear: $h(x; w, \alpha) = w^\top x + \alpha$
(2) polynomial [equivalent to linear regression with added polynomial features]
(3) logistic: $h(x; w, \alpha) = s(w^\top x + \alpha)$ recall: logistic fn $s(\gamma) = \frac{1}{1+e^{-\gamma}}$

[The last choice is interesting. You'll recall that LDA produces a posterior probability function with this equation. So this equation seems to be a natural form for modeling certain probabilities. If we want to model class probabilities, sometimes we use LDA; but alternatively, we could skip fitting Gaussians to samples, and instead just directly try to fit a logistic function to a set of probabilities.]

Some loss fns: let z be prediction $h(x)$; y be true value
(A) $L(z, y) = (z - y)^2$ squared error
(B) $L(z, y) = |z - y|$ absolute error
(C) $L(z, y) = -y \ln z - (1 - y) \ln(1 - z)$ logistic loss, aka cross-entropy: $y \in [0, 1]$, $z \in (0, 1)$

Some cost fns to minimize:
(a) $J(h) = \frac{1}{n} \sum_{i=1}^{n} L(h(X_i), y_i)$ mean loss [you can leave out the "$\frac{1}{n}$"]
(b) $J(h) = \max_{i=1}^{n} L(h(X_i), y_i)$ maximum loss
(c) $J(h) = \sum_{i=1}^{n} \omega_i L(h(X_i), y_i)$ weighted sum [some samples are more important than others]
(d) $J(h) = \frac{1}{n} \sum_{i=1}^{n} L(h(X_i), y_i) + \lambda |w|^2$ ℓ_2 penalized/regularized
(e) $J(h) = \frac{1}{n} \sum_{i=1}^{n} L(h(X_i), y_i) + \lambda |w|_{\ell_1}$ ℓ_1 penalized/regularized

Some famous regression methods:

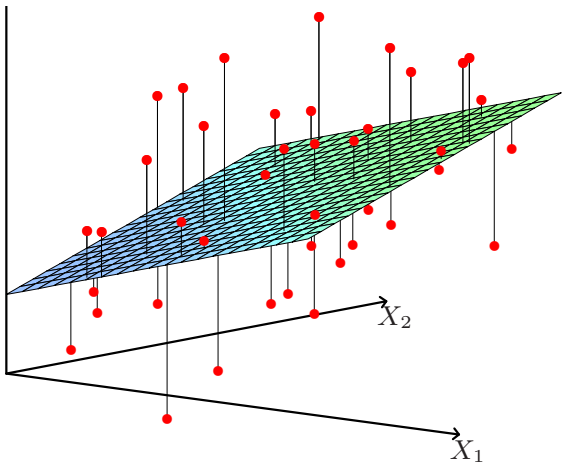
Least-squares linear regr.:	(1) + (A) + (a)	quadratic; minimize w/calculus
Weighted least-squ. linear:	(1) + (A) + (c)	
Ridge regression:	(1) + (A) + (d)	
Lasso:	(1) + (A) + (e)	minimize w/gradient descent
Logistic regr.:	(3) + (C) + (a)	
Least absolute deviations:	(1) + (B) + (a)	minimize w/linear programming
Chebyshev criterion:	(1) + (B) + (b)	

[I have given you several choices of regression function form, several choices of loss function, and several choices of objective function. These are interchangeable parts where you can snap one part out and replace it with a different one. But the optimization algorithm and its speed depend crucially on which parts you pick. Let's see some examples.]

LEAST-SQUARES LINEAR REGRESSION (Gauss, 1801)

linear regression fn (1) + squared loss fn (A) + cost fn (a).

Find w, α that minimizes $\sum_{i=1}^{n} (X_i \cdot w + \alpha - y_i)^2$



linregress.pdf (ISL, Figure 3.4) [An example of linear regression.]

Convention: X is $n \times d$ design matrix of samples
y is d-vector of dependent scalars.

$$\begin{bmatrix} X_{11} & X_{12} & \dots & X_{1j} & \dots & X_{1d} \\ X_{21} & X_{22} & & X_{2j} & & X_{2d} \\ \vdots & & & & & \\ X_{i1} & X_{i2} & & X_{ij} & & X_{id} \\ \vdots & & & & & \\ X_{n1} & X_{n2} & & X_{nj} & & X_{nd} \end{bmatrix} \leftarrow \text{sample } X_i^\top \qquad \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$$

$\uparrow$ feature column X_{*j} $\qquad$ $\uparrow$ y

Usually $n > d$.

Recall fictitious dimension trick [from Lecture 3]: replace $x \cdot w + \alpha$ with

$$[x_1 \quad x_2 \quad 1] \cdot \begin{bmatrix} w_1 \\ w_2 \\ \alpha \end{bmatrix}$$

Now X is an $n \times (d+1)$ matrix; w is a $(d+1)$-vector.

Find w that minimizes $|Xw - y|^2$ = RSS(w), for residual sum of squares

[We know X and y, but w is unknown; we want to solve for w.]

Optimize by calculus:

$$\begin{aligned} \text{minimize RSS}(w) &= w^\top X^\top X w - 2y^\top X w + y^\top y \\ \nabla \text{RSS} &= 2X^\top X w - 2X^\top y = 0 \\ &\Rightarrow \underbrace{X^\top X}_{(d+1)\times(d+1)} \underbrace{w = X^\top y}_{(d+1)-\text{vectors}} \quad \Leftarrow \text{the normal equations } [w \text{ unknown; } X \,\&\, y \text{ known}] \end{aligned}$$

If $X^\top X$ is singular, problem is underconstrained
[because the samples all lie on a common hyperplane. Notice that $X^\top X$ is always positive semidefinite.]

We use a linear solver to find $w = \underbrace{(X^\top X)^{-1} X^\top}_{X^+,\ \text{the pseudoinverse of } X,\ (d+1)\times n} y$ [never actually invert the matrix!]

[X is usually not square, so it can't have an inverse. However, every X has a pseudoinverse X^+, and if X has rank $d+1$, then X^+ is a "left inverse."]

Observe: $X^+X = (X^\top X)^{-1} X^\top X = I \quad \Leftarrow (d+1) \times (d+1)$ [which explains the name "left inverse"]

Observe: the predicted values of y are $\hat{y}_i = h(x_i) \quad \Rightarrow \quad \hat{y} = Xw = XX^+y = Hy$
where $H = XX^+$ is called the hat matrix because it puts the hat on y
$\uparrow\ n \times n$ [and if $n > d+1$, it is not the identity matrix!]

Interpretation as a projection:
- $\hat{y} = Xw \in \mathbb{R}^n$ is a linear combination of columns of X (one per feature)
- For fixed X, varying w, Xw is subspace of $\mathbb{R}^n$ spanned by columns

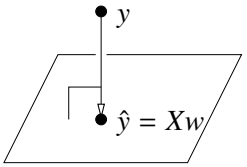


Figure in n-dimensional space (1 dim/sample)
NOT d-dimensional feature space (row space of X).

$\Leftarrow$ subspace spanned by X's columns
(at most $d+1$ dimensions)

- Minimizing $|\hat{y} - y|$ finds point $\hat{y}$ nearest y on subspace
  $\Rightarrow$ projects y onto subspace
  [the vertical line is the direction of projection and the error vector]
- Error is smallest when line is perpendicular to subspace: $X^\top(Xw - y) = 0$
  $\Rightarrow$ the normal equations!
- Hat matrix H does the projecting. [H is also sometimes called the projection matrix.]

Advantages:
- Easy to compute; just solve a linear system.
- Unique, stable solution. [Except when the problem is underconstrained.]

Disadvantages:
- Very sensitive to outliers, because error is squared!
- Fails if $X^\top X$ is singular.

[Least-squares linear regression was apparently first posed and solved in 1801 by the great mathematician Carl Friedrich Gauss, who used least squares to predict the trajectory of the planetoid Ceres. A paper he wrote on the topic is regarded as the birth of modern linear algebra.]

LOGISTIC REGRESSION (David Cox, 1958)

logistic regression fn (3) + logistic loss fn (C) + cost fn (a).
Fits "probabilities" in range (0, 1).

Usually used for classification. The input y_i's *can* be probabilities,
but in most applications they're all 0 or 1.

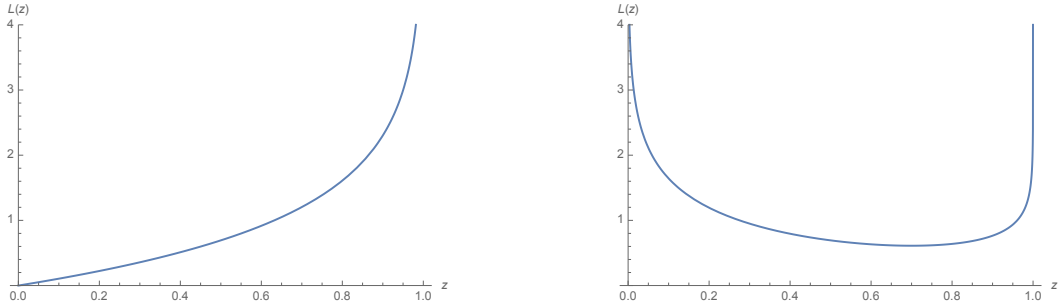
QDA, LDA: generative models
logistic regression: discriminative model

With X and w including the fictitious dimension; α is w's last component ...

Find w that maximizes

$$J = \sum_{i=1}^{n} \left(y_i \ln s(X_i \cdot w) + (1 - y_i) \ln (1 - s(X_i \cdot w)) \right)$$

[Note that we are maximizing, not minimizing, this function because
I've flipped the sign of the logistic loss (C).]

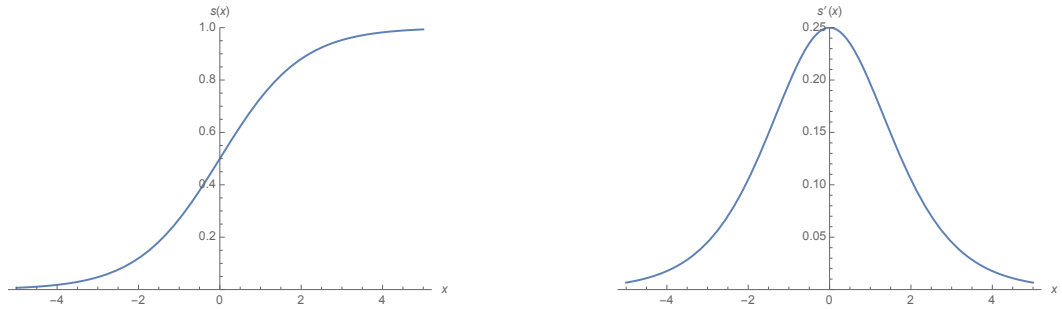


logloss0.pdf, loglosspt7.pdf [Plots of the loss $L(z, y)$ for $y = 0$ (left) and $y = 0.7$ (right). As you might guess, the left function is minimized at $z = 0$, and the right function is minimized at $z = 0.7$.]

$-J(w)$ is convex! [J is "concave."] Solve by gradient ascent.

[To do gradient ascent, we'll need to compute some derivatives.]

$$\begin{aligned} s'(\gamma) &= \frac{\mathrm{d}}{\mathrm{d}\gamma} \frac{1}{1 + e^{-\gamma}} = \frac{e^{-\gamma}}{(1 + e^{-\gamma})^2} \\ &= s(\gamma)\,(1 - s(\gamma)) \end{aligned}$$



logistic.pdf, dlogistic.pdf [Plots of $s(\gamma)$ (left) and $s'(\gamma)$ (right).]

Let $s_i = s(X_i \cdot w)$

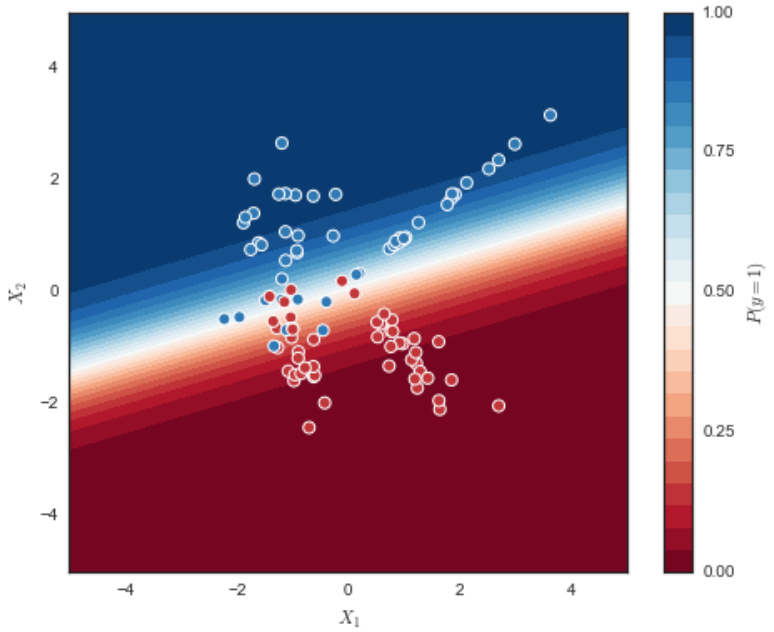
$$\begin{aligned}
\nabla_w J &= \sum \left(\frac{y_i}{s_i} \nabla s_i - \frac{1 - y_i}{1 - s_i} \nabla s_i \right) \\
&= \sum \left(\frac{y_i}{s_i} - \frac{1 - y_i}{1 - s_i} \right) s_i (1 - s_i) X_i \\
&= \sum (y_i - s_i) X_i \\
&= X^\top (y - s) \qquad \text{where } s = s(Xw) = \begin{bmatrix} s_1 \\ s_2 \\ \vdots \\ s_n \end{bmatrix}
\end{aligned}$$

Gradient ascent rule: $w \leftarrow w + \epsilon X^\top (y - s(Xw))$

Stochastic gradient ascent: $w \leftarrow w + \epsilon (y_i - s(X_i \cdot w)) X_i$
Works best if we shuffle samples in random order, process one by one.
For very large n, sometimes converges before we visit all samples!

[This looks a lot like the perceptron learning rule. The only difference is that the "$-s_i$" part is new.]

Starting from $w = 0$ works well in practice.



problogistic.png, by "mwascom" of Stack Overflow

http://stackoverflow.com/questions/28256058/plotting-decision-boundary-of-logistic-regression

[An example of logistic regression.]

11 More Regression; Newton's Method; ROC Curves

WEIGHTED LEAST-SQUARES REGRESSION

linear regression fn (1) + squared loss fn (A) + cost fn (c).

[The idea of weighted least-squares is that some sample points might be more trusted than others, or there might be certain points you want to fit particularly well. So you assign those more trusted points a higher weight. If you suspect some points of being outliers, you can assign them a lower weight.]

Assign each sample point a weight ω_i; collect them in $n \times n$ diagonal matrix Ω.

Greater $\omega_i \to$ work harder to minimize $|\hat{y}_i - y_i|$ recall: $\hat{y} = Xw$

$$\boxed{\text{Find } w \text{ that minimizes } (Xw-y)^\top \Omega (Xw-y)} = \sum_{i=1}^{n} \omega_i \left((Xw)_i - y_i\right)^2$$

Solve for w in normal equations: $X^\top \Omega X w = X^\top \Omega y$

[Once again, you can interpret it as a projection:]

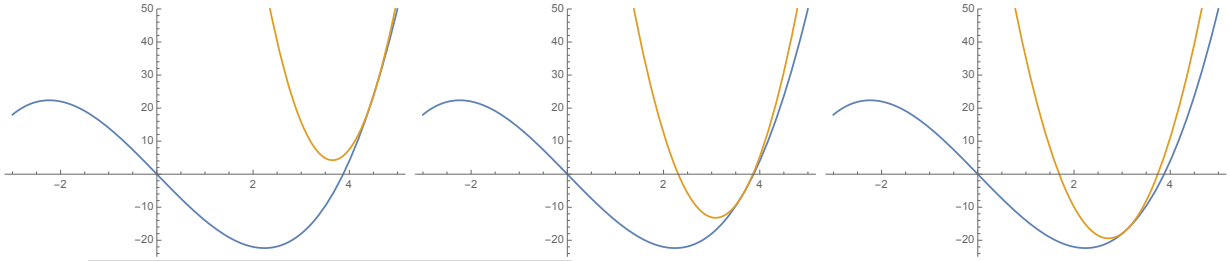
Note: $\Omega^{1/2}\hat{y}$ is orthogonal projection of $\Omega^{1/2}y$ onto subspace spanned by columns of $\Omega^{1/2}X$.

[If you stretch the n-dimensional space by applying the linear transformation $\Omega^{1/2}$, it's an orthogonal projection in that stretched space.]

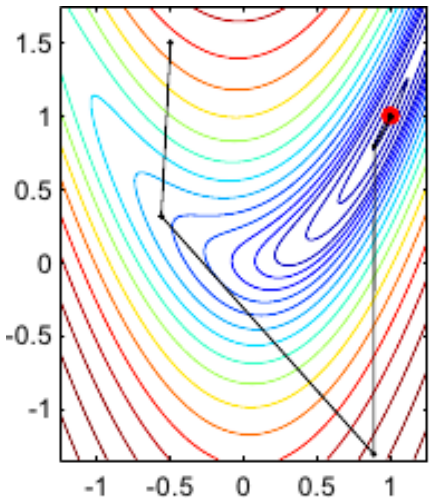
NEWTON'S METHOD

Iterative optimization method for smooth fn $J(w)$.
Often much faster than gradient descent. [We'll use for logistic regression.]

Idea: You're at point v. Approximate $J(w)$ near v by quadratic fn.
Jump to its unique critical pt. Repeat until bored.



newton1.pdf, newton2.pdf, newton3.pdf [Three iterations of Newton's method in one-dimensional space. We seek the minimum of the blue curve, J. Each brown curve is a local quadratic approximation to J. Each iteration, we jump to the bottom of the brown parabola.]



newton2D.png [Steps taken by Newton's method in two-dimensional space.]

Taylor series about v:

$$\nabla J(w) = \nabla J(v) + (\nabla^2 J(v))\,(w - v) + O(|w - v|^2)$$

where $\nabla^2 J(v)$ is the Hessian matrix of $J(w)$ at v.

Find critical pt w by setting $\nabla J(w) = 0$:

$$w = v - (\nabla^2 J(v))^{-1}\,\nabla J(v)$$

[This is an iterative update rule you can repeat until it converges to a solution. As usual, we don't really want to compute a matrix inverse. Instead, we solve a linear system of equations.]

Newton's method:

pick starting point w
repeat until convergence
    $e \leftarrow$ solution to linear system $(\nabla^2 J(w))\,e = -\nabla J(w)$
    $w \leftarrow w + e$

Warning: Doesn't know difference between minima, maxima, saddle points.
Starting point must be "close enough" to desired solution.

[If the objective function J is actually quadratic, Newton's method needs only one step to find the correct answer. The closer J is to quadratic, the faster Newton's method tends to converge.]

[Newton's method is superior to blind gradient descent for some optimization problems for several reasons. First, it tries to find the right step length to reach the minimum, rather than just walking an arbitrary distance downhill. Second, rather than follow the direction of steepest descent, it tries to optimize all directions at once.]

[Nevertheless, it has some major disadvantages. The biggest one is that computing the Hessian can be quite expensive, and it has to be recomputed every iteration. It can work well for low-dimensional feature spaces, but you would never use it for a neural net, because there are too many weights. Newton's method also doesn't work for most nonsmooth functions. It particularly fails for the perceptron risk function, whose Hessian is zero, except where it's not even defined.]

LOGISTIC REGRESSION (continued)

[Let's use Newton's method to solve logistic regression faster.]

Recall: $s'(\gamma) = s(\gamma)\,(1 - s(\gamma))$, $\qquad s_i = s(X_i \cdot w)$,

$$\nabla_w J = \sum_{i=1}^{n} (y_i - s_i)\, X_i = X^\top (y - s) \qquad \text{where } s = \begin{bmatrix} s_1 \\ s_2 \\ \vdots \\ s_n \end{bmatrix}$$

[Now we work out the Hessian too, so we can use Newton's method.]

$$\nabla_w^2 J(w) = -\sum_{i=1}^{n} s_i(1 - s_i)\, X_i X_i^\top = -X^\top \Omega X \quad \text{where } \Omega = \begin{bmatrix} s_1(1-s_1) & 0 & \dots & 0 \\ 0 & s_2(1-s_2) & & 0 \\ \vdots & & \ddots & \vdots \\ 0 & 0 & \dots & s_n(1-s_n) \end{bmatrix}$$

Ω is +ve definite $\forall w$, so $X^\top \Omega X$ is +ve semidefinite $\forall w$, so $-J(w)$ is convex.
[Therefore, the logistic regression cost function is convex and has only one local optimum, so Newton's method finds an optimal point if it converges at all.]

Newton's method:

Solve for e in normal equations: $(X^\top \Omega X)\, e = X^\top (y - s)$ $\qquad$ Recall: Ω, s are fns of w
$w \leftarrow w + e$
repeat until "convergence"

[Notice that this looks a lot like weighted least squares, but the weight matrix Ω and the right-hand-side vector $y - s$ change every iteration.]
An example of iteratively reweighted least squares.
Ω prioritizes points with s_i near 0.5; tunes out points near 0/1.
[In other words, sample points near the decision boundary have the biggest effect on the iterations. Meanwhile, the iterations move the decision boundary; in turn, that movement may change which points have the most influence. In the end, only the points near the decision boundary make a big contribution to the logistic fit.]

Idea: If n very large, save time by using a random subsample of the points per iteration. Increase sample size as you go.

[The principle is that the first iteration isn't going to take you all the way to the optimal point, so why waste time looking at all the sample points? Whereas the last iteration should be the most accurate one.]

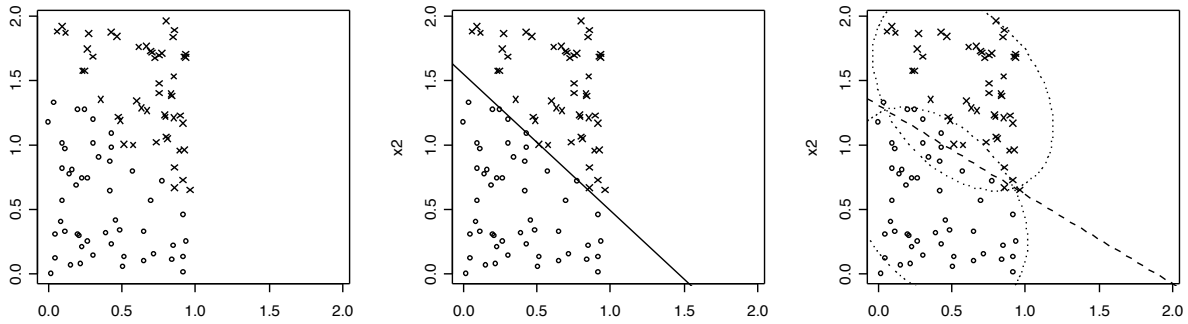
LDA vs. logistic regression

Advantages of LDA:

- For well-separated classes, LDA stable; log. reg. surprisingly unstable
- > 2 classes easy & elegant; log. reg. needs modifying ("softmax regression")
- Slightly more accurate when classes nearly normal, especially if n is small

Advantages of log. reg.:

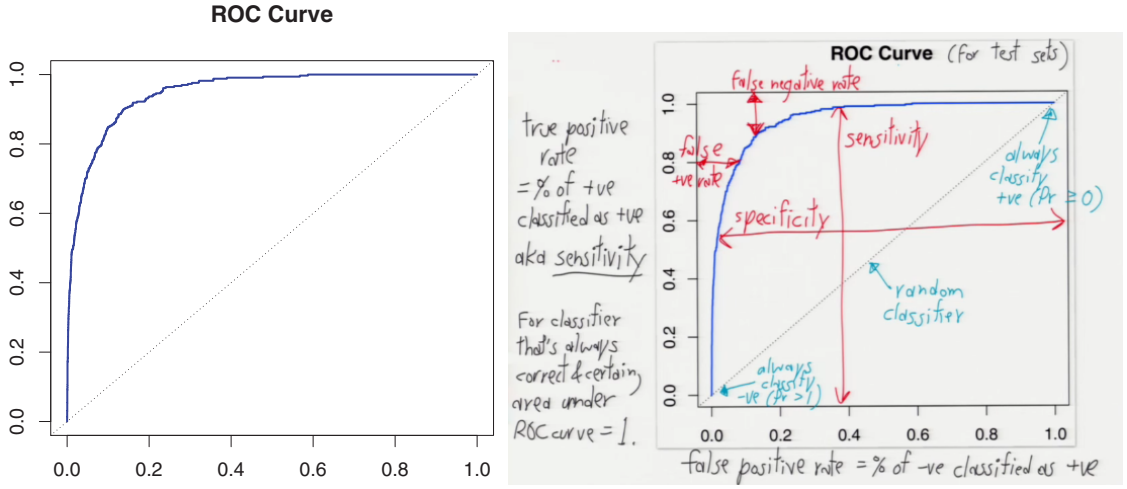
- More emphasis on decision boundary [though not as much as SVMs]



logregvsLDAuni.pdf [Logistic regression vs. LDA for a linearly separable data set with a very narrow margin. Logistic regression (center) succeeds in separating these classes, because points near the decision boundary have more influence. (Some luck is involved too; unlike SVMs, logistic regression doesn't always separate separable points.) In this example, LDA (right) misclassifies some of the training points.]

- Hence less sensitive to outliers
- Easy & elegant treatment of "partial" class membership; LDA is all-or-nothing
- More robust on some non-Gaussian distributions (e.g. large skew)

ROC CURVES (for test sets)



ROC.pdf

[This is a ROC curve. That stands for receiver operating characteristics, which is an awful name but we're stuck with it for historical reasons.
It is made by running a classifier on the test set or validation set.
It shows the rate of false positives vs. true positives for a range of settings.
We assume there is a knob we can turn to trade off these two types of error; in this case, that knob is the probability threshold for Gaussian discriminant analysis or linear regression.
However, neither axis of this plot is the probability threshold.]
x-axis: "false positive rate = % of −ve classified as +ve"
y-axis: "true positive rate = % of +ve classified as +ve aka sensitivity"
"false negative rate": vertical distance from curve to top
"specificity": horizontal distance from curve to right
[You generate this curve by trying *every* probability threshold; for each threshold, measure the false positive & true positive rates and plot a point.]
upper right corner: "always classify +ve ($\Pr \geq 0$)"
lower left corner: "always classify −ve ($\Pr > 1$)"
diagonal: "random classifiers"
[A rough measure of a classifier's effectiveness is the area under the curve. For a classifier that is always correct, the area under the curve is one.]

[IMPORTANT: In practice, the trade-off between false negatives and false positives is usually negotiated by choosing a point on this plot, based on real test data, and NOT by taking the choice of threshold that's best in theory.]

LEAST-SQUARES POLYNOMIAL REGRESSION

Replace each X_i with feature vector $\Phi(X_i)$ with all terms of degree $1 \ldots p$

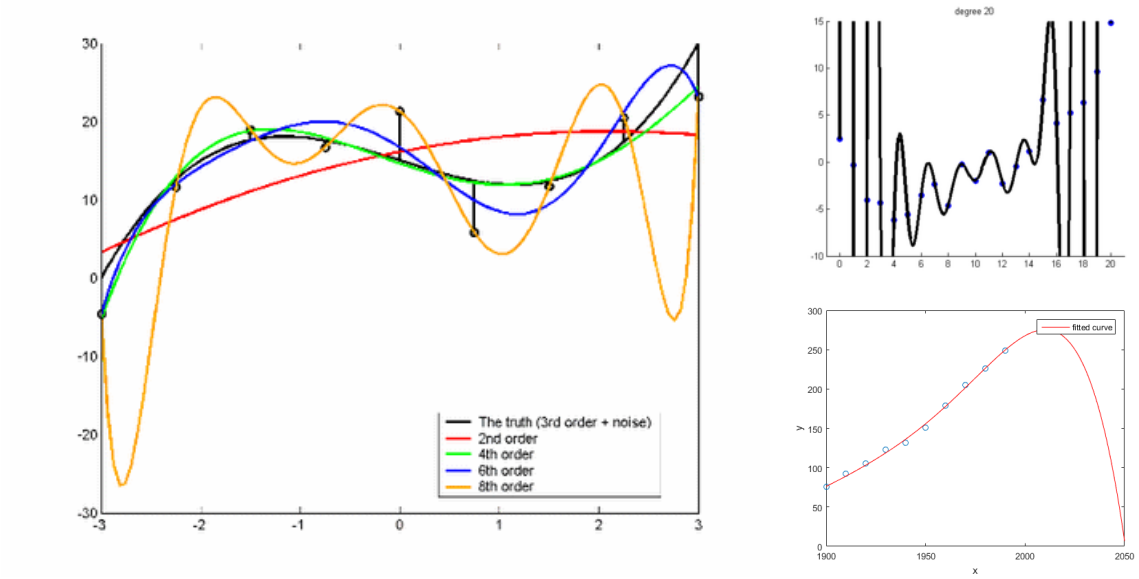
e.g. $\Phi(X_i) = [X_{i1}^2 \quad X_{i1}X_{i2} \quad X_{i2}^2 \quad X_{i1} \quad X_{i2} \quad 1]^\top$

[Notice that we've added the fictitious dimension "1" here, so we don't need to add it again to do linear or logistic regression. This basis covers all polynomials quadratic in X_{i1}, X_{i2}.]

Can also use non-polynomial features (e.g. edge detectors).
Otherwise just like linear or logistic regression.

Log. reg. + quadratic features = same logistic posteriors as QDA.

Very easy to overfit!



overunder.png, degree20.png, UScensusquartic.png

[I close with some examples of polynomial overfitting, to show the importance of choosing the polynomial degree very carefully. At left, we have sampled points from a degree-3 curve (black) with added noise. We show best-fit polynomials of degrees 2, 4, 6, and 8 found by regression of the black points. The degree-4 curve (green) fits the true curve (black) well, whereas the degree-6 and 8 curves overfit the noise and oscillate, and the degree-2 curve underfits. The oscillations in the yellow degree-8 curve are a characteristic problem of polynomial interpolation.]

[At upper right, a degree-20 curve shows just how insane high-degree polynomial oscillations can get.]

[At lower right, somebody has regressed a degree-4 curve to U.S. census population numbers. The curve doesn't oscillate, but can you nevertheless see a flaw? This shows the difficulty of *extrapolation* outside the range of the data.]

12 Statistical Justifications; the Bias-Variance Decomposition

STATISTICAL JUSTIFICATIONS FOR REGRESSION

[So far, I've talked about regression as a way to fit curves to points. Recall how early in the semester I divided machine learning into 4 levels: the application, the model, the optimization problem, and the optimization algorithm. My last two lectures about regression were at the bottom two levels: optimization. The cost functions that we optimize are somewhat arbitrary. Today, let's take a step back to the second level, the model. I will describe some models, how they lead to those optimization problems, and how they contribute to underfitting or overfitting.]

Typical model of reality:

- sample points come from unknown prob. distribution: $X_i \sim D$
- y-values are sum of unknown, non-random surface + random noise: for all X_i,

$$y_i = f(X_i) + \epsilon_i, \qquad \epsilon_i \sim D', \quad D' \text{ has mean zero}$$

[We are positing that reality is described by a function f. We don't know f, but f is not a random variable; it represents a real relationship between x and y that we can estimate. We add to that a random variable ϵ, which represents measurement errors and all the other sources of statistical error when we measure real-world phenomena. Notice that the noise is independent of x. That's a pretty big assumption, and often it does not apply in practice, but that's all we'll have time to deal with this semester. Also notice that this model leaves out systematic errors, like when your measuring device adds one to every measurement, because we usually can't diagnose systematic errors from data alone.]

Goal of regression: find h that estimates f.

Ideal approach: choose $h(x) = \underbrace{\mathrm{E}_Y[Y|X = x]}_{} = f(x) + \mathrm{E}[\epsilon] = f(x)$

[If this expectation exists at all, it partly justifies our model of reality. We can retroactively define f to be this expectation.]
[Draw figure showing example f, distribution for a fixed x.]

Least-Squares Regression from Maximum Likelihood

Suppose $\epsilon_i \sim \mathcal{N}(0, \sigma^2)$; then $y_i \sim \mathcal{N}(f(X_i), \sigma^2)$

Recall that log likelihood for normal dist. is

$$\ln P(y_i) = -\frac{|y_i - \mu|^2}{2\sigma^2} - \text{constant}, \qquad \Leftarrow \mu = f(X_i)$$

$$\ln\left(P(y_1)P(y_2)\dots P(y_n)\right) = \ln P(y_1) + \ln P(y_2) + \dots + \ln P(y_n) = -\frac{1}{2\sigma^2}\sum |y_i - f(X_i)|^2 - \text{constant}$$

Takeaway: Max likelihood $\Rightarrow$ find f by least-squares

[So if the noise is normally distributed, maximum likelihood justifies using the least-squares cost function.]

[However, I've told you in previous lectures that least-squares is very sensitive to outliers. If the error is truly normally distributed, that's not a big deal, especially when you have a lot of sample points. But in the real world, the real distribution of outliers often isn't normal. Outliers might come from wrongly measured measurements, data entry errors, anomalous events, or just not having a normal distribution. When you have a heavy-tailed distribution, for example, least-squares isn't a good choice.]

Empirical Risk

The risk for hypothesis h is expected loss $R(h) = \mathrm{E}[L]$ over all x, y.
Discriminative model: we don't know X's dist. D. How can we minimize risk?
[If we have a generative model, we can estimate the probability distribution for (X, Y) and derive the expected loss. That's what we did for Gaussian discriminant analysis. But today I'm assuming we don't have a generative model, so we don't know those probabilities. Instead, we're going to approximate the distribution in a very crude way: we pretend that the sample points *are* the distribution.]

Empirical distribution: a discrete probability distribution that IS the sample, with each point equally likely
Empirical risk: expected loss under empirical distribution

$$\hat{R}(h) = \frac{1}{n} \sum_{i=1}^{n} L(h(X_i), y_i)$$

[The hat on the R indicates it's only a cheap approximation of the true, unknown statistical risk we really want to optimize. Often, this is the best we can do. For many but not all distributions, it converges to the true risk in the limit as $n \to \infty$.]

Takeaway: this is why we [usually] minimize the sum of loss fns.

Logistic regression from maximum likelihood

If we accept the logistic regression fn, what cost fn should we use?

Given arbitrary sample point x, write probability it is in (not in) the class:
(Fictitious dimension: x ends w/1; w ends w/α)

$$P(y = 1|x; w) = h(x; w) \qquad \Leftarrow \quad h(x; w) = s(w^\top x)$$
$$P(y = 0|x; w) = 1 - h(x; w) \qquad [s \text{ is logistic fn}]$$

Combine these 2 facts into 1:

$$P(y|x; w) = h(x)^y (1 - h(x))^{1-y} \qquad \text{[A bit of a hack, but it works nicely for intermediate values of y]}$$

Likelihood is

$$\mathcal{L}(w; x_1, \ldots, x_n) = \prod_{i=1}^{n} P(y_i|X_i; w)$$

Log likelihood is

$$\begin{aligned} \ell(w) &= \ln \mathcal{L}(w) = \sum_{i=1}^{n} \ln P(y_i|X_i; w) \\ &= \sum_{i=1}^{n} \Big(y_i \ln h(X_i) + (1 - y_i) \ln(1 - h(X_i)) \Big) \end{aligned}$$

... which is negated logistic cost fn $J(w)$.
We want to maximize log likelihood $\Rightarrow$ minimize J.
[So that explains where the weird logistic loss function comes from.]

THE BIAS-VARIANCE DECOMPOSITION

There are 2 sources of error in a hypothesis h:

bias: error due to inability of hypothesis h to fit f perfectly
e.g. fitting quadratic f with a linear h

variance: error due to fitting random noise in data
e.g. we fit linear f with a linear h, yet $h \neq f$.

Model: generate points $X_1 \dots X_n$ from some distribution D
values $y_i = f(X_i) + \epsilon_i$ [remember that ϵ is random with mean zero]
fit hypothesis h to X, y

Now h is a random variable; i.e. its weights are random

Consider an arbitrary pt $z \in \mathbb{R}^d$ (not necessarily a sample point!) and $\gamma = f(z) + \epsilon$.
[So z is *arbitrary*, whereas γ is *random*.]
Note: $\mathrm{E}[\gamma] = f(z)$; $\mathrm{Var}(\gamma) = \mathrm{Var}(\epsilon)$ [So the mean comes from f, and the variance comes from ϵ.]

Risk fn when loss is squared error:

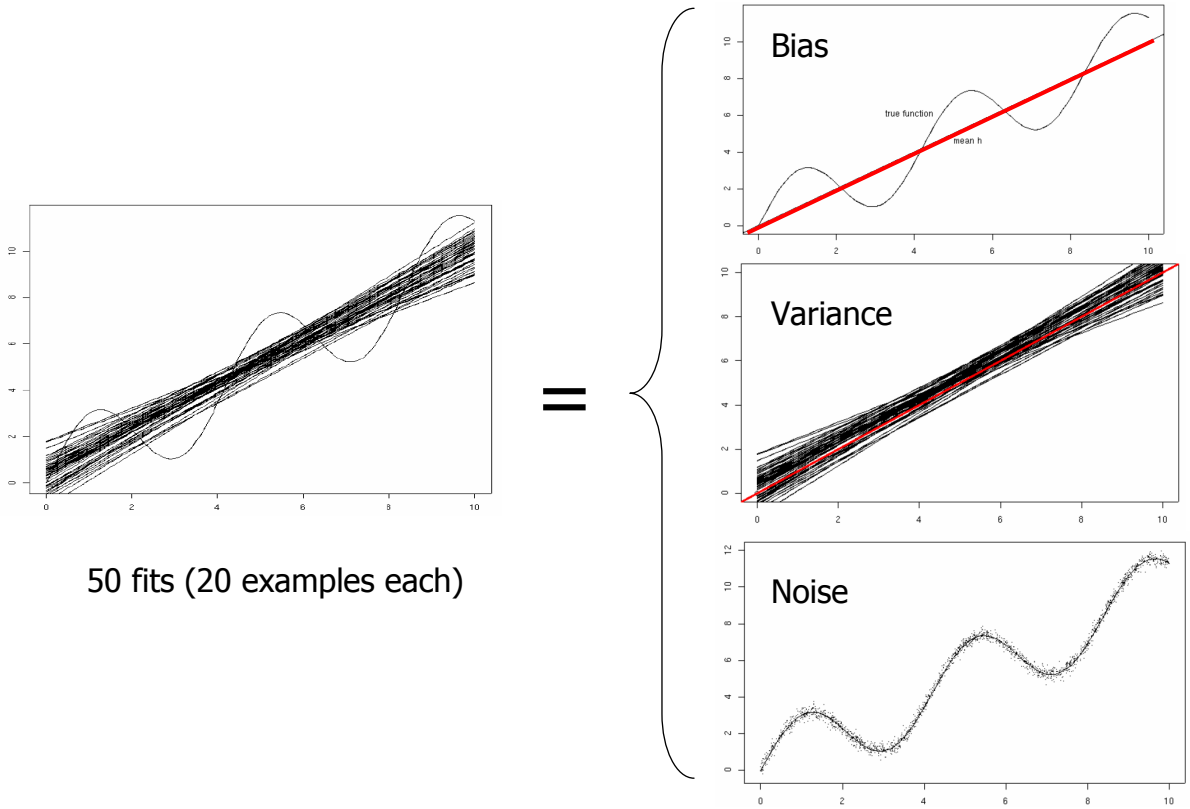
$$R(h) = \mathrm{E}[L(h(z), \gamma)]$$

↑ we take expectation over possible training sets X, y and values of γ

[Stop and take a close look at this expectation. Remember that the hypothesis h is a random variable. We are taking a mean over the probability distribution of hypotheses. That seems pretty weird if you've never seen it before. But remember, the training data X and y come from probability distributions. We use the training data to choose weights, so the weights that define h also come from some probability distribution. It might be hard to work out what that distribution is, but it exists. This "E[·]" is integrating the loss over all possible values of the weights.]

$$\begin{aligned}
&= \mathrm{E}[(h(z) - \gamma)^2] \\
&= \mathrm{E}[h(z)^2] + \mathrm{E}[\gamma^2] - 2\,\mathrm{E}[\gamma\, h(z)] && \text{[Observe that } \gamma \text{ and } h(z) \text{ are independent]} \\
&= \mathrm{Var}(h(z)) + \mathrm{E}[h(z)]^2 + \mathrm{Var}(\gamma) + \mathrm{E}[\gamma]^2 - 2\mathrm{E}[\gamma]\,\mathrm{E}[h(z)] \\
&= (\mathrm{E}[h(z)] - \mathrm{E}[\gamma])^2 + \mathrm{Var}(h(z)) + \mathrm{Var}(\gamma) \\
&= \underbrace{\mathrm{E}[h(z) - f(z)]^2}_{\text{bias}^2 \text{ of method}} + \underbrace{\mathrm{Var}(h(z))}_{\text{variance of method}} + \underbrace{\mathrm{Var}(\epsilon)}_{\text{irreducible error}}
\end{aligned}$$

[This is called the *bias-variance decomposition* of the risk function. Let's look at an intuitive interpretation of these three parts.]

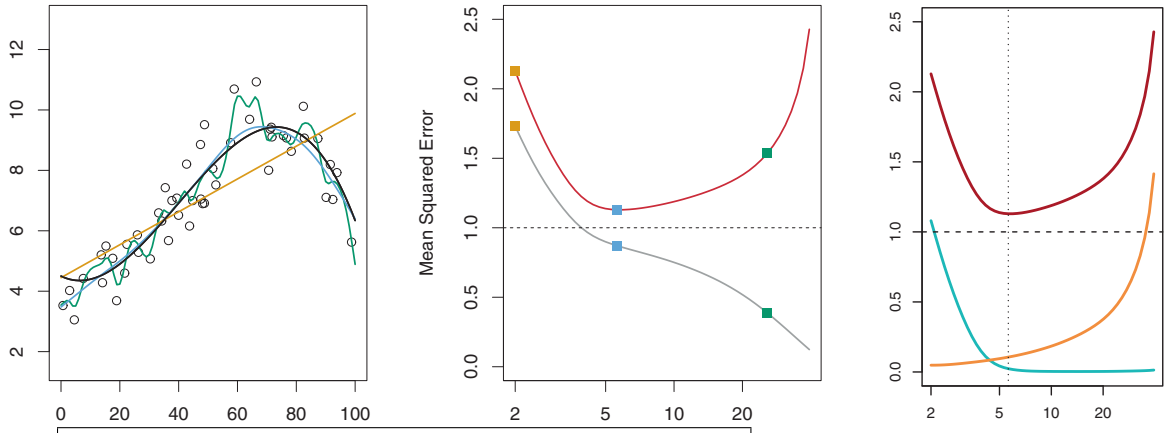


bvn.pdf [In this example, we're trying to fit a sine wave with lines, which obviously aren't going to be accurate. At left, we have generated 50 different hypotheses (lines). At upper right, we see that most test points have a large bias (squared difference between the black and red curves), because lines don't fit sine waves well. However, a few lucky test points have a small bias. At center right, the variance is the expected squared difference between a random black line and the red line. At lower right, the irreducible error is the expected squared difference between a random test point and the sine wave.]

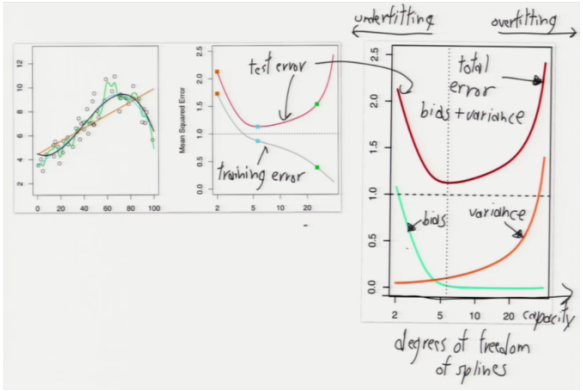
This is pointwise version. Mean version: let $z \sim D$ be random variable;
take expectation over z of bias^2, variance.
[So you can decompose one test point's error into these three components, or you can decompose the error of the hypothesis over its entire range into three components, which tells you how big they'll be on a large test set.]

[Now I will write down a list of consequences of what we've just learned.]

- Underfitting = too much bias
- Overfitting caused by too much variance
- Training error reflects bias but not variance; test error reflects both
  [which is why low training error can fool you when you've overfitted]
- For many distributions, variance $\to 0$ as $n \to \infty$
- If h can fit f exactly, for many distributions bias $\to 0$ as $n \to \infty$
- If h cannot fit f well, bias is large at "most" points
- Adding a good feature reduces bias; adding a bad feature rarely increases it
- Adding a feature usually increases variance
- Can't reduce irreducible error [hence its name]
- Noise in test set affects only Var(ϵ);
  noise in training set affects only bias & Var(h)
- For real-world data, f is rarely knowable (& noise model might be wrong)
  [so we can't actually put numbers to the bias-variance decomposition on real-world data]
- But we can test learning algs by choosing f & making synthetic data



splinefit.pdf, biasvarspline.pdf (ISL, Figures 2.9 and 2.12) [At left, a data set is fit with splines having various degrees of freedom. The synthetic data is taken from the black curve with added noise. At center, we plot training error (gray) and test error (red) as a function of the number of degrees of freedom. At right, we plot the squared test error as a sum of squared bias (blue) and variance (orange). As the number of degrees of freedom increases, the training and test errors both decrease up to degree 6 because the bias decreases, but for higher degrees the test error increases because the variance increases.]



Example: Least-Squares Linear Reg.

For simplicity, assume no fictitious dimension.
[This implies that our linear regression fn has to be zero at the origin.]

Model: $f(z) = v^\top z$ (reality is linear)
[So we could fit f perfectly with a linear h if not for the noise in the training set.]
Let e be noise n-vector, $e \sim \mathcal{N}(0, \sigma^2)$
Training values: $y = Xv + e$
[X & y are the inputs to linear regression. We don't know v or e.]

Lin. reg. computes weights

$$w = X^+ y = X^+(Xv + e) = v + \underbrace{X^+ e}_{\text{noise in weights}}$$ [We want $w = v$, but the noise in y becomes noise in w.]

BIAS is $\mathrm{E}[h(z) - f(z)] = \mathrm{E}[w^\top z - v^\top z] = \mathrm{E}[z^\top X^+ e] = z^\top X^+ \mathrm{E}[e] = 0$

Warning: This does not mean $h(z) - f(z)$ is everywhere 0!
Sometimes +ve, sometimes −ve, mean over training sets is 0.
[Those deviations from the mean are captured in the variance.]

[Not all learning methods give you a bias of zero when a perfect fit is possible; here it's a benefit of the squared error loss function. With a different loss function, we might have a nonzero bias even fitting a linear h to a linear f.]

VARIANCE is $\mathrm{Var}(h(z)) = \mathrm{Var}(z^\top v + z^\top X^+ e) = \mathrm{Var}(z^\top X^+ e)$

[This is the dot product of a vector $z^T X^+$ with an isotropic, normally distributed vector e. The dot product reduces it to a one-dimensional Gaussian along the direction $z^T X^+$, so this variance is just the variance of the 1D Gaussian times the squared length of the vector $z^T X^+$.]

$$\begin{aligned} &= \sigma^2 \left| z^\top X^+ \right|^2 = \sigma^2 z^\top (X^\top X)^{-1} X^\top X (X^\top X)^{-1} z \\ &= \sigma^2 z^\top (X^\top X)^{-1} z \end{aligned}$$

If we choose coordinate system so $\mathrm{E}[X] = 0$,
then $X^\top X \to n\,\mathrm{Cov}(D)$ as $n \to \infty$, so one can show that for $z \sim D$,

$$\mathrm{Var}(h(z)) \sim \sigma^2 \frac{d}{n}$$

[where d is the dimension—the number of features per sample point.]

Takeaways: Bias can be zero when hypothesis function can fit the real one!
[This is a nice property of squared error loss fn.]
Variance portion of RSS (overfitting) decreases as $1/n$ (sample points),
increases as d (features).

[I've used linear regression because it's a relatively simple example. But this bias-variance trade-off applies to nearly all learning algorithms, including classification as well as regression. Of course, for many learning algorithms the math gets a lot more complicated than this, if you can do it at all.]

13 Ridge Regression and the Kernel Trick

RIDGE REGRESSION (aka Tikhonov regularization)

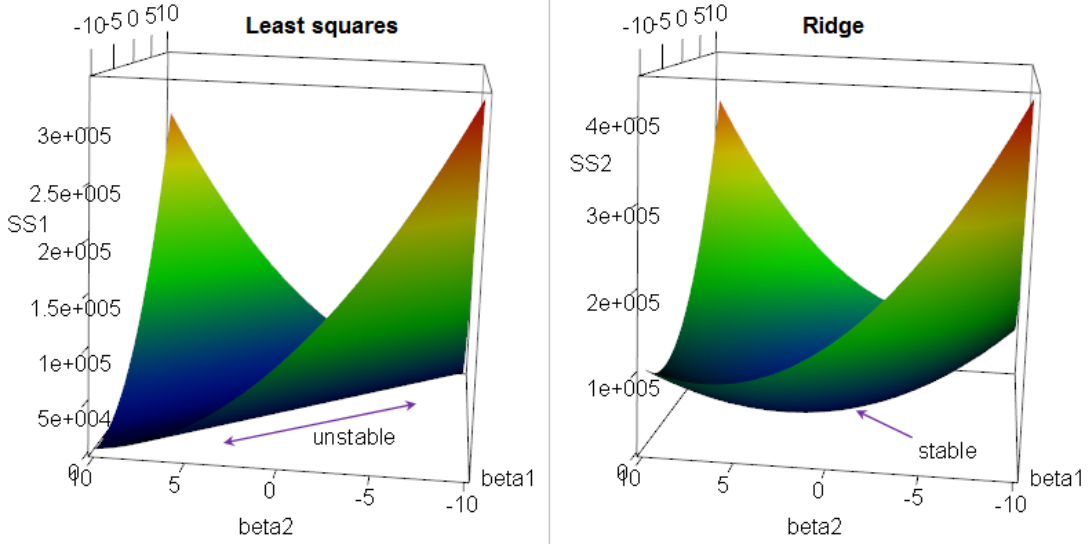
(1) + (A) + ℓ_2 penalized mean loss (d).

Find w that minimizes $|Xw - y|^2 + \lambda\,|w'|^2$ $= J(w)$

where w' is w with component α replaced by 0. X has fictitious dimension but we DON'T penalize α.

Adds a penalty term to encourage small $|w'|$—called shrinkage. Why?

– Guarantees positive definite normal eq's; always unique solution.
[When sample points lie on a common hyperplane in feature space.] E.g. when $d > n$.

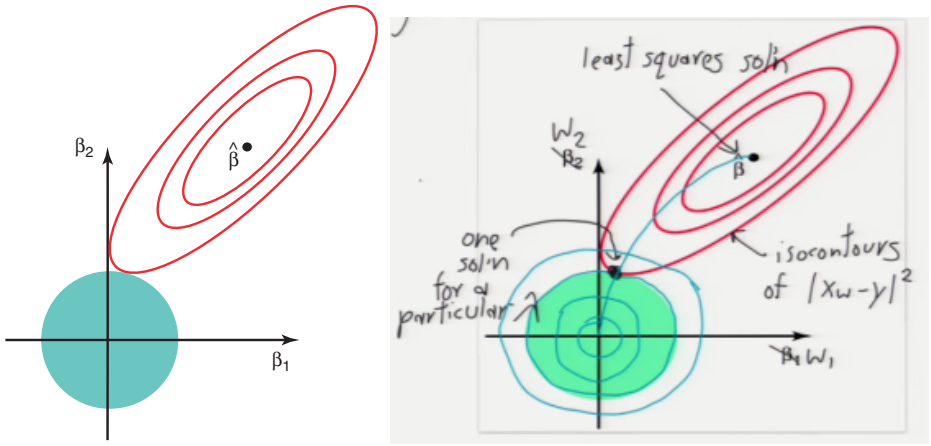


ridgequad.png

[At left, we see a quadratic form for positive semidefinite normal equations, which has many minima. By adding a small penalty term, we obtain positive definite normal equations (right), which have one unique minimum.]

[That was the original motivation, but the next has become more important . . .]

– Reduces overfitting by reducing variance. Why?
Imagine: $275238x_1^2 - 543845x_1x_2 + 385832x_2^2$ is best fit for
well-spaced points all with $|y| < 1$.
Small change in x $\Rightarrow$ big change in y!
[Given that all the y values in the data are small, it's a sure sign of overfitting if tiny changes in x cause huge changes in y.]
So we penalize large weights.



ridgeterms.pdf (ISL, Figure 6.7) [In this plot, β is the least-squares solution. The red ellipses are the isocontours of $|Xw - y|^2$. The isocontours of $|w'|$ are circles centered at the origin (blue). The solution lies where a red isocontour just touches a blue isocontour tangentially. As λ increases, the solution will occur at a more outer red isocontour and a more inner blue isocontour. This helps to reduce overfitting.]

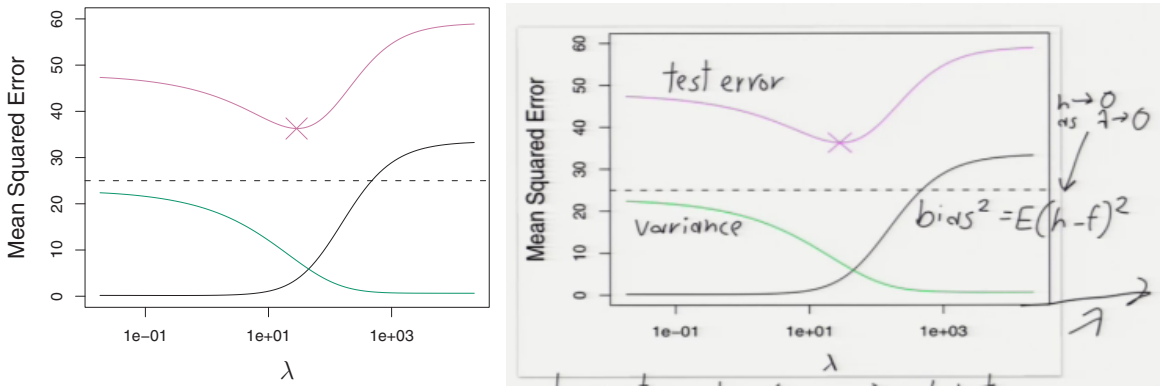
Setting $\nabla J = 0$ gives normal eq'ns

$$(X^\top X + \lambda I')\, w = X^\top y$$

where I' is identity matrix w/bottom right set to zero. [Don't penalize the bias term α.]

Algorithm: Solve for w. Return $h(z) = w^\top z$.

Increasing λ $\Rightarrow$ more regularization; smaller $|w'|$
Given our data model $y = Xv + e$, where e is noise,
variance of ridge reg. is $\mathrm{Var}(z^\top (X^\top X + \lambda I')^{-1} X^\top e)$.
As $\lambda \to \infty$, variance $\to 0$, but bias increases.



ridgebiasvar.pdf (ISL, Figure 6.5) [Plot of bias^2 & variance as λ increases.]

[So, as usual for the bias-variance trade-off, the test error as a function of λ is a U-shaped curve, and we find the bottom by validation.]

λ is a hyperparameter; tune by (cross-)validation.

Ideally, features should be "normalized" to have same variance.
Alternative: use asymmetric penalty by replacing I' w/other diagonal matrix.

Bayesian justification for ridge reg.

Assign a prior probability on w': a Gaussian centered at 0.

Posterior prob $\propto$ likelihood of $w \times$ prior $P(w')$ $\quad\Leftarrow$ Gaussian PDF
Maximize log posterior $\propto \ln$ likelihood $+ \ln P(w')$
$= -\text{const}\,|Xw - y|^2 - \text{const}\,|w'|^2 - \text{const}$

This method (using likelihood, but maximizing posterior) is called maximum *a posteriori* (MAP).

KERNELS

Recall: with d input features, degree-p polynomials blow up to $O(d^p)$ features.
[When d is large, this gets computationally intractable really fast.
As I said in Lecture 4, if you have 100 features per feature vector and you want to use degree-4 predictor functions, then each lifted feature vector has a length on the order of 100 million.]
Today we use magic to use those features without computing them!

Observation: In many learning algs,
- the weights can be written as a linear combo of sample points, &
- we can use inner products of $\Phi(x)$'s only $\quad\Rightarrow\quad$ don't need to compute $\Phi(x)$!

$$\text{Suppose } w = X^\top a = \sum_{i=1}^{n} a_i X_i \quad \text{for some } a \in \mathbb{R}^n.$$

Substitute this identity into alg. and optimize n dual weights a (aka dual parameters) instead of $d + 1$ primal weights w.

Kernel Ridge Regression

Center X and y so their means are zero; e.g. $X_i \leftarrow X_i - \mu_X$
This lets us replace I' with I in normal equations:

$$(X^\top X + \lambda I)w = X^\top y$$

[When we center X and y, the expected value of the intercept $w_{d+1} = \alpha$ is zero. The actual value won't usually be exactly zero, but it will be close enough that we won't do much harm by penalizing the intercept.]

$$\Rightarrow \quad w = \frac{1}{\lambda}(X^\top y - X^\top X w) = X^\top a \quad \text{where } a = \frac{1}{\lambda}(y - Xw)$$
$$\uparrow\ w = X^\top a$$

This shows that w is a linear combo of sample points! To compute a:

$$\lambda a = (y - XX^\top a) \quad\Rightarrow\quad a = (XX^\top + \lambda I)^{-1} y$$

a is the dual solution; solves the dual form of ridge regression:

Find a that minimizes $\lvert XX^\top a - y\rvert^2 + \lambda \lvert X^\top a\rvert^2$

Regression fn is

$$h(z) = w^\top z = a^\top X z = \sum_{i=1}^{n} a_i (X_i^\top z) \qquad \Leftarrow \text{ weighted sum of inner products}$$

Let $k(x, z) = x^\top z$ be kernel fn.
[Later, we'll replace x and z with $\Phi(x)$ and $\Phi(z)$, and that's where the magic will happen.]

Let $K = XX^\top$ be $n \times n$ kernel matrix. Note $K_{ij} = k(X_i, X_j)$.

K is singular if $n > d$. [And sometimes otherwise.]
In that case, no solution if $\lambda = 0$. [Then the penalty term is necessary. But that's okay.]

Summary of kernel ridge reg.:

$$\begin{array}{ll} K_{ij} = k(X_i, X_j) \quad \forall i, j & \Leftarrow O(n^2 d) \text{ time} \\ \text{Solve } (K + \lambda I)\, a = y \quad \text{for } a & \Leftarrow O(n^3) \text{ time} \\ \text{for each test pt } z & \\ \quad h(z) = \sum_{i=1}^{n} a_i\, k(X_i, z) & \Leftarrow O(nd) \text{ time} \end{array}$$

Does not use X directly! Only k. [This will become important soon.]

Dual: solve $n \times n$ linear system
Primal: solve $d \times d$ linear system
[So we prefer the dual form when $d > n$. If we add new features like polynomial terms, the d in the primal running time increases, but we will see that the d in the kernel running time does not.]

The Kernel Trick (aka kernelization)

[Here's the magic part. We will see that we can compute a polynomial kernel that involves many monomial terms without actually computing those terms.]

The polynomial kernel of degree p is $k(x, z) = (x^\top z + 1)^p$

Theorem: $(x^\top z + 1)^p = \Phi(x)^\top \Phi(z)$ where $\Phi(x)$ contains every monomial in x of degree $0 \ldots p$.

Example for $d = 2$, $p = 2$:

$$\begin{aligned} (x^\top z + 1)^2 &= x_1^2 z_1^2 + x_2^2 z_2^2 + 2x_1 z_1 x_2 z_2 + 2x_1 z_1 + 2x_2 z_2 + 1 \\ &= \begin{bmatrix} x_1^2 & x_2^2 & \sqrt{2}x_1 x_2 & \sqrt{2}x_1 & \sqrt{2}x_2 & 1 \end{bmatrix} \begin{bmatrix} z_1^2 & z_2^2 & \sqrt{2}z_1 z_2 & \sqrt{2}z_1 & \sqrt{2}z_2 & 1 \end{bmatrix}^\top \\ &= \Phi(x)^\top \Phi(z) \qquad \text{[This is how we're defining } \Phi\text{.]} \end{aligned}$$

[Notice the factors of $\sqrt{2}$. If you try a higher polynomial degree p, you'll see a wider variety of these constants. We have no control of the constants used in $\Phi(x)$, but they don't matter very much, because the primal weights w will scale themselves to compensate. Even though we aren't directly computing the primal weights ... they still implicitly exist.]

Key win: compute $\Phi(x)^\top \Phi(z)$ in $O(d)$ time instead of $O(d^p)$,
even though $\Phi(x)$ has length $O(d^p)$.

Kernel ridge regression replaces X_i with $\Phi(X_i)$:

Let $k(x, z) = \Phi(x)^\top \Phi(z)$

[I think what we've done here is pretty mind-blowing: we can now do polynomial regression with an exponentially long, high-order polynomial in less time than it would take even to compute the final polynomial. The running time is sublinear, actually much smaller than linear, in the size of the Φ vectors.]

14 More Kernelized Algorithms; Subset Selection; Lasso

Kernel Perceptrons

Note: Everywhere below, we can replace X_i with $\Phi(X_i)$ [that's usually our motivation for using a kernel]

Recall perceptron alg:

while some $y_i X_i \cdot w < 0$
  $w \leftarrow w + \epsilon\, y_i X_i$
for each test pt z
  $h(z) \leftarrow w^\top z$

Kernelize with $w = X^\top a$: $X_i \cdot w = (XX^\top a)_i = (Ka)_i$

Dual perceptron alg:

$a = [y_1 \quad 0 \quad \dots \quad 0]^\top$ [starting point is arbitrary, but can't be 0]
$K_{ij} = k(X_i, X_j) \qquad \forall i, j$ $\Leftarrow O(n^2 d)$ time (kernel trick)
while some $y_i\,(Ka)_i < 0$
  $a_i \leftarrow a_i + \epsilon\, y_i$ $\Leftarrow$ optimization: can update Ka in $O(n)$ time
for each test pt z
  $h(z) \leftarrow \sum_{j=1}^{n} a_j\, k(X_j, z)$ $\Leftarrow O(nd)$ time

[The d's above do not increase if we replace X_i with a kernelized $\Phi(X_i)$!]
OR we can compute $w = X^\top a$ once in $O(nd')$ time
  & evaluate test pts in $O(d')$ time/pt, where d' is length of $\Phi(\cdot)$

Interpretation: a_i records the multiple of X_i we have added to w [in the primal view of the problem]

Kernel Logistic Regression

[The stochastic gradient ascent step for logistic regression is just a small modification of the step for perceptrons. However, remember that we're no longer looking for misclassified sample points; instead, we apply the gradient ascent rule to weights in a stochastic, random order—or, alternatively, to all the weights at once.]

Stochastic gradient ascent step:

$a_i \leftarrow a_i + \epsilon\,(y_i - s((Ka)_i))$ [where s is the logistic function]

[Just like with perceptrons, every time you update one weight a_i, if you're clever you can update Ka in $O(n)$ time so you don't have to compute it from scratch on the next iteration.]

Gradient ascent step:

$a \leftarrow a + \epsilon\,(y - s(Ka))$ $\Leftarrow$ applying s component-wise to vector Ka

for each test pt z:

$$h(z) \leftarrow s\left(\sum_{j=1}^{n} a_j\, k(X_j, z)\right)$$

[or, if you're using logistic regression as a classifier and you don't care about the probability, you can skip the logistic fn and just compute the sum.]

The Gaussian Kernel

[Mind-blowing as the polynomial kernel is, I think our next trick is even more mind-blowing. Since we can now do fast computations in spaces with exponentially large dimension, why don't we go all the way and do computations in infinite-dimensional space?]

Gaussian kernel, aka radial basis fn kernel:
There exists a $\Phi(x)$ such that

$$k(x,z) = \exp\left(-\frac{|x-z|^2}{2\sigma^2}\right)$$

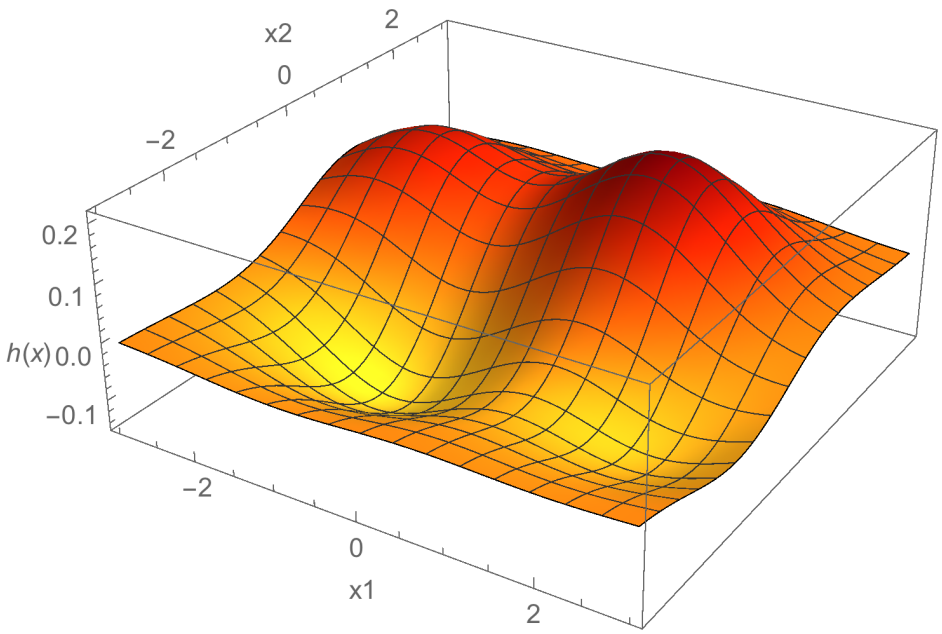
[In case you're curious, here's the feature vector that gives you this kernel, for the case where you have only one input feature per sample point.]

e.g. for $d = 1$,

$$\Phi(x) = \exp\left(-\frac{x^2}{2\sigma^2}\right)\left[1, \frac{x}{\sigma\sqrt{1!}}, \frac{x^2}{\sigma^2\sqrt{2!}}, \frac{x^3}{\sigma^3\sqrt{3!}}, \ldots\right]^\top$$

[This is an infinite vector, and $\Phi(x)^\top\Phi(z)$ is a converging series. Nobody actually uses this value of $\Phi(x)$ directly, or even cares about it; they just use the kernel function $k(\cdot,\cdot)$.]
[At this point, it's best *not* to think of points in a high-dimensional space. Instead, think of the kernel k as a measure of how similar or close together two points are to each other.]

Key observation: hypothesis $h(z) = \sum_{j=1}^{n} a_j\, k(X_j, z)$ is a linear combo of Gaussians centered at sample points.
[The dual weights are the coefficients of the linear combo.]
[Think of the Gaussians as a basis for the hypothesis.]



gausskernel.pdf [A hypothesis h that is a linear combination of Gaussians centered at four sample points, two with positive weights and two with negative weights.]

Very popular in practice! Why?

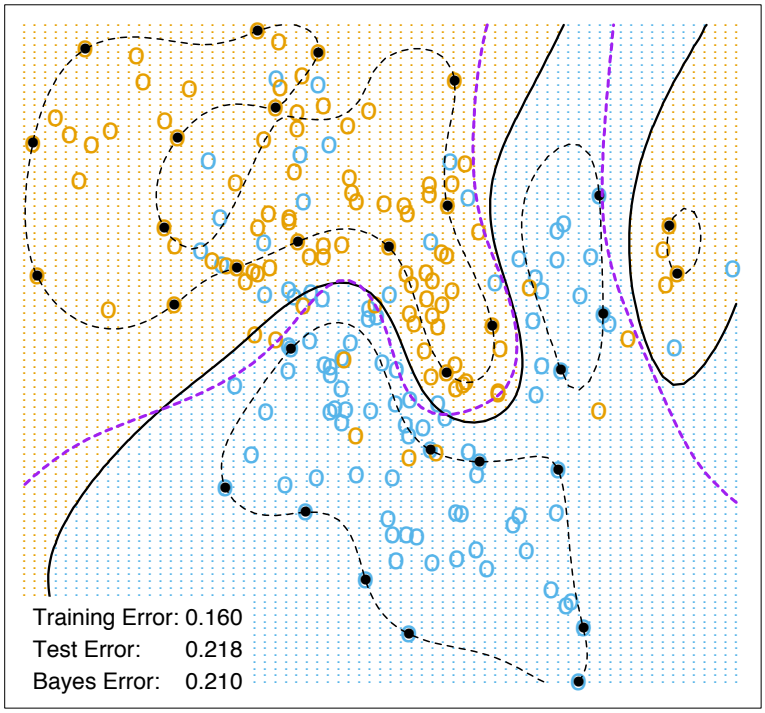
- Gives very smooth h
- Behaves somewhat like k-nearest neighbor, but smoother
- Oscillates less than polynomials (depending on σ)
- $k(x, z)$ can be interpreted as a "similarity measure." Gaussian is maximum when $z = x$, goes to 0 as distance increases.
- Sample points "vote" for value at z, but closer points get weightier vote.

[The "standard" kernel $x \cdot z$ assigns high value to vectors that point in roughly the same direction. By contrast, the Gaussian kernel assigns high value to points near each other.]

σ trades off bias vs. variance:

larger σ $\rightarrow$ wider Gaussians, smoother h $\rightarrow$ more bias, less variance

Choose by (cross-)validation.



gausskernelsvm.pdf (ESL, Figure 12.3) [The decision boundary (solid black) of an SVM with a Gaussian kernel. Observe that in this example, it comes reasonably close to the Bayes optimal decision boundary (dashed purple).]

[By the way, there are many other kernels like this that are defined directly as kernel functions without worrying about Φ. But not every function can be a kernel function. A function is qualified only if it always generates a positive semidefinite kernel matrix, for every sample.]

SUBSET SELECTION

All features increase variance, but not all features reduce bias (much).

Idea: Identify poorly predictive features, ignore them (weight zero).
Less overfitting, lower test errors.
2nd motivation: Inference. Simpler models convey interpretable wisdom.

Useful in all classification & regression methods.

Sometimes it's hard: Different features can partly encode same information.
Combinatorially hard to choose best feature subset.

Alg: Best subset selection. Try all $2^d - 1$ nonempty subsets of features.
Choose the best model by (cross-)validation. Slow.

[Obviously, best subset selection isn't tractable if we have a lot of features. But it gives us an "ideal" algorithm to compare practical algorithms with. If d is large, there is no algorithm that's both guaranteed to find the best subset and that runs in acceptable time. But heuristics often work well.]

Heuristic 1: Forward stepwise selection.
Start with null model (0 features); repeatedly add best feature until
test errors start increasing (due to overfitting) instead of decreasing.
At each outer iteration, inner loop tries every feature and chooses the best
by cross-validation. Requires training $O(d^2)$ models instead of $O(2^d)$.

Not perfect: e.g. won't find the best 2-feature model if neither of those
features yields the best 1-feature model.
[That's why it's a heuristic.]

Heuristic 2: Backward stepwise selection.
Start with all d features; repeatedly remove feature whose removal gives
best reduction in test errors. Also trains $O(d^2)$ models.
Additional heuristic: Only try to remove features with small weights.
Q: small relative to what?
Recall: variance of least-squ. regr. is proportional to $\sigma^2(X^\top X)^{-1}$
z-score of weight w_i is $z_i = \frac{w_i}{\sigma\sqrt{v_i}}$ where v_i is ith diagonal entry of $(X^\top X)^{-1}$.
Small z-score hints "true" w_i could be zero.

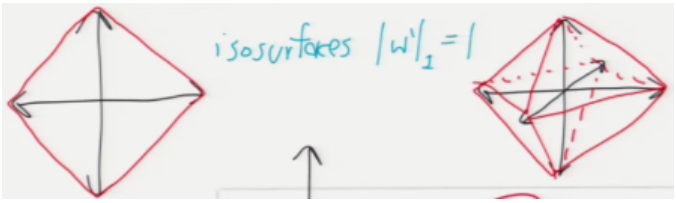
[Forward stepwise selection is a better choice when you suspect only a few features will be good predictors. Backward stepwise is better when you suspect most of the features will be necessary. If you're lucky, you'll stop early.]

LASSO (Robert Tibshirani, 1996)

Regression w/regularization: (1) + (A) + ℓ_1 penalized mean loss (e).
"Least absolute shrinkage and selection operator"
[This is a regularized regression method similar to ridge regression, but it has the advantage that it often naturally sets some of the weights to zero.]

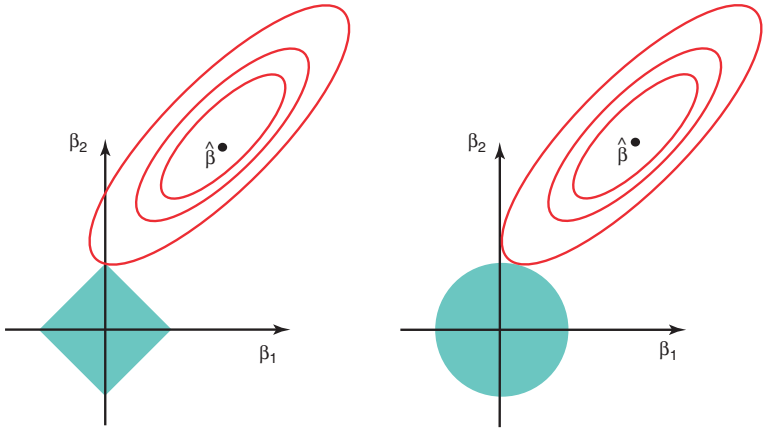
Find w that minimizes $|Xw - y|^2 + \lambda |w'|_1$ where $|w'|_1 = \sum_{i=1}^{d} |w_i|$ (Don't penalize α.)

Recall ridge regr.: isosurfaces of $|w'|^2$ are hyperspheres.
The isosurfaces of $|w'|_1$ are cross polytopes.
The unit cross-polytope is the convex hull of all the positive and negative unit coordinate vectors.



[Draw this figure by hand crosspolys.png]

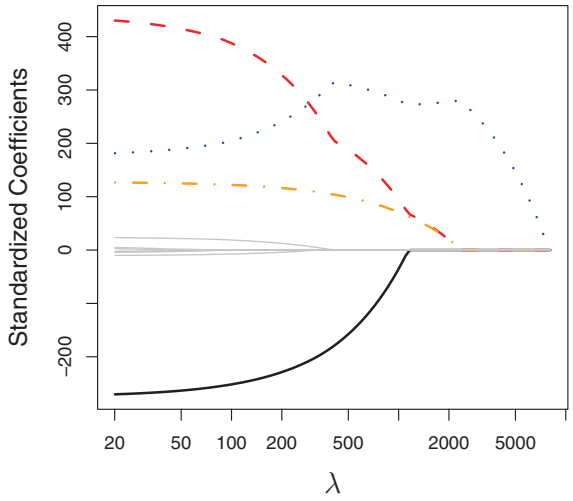
[You get larger and smaller cross-polytope isosurfaces by scaling these.]



lassoridge.pdf [Isocontours of the terms of the objective function for the Lasso appear at left. (The ridge regression terms appear at right for comparison.)]

[The red ellipses are the isocontours of $|Xw - y|^2$, and the least-squares solution lies at their center. The isocontours of $|w'|_1$ are diamonds centered at the origin (blue). The solution lies where a red isocontour just touches a blue diamond. What's interesting here is that in this example, the red isocontour touches just the tip of the diamond. So the weight w_1 gets set to zero. That's what we want to happen to weights that don't have enough influence. This doesn't always happen—for instance, the red isosurface could touch a side of the diamond instead of a tip of the diamond.]

[When you go to higher dimensions, you might have several weights set to zero. For example, in 3D, if the red isosurface touches a sharp corner of the cross-polytope, two of the three weights get set to zero. If it touches a sharp edge of the cross-polytope, one weight gets set to zero. If it touches a flat side of the cross-polytope, no weight is zero.]



lassoweights.pdf (ISL, Figure 6.6) [Weights as a function of λ.]

[This shows the weights for a typical linear regression problem with about 10 variables. You can see that as lambda increases, more and more of the weights become zero. Only four of the weights are really useful for prediction; they're in color. Statisticians used to choose λ by looking at a chart like this and trying to eyeball a spot where there aren't too many predictors and the weights aren't changing too fast. But nowadays they prefer cross-validation.]

Sometimes sets some weights to zero, especially for large λ.
Algs: subgradient descent, least-angle regression (LARS), forward stagewise

[Lasso's objective function is not smooth, and that makes it tricky to optimize. I'm not going to teach you an optimization method for Lasso. If you need one, look up the last two of these algorithms. LARS is built into the R Programming Language for statistics.]

[As with ridge regression, you should probably normalize the features first before applying this method.]

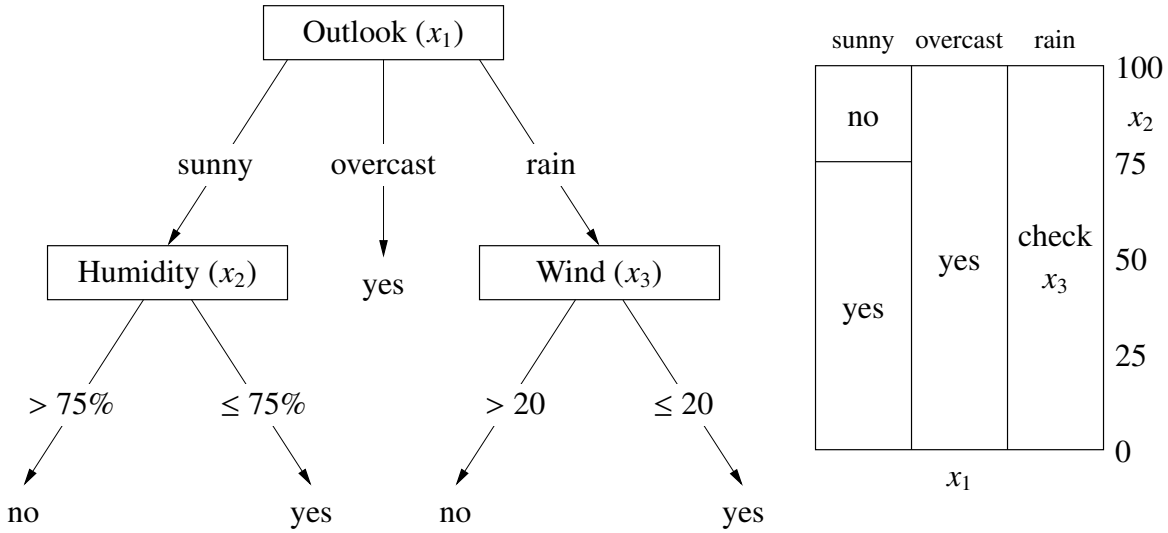
15 Decision Trees

DECISION TREES

Nonlinear method for classification and regression.

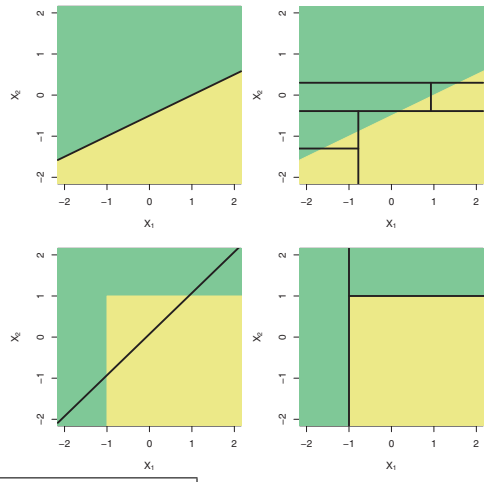
Uses tree w/two node types:

- internal nodes test feature values (usually just one) & branch accordingly
- leaf nodes specify class $h(x)$



[Draw this by hand. dectree.pdf]

- Cuts x-space into rectangular cells
- Works well with both categorical and quantitative features
- Interpretable result (inference)
- Decision boundary can be arbitrarily complicated



treelinearcompare.pdf (ISL, Figure 8.7) [Comparison of SVMs (left) vs. decision trees (right) on 2 examples.]

Consider classification first. Greedy, top-down learning heuristic:
[This algorithm is more or less obvious, and has been rediscovered many times. It's naturally recursive. I'll show how it works for classification first; later I'll talk about how it works for regression.]

Let $S \subseteq \{1, 2, \ldots, n\}$ be list of sample point indices.
Top-level call: $S = \{1, 2, \ldots, n\}$.

```
GrowTree(S)
    if (y_i = C for all i ∈ S and some class C) then {
        return new leaf(C)                    [We say the leaves are "pure"]
    } else {
        choose best splitting feature j and splitting value β     (*)
        S_l = {i : X_ij < β}                  [Or you could use ≤ and >]
        S_r = {i : X_ij ≥ β}
        return new node(j, β, GrowTree(S_l), GrowTree(S_r))
    }
```

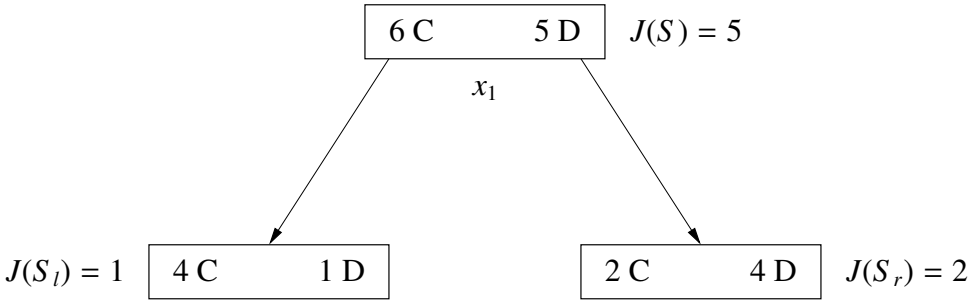
(*) How to choose best split?

- Try all splits. [All features, and all splits within a feature.]
- For a set S, let $J(S)$ be the cost of S.
- Choose the split that minimizes $J(S_l) + J(S_r)$; or,
  the split that minimizes weighted average $\frac{|S_l|J(S_l)+|S_r|J(S_r)}{|S_l|+|S_r|}$.

[Here, I'm using the vertical brackets $|\cdot|$ to denote set cardinality.]

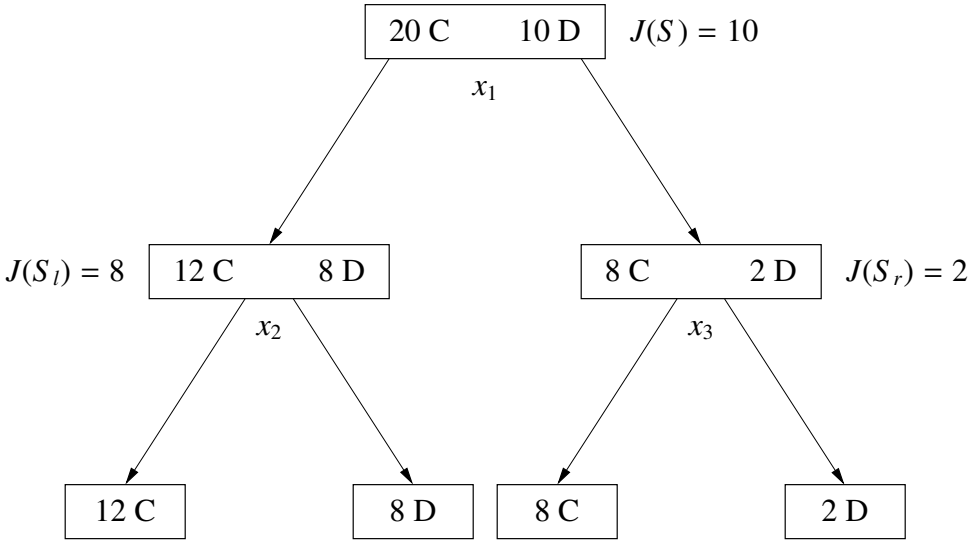
How to choose cost $J(S)$?
[I'm going to start by suggesting a mediocre cost function, so you can see why it's mediocre.]

Idea 1 (bad): Label S with the class C that labels the most points in S.
$J(S) \leftarrow$ # of points in S not in class C.



[Draw this by hand. badcost.pdf]

Problem: Sometimes we make "progress," yet $J(S_l) + J(S_r) = J(S)$.



[Draw this by hand. delayed.pdf]

[Notice that even though the first split doesn't reduce the total cost at all, we're still making progress, because after one more level of splits, we're done!]

Idea 2 (good): Measure the entropy. [An idea from information theory.]
Let Y be a random class variable, and suppose $P(Y = \mathrm{C}) = p_{\mathrm{C}}$.
The surprise of Y being class C is $S(Y = \mathrm{C}) = -\log_2 p_{\mathrm{C}}$. [Always nonnegative.]

- event w/prob. 1 gives us zero surprise.
- event w/prob. 0 gives us infinite surprise!

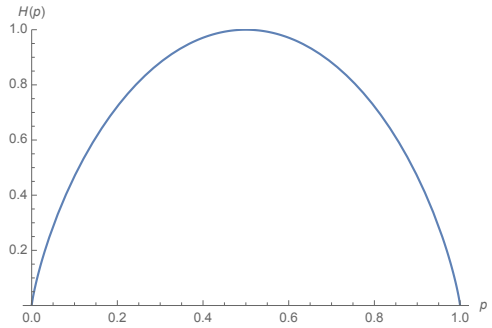
[In information theory, the surprise is equal to the expected number of bits of information we need to transmit which events happened to a recipient who knows the probabilities of the events. Often this means using fractional bits, which may sound crazy, but it makes sense when you're compiling lots of events into a single message; e.g. a sequence of biased coin flips.]

The entropy of an index set S is the average surprise

$$H(S) = -\sum_{\mathrm{C}} p_{\mathrm{C}} \log_2 p_{\mathrm{C}}, \qquad \text{where } p_{\mathrm{C}} = \frac{|\{i \in S : y_i = \mathrm{C}\}|}{|S|}.$$

[The proportion of points in S that are in class C.]

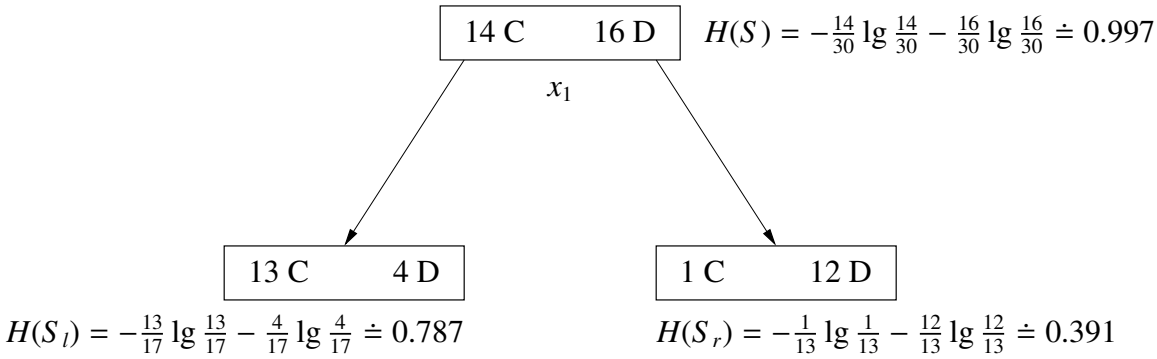
If all points in S belong to same class? $H(S) = -1 \log_2 1 = 0$.
Half class C, half class D? $H(S) = -0.5 \log_2 0.5 - 0.5 \log_2 0.5 = 1$.
n points, all different classes? $H(S) = -\log_2 \frac{1}{n} = \log_2 n$.



entropy.pdf [Plot of entropy $H(p_C)$ when there are only two classes.]

Weighted avg entropy after split is $H_{\text{after}} = \frac{|S_l|H(S_l)+|S_r|H(S_r)}{|S_l|+|S_r|}$.

Choose split that maximizes information gain $H(S) - H_{\text{after}}$.

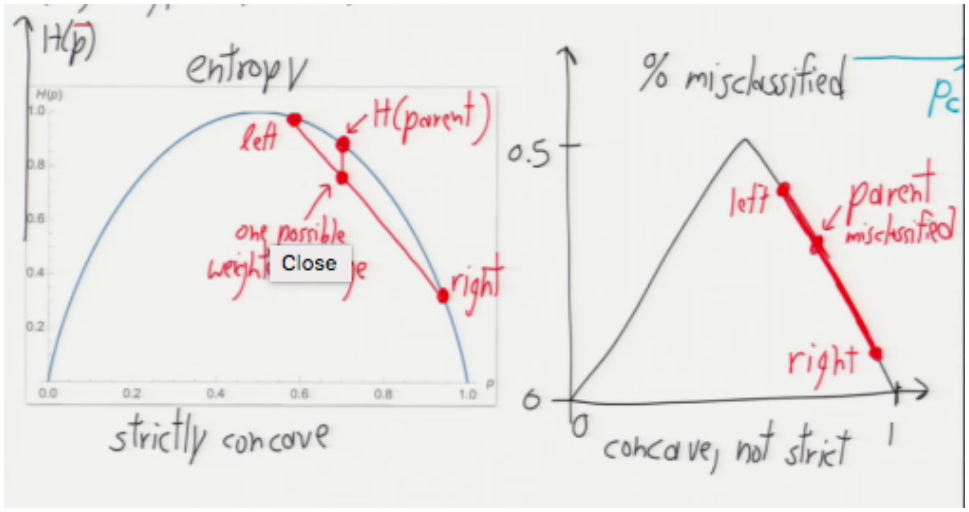


[Draw this by hand. infogain.pdf]

$H_{\text{after}} = 0.615$; info gain = 0.382

Info gain always positive *except* when one child is empty or for all C, $P(y_i = \text{C}|i \in S_l) = P(y_i = \text{C}|i \in S_r)$:

[Recall graph of entropy.]

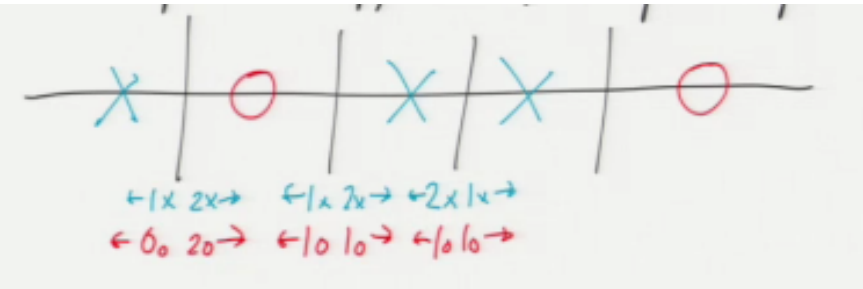


[Draw this by hand on entropy.pdf. concave.png]

[Suppose we pick two points on the entropy curve, then draw a line segment connecting them. Because the entropy curve is strictly concave, the interior of the line segment is strictly below the curve. Any point on that segment represents a weighted average of the two entropies for suitable weights. Whereas the point directly above that point represents the entropy if you unite the two sets into one. So the union of two nonempty sets has greater entropy than the weighted average of the entropies of the two sets, unless the two sets both have exactly the same p.]
[On the other hand, for the graph on the right, showing the % misclassified, if we draw a line segment connecting two points on the curve, the segment might lie entirely on the curve. In that case, uniting the two sets into one, or splitting one into two, changes neither the total misclassified sample points nor the % misclassified.]
[By the way, the entropy is not the only function that works well. Many concave functions work fine, including the simple polynomial $p(1-p)$.]

More on choosing a split:

- For binary feature x_i, children are $x_i = 0$ & $x_i = 1$.
- If x_i has 3+ discrete values, split depends on application.
  [Sometimes it makes sense to use multiway splits; sometimes binary splits.]
- If x_i is quantitative, sort points in S by feature x_i; remove duplicates; [duplicate values, not points] try splitting between each pair of consecutive values.
  [We can radix sort the values in linear time, and if n is huge we should.]
  Clever Bit: As you scan sorted list from left to right, you can update entropy in $O(1)$ time per point!
  [This is important for obtaining a fast tree-building time.]
  [Draw a row of X's and O's; show how we update the # of X's and # of O's in each of S_l and S_r as we scan from left to right.]



scan.png

Algs & running times:

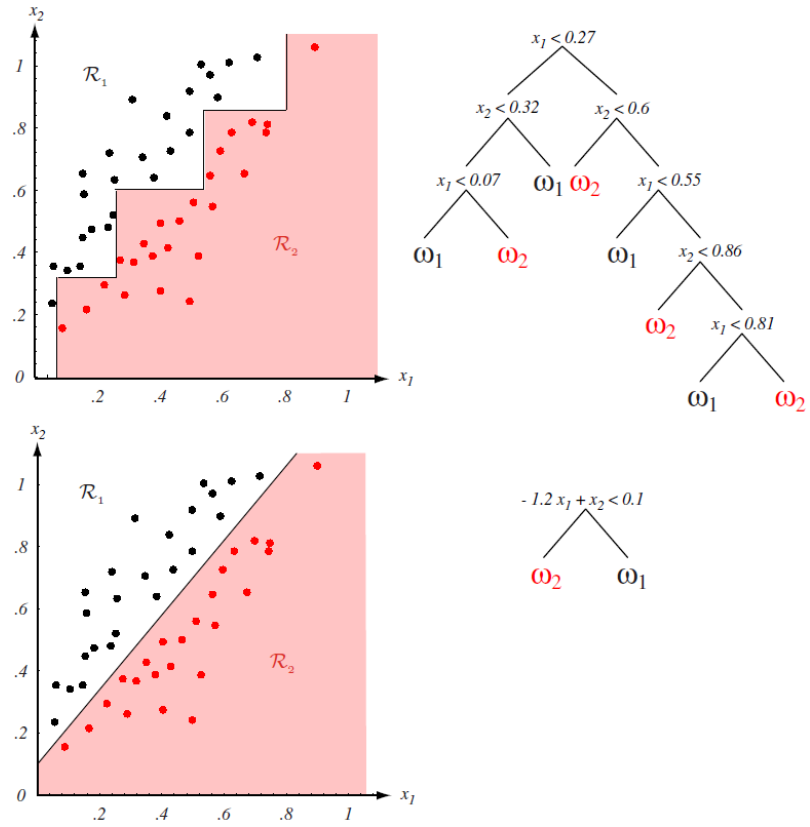
- Test point: Walk down tree until leaf. Return its label.
  Worst-case time is O(tree depth).
  For binary features, that's $\leq d$. [Quantitative features may go deeper.]
  Usually (not always) $\leq O(\log n)$.
- Training: For binary features, try $O(d)$ splits at each node.
  For quantitative features, try $O(n'd)$ splits; n' = points in node
  Either way $\Rightarrow O(n'd)$ time at this node
  [Quantitative features are asymptotically just as fast as binary features because of our clever way of computing the entropy for each split.]
  Each point participates in O(depth) nodes, costs $O(d)$ time in each node.
  Running time $\leq O(nd$ depth).
  [As nd is the size of the design matrix X, and the depth is often logarithmic, this is a surprisingly reasonable running time.]

16 More Decision Trees, Ensemble Learning, and Random Forests

DECISION TREES (continued)

Multivariate splits

Find non-axis-aligned splits with other classification algs or by generating them randomly.



multivariate.pdf [An example where an ordinary decision tree needs many splits to approximate a diagonal linear decision boundary, but a single multivariate split takes care of it.]

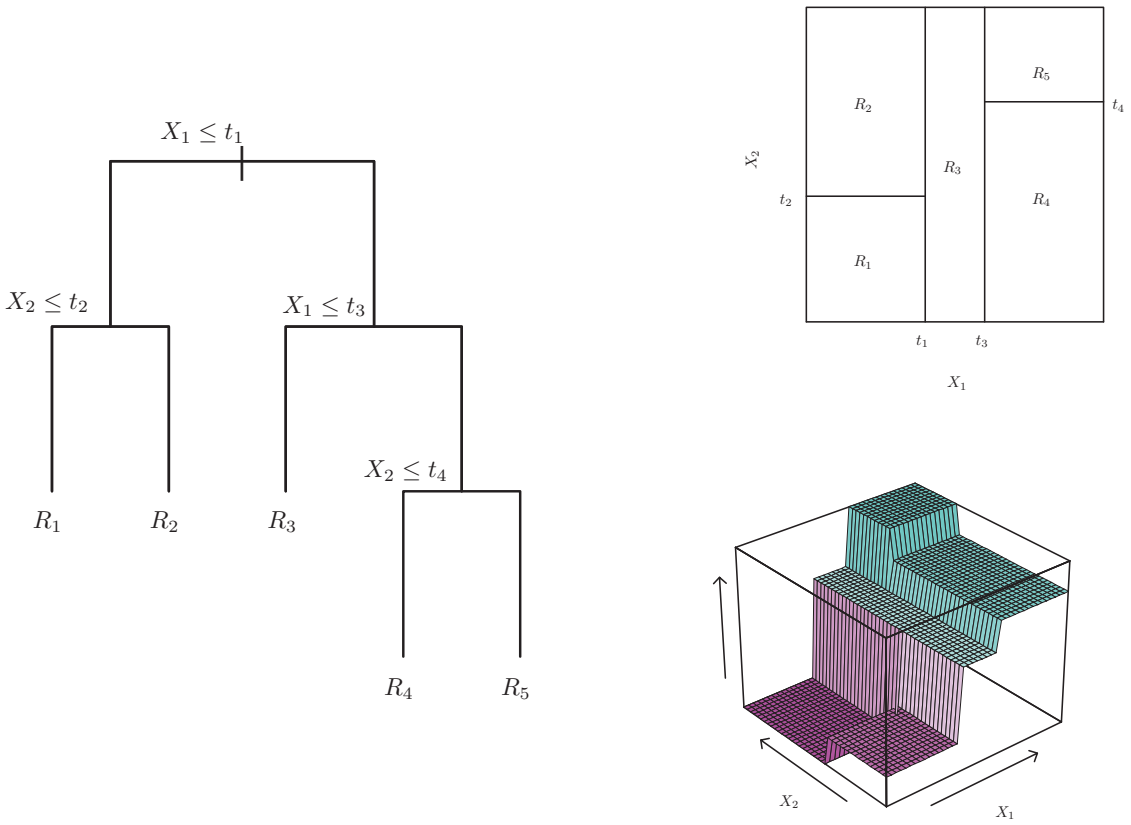
[This gives you all the power of classification algorithms such as SVMs, logistic regression, and Gaussian discriminant analysis; moreover, it can make them more powerful by making them hierarchical, so they're not limited to just one boundary.]

May gain better classifier at cost of worse interpretability or speed.

Can limit # of features per split: forward stepwise selection, Lasso.

Decision Tree Regression

Creates a piecewise constant regression fn.



regresstree.pdf, regresstreefn.pdf (ISL, Figure 8.3) [Decision tree regression.]

Cost $J(S) = \sum_{i \in S} (y_i - \bar{y})^2$, where $\bar{y}$ is the mean y_i for sample points in subset S.

[So if all the points in a node have the same y-value, then the cost is zero.]
[We choose the split that minimizes the weighted average of the costs of the children after the split.]

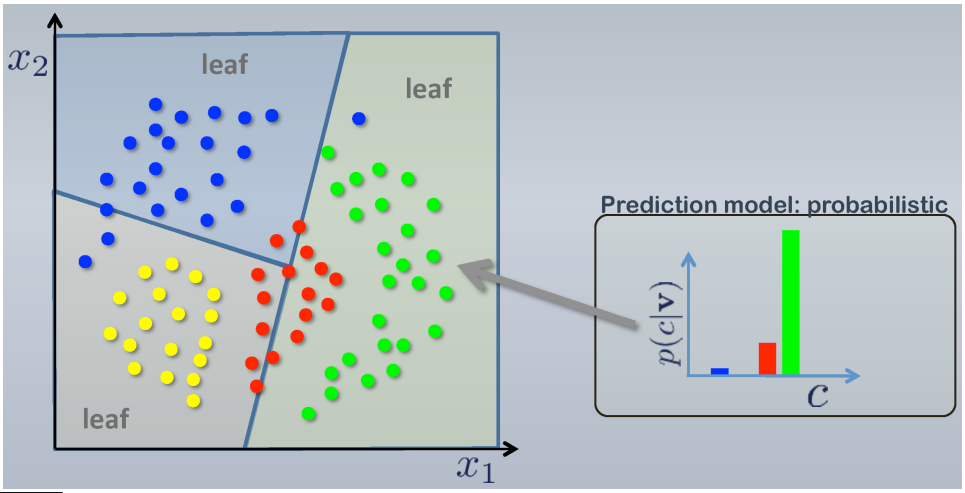
Stopping early

[The basic version of the decision tree algorithm keeps subdividing treenodes until every leaf is pure. We don't have to do that; sometimes we prefer to stop subdividing treenodes earlier.]

Why?

- Limit tree depth (for speed)
- Limit tree size (big data sets)
- Complete tree may overfit
- Given noise or overlapping distributions, purity of leaves is counterproductive; better to estimate posterior probs

[When you have overlapping class distributions, it's better to estimate a posterior probability than to always give a yes/no answer. Refining the tree down to one sample point per leaf is absolutely guaranteed to overfit. Whereas if you have enough points in each leaf, you can estimate posterior probabilities.]



leaf.pdf [In the decision tree at left, each leaf has multiple classes. Instead of returning the majority class, each leaf could return a posterior probability histogram, as illustrated at right.]

How? Select stopping condition(s):

- Next split doesn't reduce entropy/error enough (dangerous; pruning is better)
- Most of node's points (e.g. > 95%) have same class [to deal with outliers]
- Node contains few sample points (e.g. < 10)
- Node covers tiny volume
- Depth too great
- Use (cross-)validation to compare

[The last is the slowest but most effective way to know when to stop: use validation to decide whether splitting the node is a win or a loss. But if your goal is to avoid overfitting, it's generally even more effective to grow the tree a little too large and then use validation to prune it back. We'll talk about that next.]

Leaves with multiple points return

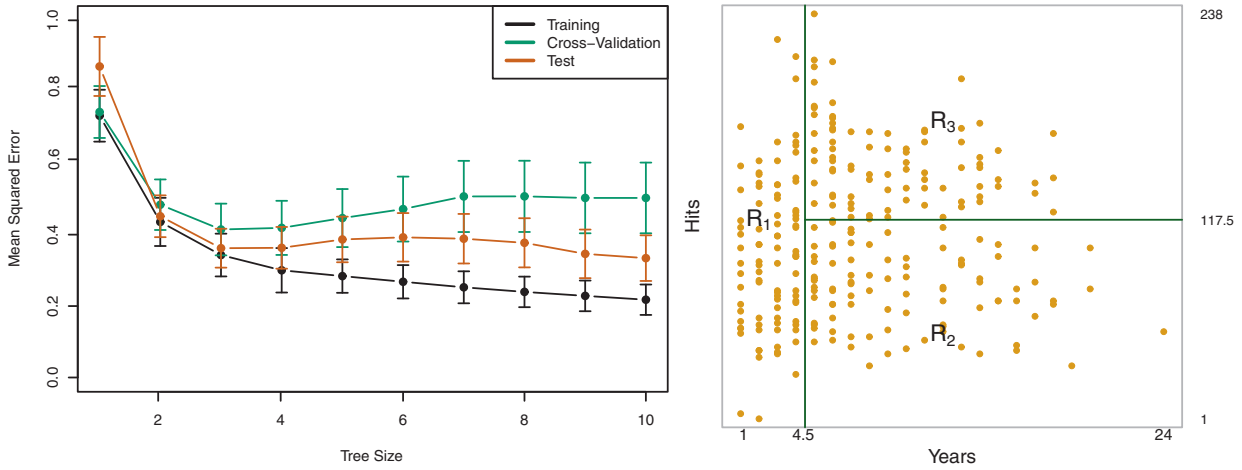
- a majority vote or class posterior probs (classification) or
- an average (regression).

Pruning

Grow tree too large; greedily remove each split whose removal improves cross-validation performance. More reliable than stopping early.

[We have to do cross-validation once for each split that we're considering removing. But you can do that pretty cheaply. What you *don't* do is reclassify every sample point from scratch. Instead, you keep track of which points in the validation set end up at which leaf. When you are deciding whether to remove a split, you just look at the points in the two leaves you're thinking of removing, and see how they will be reclassified and how that will change the error rate. You can do this very quickly.]

[The reason why pruning often works better than stopping early is because often a split that doesn't seem to make much progress is followed by a split that makes a lot of progress. If you stop early, you'll never get there.]



prunehitters.pdf, prunedhitters.pdf (ISL, Figures 8.5 & 8.2) [At left, a plot of decision tree size vs. errors for baseball hitter data. Cross-validation indicates that the best decision tree has three leaves; it appears at right. Players' salaries: R1 = \$165,174, R2 = \$402,834, R3 = \$845,346.]

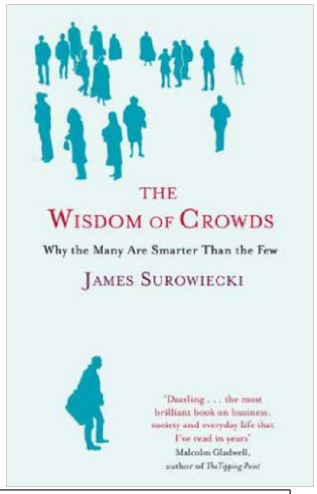
ENSEMBLE LEARNING

Decision trees are fast, simple, interpretable, easy to explain,
invariant under scaling/translation, robust to irrelevant features.

But not the best at prediction. [Compared to previous methods we've seen.]
High variance.

[For example, suppose we take a training data set, split it into two halves, and train two decision trees, one on each half of the data. It's not uncommon for the two trees to turn out very different. In particular, if the two trees pick different features for the very first split at the top of the tree, then it's quite common for the trees to be completely different. So decision trees tend to have high variance.]

[So let's think about how to fix this. As an analogy, imagine that you are generating random numbers from some distribution. If you generate just one number, it might have high variance. But if you generate n numbers and take their average, then the variance of that average is n times smaller. So you might ask yourself, can we reduce the variance of decision trees by taking an average answer of a bunch of decision trees? Yes we can.]



wisdom.jpg, penelope.jpg [James Surowiecki's book "The Wisdom of Crowds" and Penelope the cow. Surowiecki tells us this story . . .]

[A 1906 county fair in Plymouth, England had a contest to guess the weight of an ox. A scientist named Francis Galton was there, and he did an experiment. He calculated the median of everyone's guesses. The median guess was 1,207 pounds, and the true weight was 1,198 pounds, so the error was less than 1%. Even the cattle experts present didn't estimate it that accurately.]
[NPR repeated the experiment in 2015 with a cow named Penelope whose photo they published online. They got 17,000 guesses, and the average guess was 1,287 pounds. Penelope's actual weight was 1,355 pounds, so the crowd got it to within 5 percent.]
[The main idea is that sometimes the average opinion of a bunch of idiots is better than the opinion of one expert. And so it is with learning algorithms. We call a learning algorithm a "weak learner" if it does better than guessing randomly. And we combine a bunch of weak learners to get a strong one.]
[Incidentally, James Surowiecki, the author of the book, guessed 725 pounds for Penelope. So he was off by 87%. He's like a bad decision tree who wrote a book about how to benefit from bad decision trees.]

We can take average of output of:
- different learning algs
- same learning alg on many training sets [if we have tons of data]
- bagging: same learning alg on many random subsamples of one training set
- random forests: randomized decision trees on random subsamples

[These last two are the most common ways to use averaging, because usually we don't have enough training data to use fresh data for every learner.]

[Averaging is not specific to decision trees; it can work with many different learning algorithms. But it works particularly well with decision trees.]

Regression algs: take median or mean output
Classification algs: take majority vote OR average posterior probs

[Apology to readers: I show some videos in this lecture, which cannot be included in this report.]

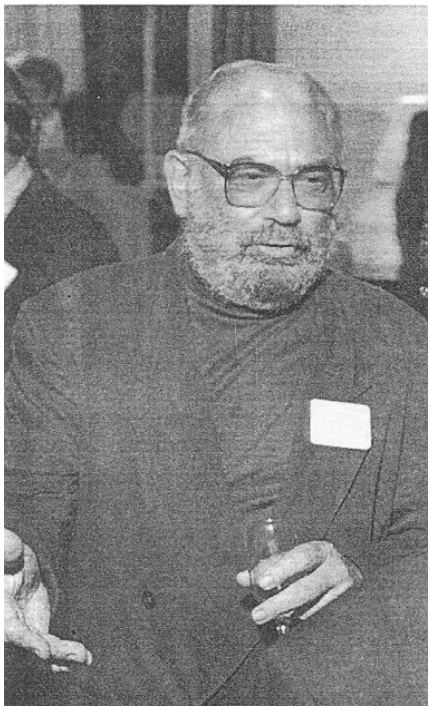
[Show averageaxis.mov] [Here's a simple classifier that takes an average of "stumps," trees of depth 1. Observe how good the posterior probabilities look.]
[Show averageaxistree.mov] [Here's a 4-class classifier with depth-2 trees.]

[The Netflix Prize was an open competition for the best collaborative filtering algorithm to predict user ratings for films, based on previous ratings. It ran for three years and ended in 2009. The winners used an extreme ensemble method that took an average of many different learning algorithms. In fact, a couple of top teams combined into one team so they could combine their methods. They said, "Let's average our models and split the money," and that's what happened.]

Use learners with low bias (e.g. deep decision trees).
High variance & some overfitting is okay. Averaging reduces the variance!
[Each learner may overfit, but each overfits in its own unique way.]
Averaging sometimes reduces the bias & increases flexibility;
  e.g. creating nonlinear decision boundary from linear classifiers.
Hyperparameter settings usually different than 1 learner.
[Because averaging learners reduces their variance. But averaging rarely reduces bias as much as it reduces variance, so you want to get the bias nice and small before you average.]
# of trees is another hyperparameter.

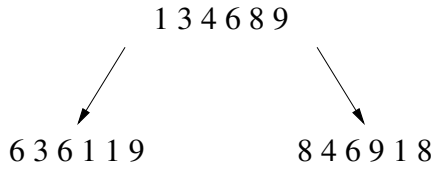
Bagging = Bootstrap AGGregatING (Leo Breiman, 1994)

[Leo Breiman was a statistics professor right here at Berkeley. He did his best work after he retired in 1993. The bagging algorithm was published the following year, and then he went on to co-invent random forests as well. Unfortunately, he died in 2005.]



breiman.gif [Leo Breiman]

[Bagging is a randomized method for creating many different learners from the same data set. It works well with many different learning algorithms. One exception seems to be k-nearest neighbors; bagging mildly degrades it.]
Given n-point training sample, generate random subsample of size n' by sampling
with replacement. Some points chosen multiple times; some not chosen.



If $n' = n$, $\sim 63.2\%$ are chosen. [On average; this fraction varies randomly.]
Build learner. Points chosen j times have greater weight:
[If a point is chosen j times, we want to treat it the same way we would treat j different points all bunched up infinitesimally close together.]

- Decision trees: j-time point has $j \times$ weight in entropy.
- SVMs: j-time point incurs $j \times$ penalty to violate margin.
- Regression: j-time point incurs $j \times$ loss.

Repeat until T learners. Metalearner takes test point, feeds it into all T learners, returns average/majority output.

Random Forests

Random sampling isn't random enough!
[With bagging, often the decision trees look very similar. Why is that?]
One really strong predictor → same feature split at top of every tree.
[For example, if you're building decision trees to identify spam, the first split might always be "viagra." Random sampling doesn't change that. If the trees are very similar, then taking their average doesn't reduce the variance much.]

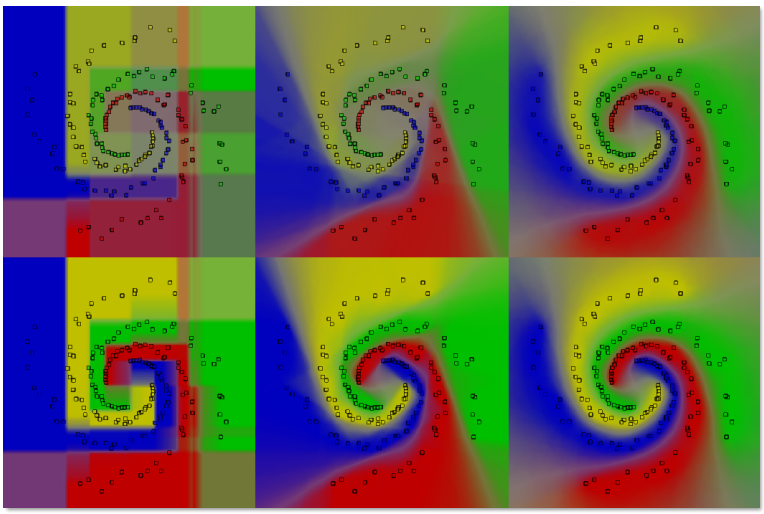
Idea: At each split, take random sample of m features (out of d).
Choose best split from m features.
[We're not allowed to split on the other $d - m$ features!]
Different random sample for each split.
$m \approx \sqrt{d}$ works well for classification; $m \approx d/3$ for regression.
[So if you have 100-dimensional feature space, you randomly choose 10 features and pick the one of those 10 that gives the best split. But m is a hyperparameter, and you might get better results by tuning it for your particular application.]
Smaller $m \rightarrow$ more randomness, less tree correlation, more bias

[One reason this works is if there's a really strong predictor, only a fraction of the trees can choose that predictor as the first split. That fraction is m/d. So the split tends to "decorrelate" the trees. And that means that when you take the average of the trees, you'll have less variance.]
[You have to be careful, though, because you don't want to dumb down the trees too much in your quest for decorrelation. Averaging works best when you have very strong learners that are also diverse. But it's hard to create a lot of learners that are very different yet all very smart. The Netflix Prize winners did it, but it was a huge amount of work.]

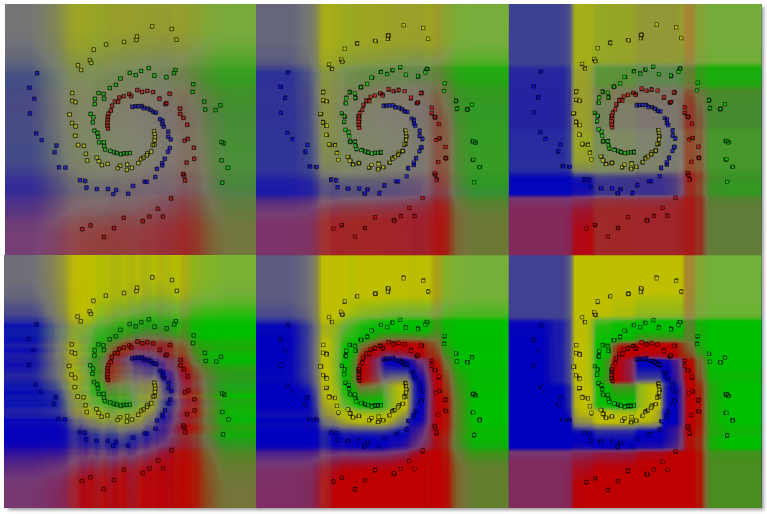
Sometimes test error reduction up to 100s or even 1,000s of decision trees!
Disadvantage: loses interpretability/inference.
[But the compensation is it's more accurate than a single decision tree.]

Variation: generate m random multivariate splits (oblique lines, quadrics); choose best split.
[You have to be a bit clever about how you generate random decision boundaries; I'm not going to discuss that today. I'll just show lots of examples.]

[Show treesidesdeep.mov] [Lots of good-enough conic random decision trees.]
[Show averageline.mov]
[Show averageconic.mov]
[Show square.mov] [Depth 2; look how good the posterior probabilities look.]
[Show squaresmall.mov] [Depth 2; see the uncertainty away from the center.]
[Show spiral2.mov] [Doesn't look like a decision tree at all, does it?]
[Show overlapdepth14.mov] [Overlapping classes. This example overfits!]
[Show overlapdepth5.mov] [Better fit.]



500.pdf [Decision trees for 4-class spiral data. The top row shows trees of depth 4. The bottom row shows trees of depth 12. From left to right, we have axis-aligned splits, splits with lines with arbitrary rotations, and splits with conic sections. Each split is chosen to be the best of 500 random choices.]



randomness.pdf [Decision trees for the same 4-class spiral data. In these examples, *all* the splits are axis-aligned. The top row shows trees of depth 4. The bottom row shows trees of depth 12. From left to right, we choose each split from 1, 5, or 50 random choices. The more choices, the better the tree.]

17 Neural Networks

NEURAL NETWORKS

Can do both classification and regression.

[They tie together several ideas from the course: perceptrons, logistic regression, ensembles of learners, and stochastic gradient descent. They also tie in the idea of lifting sample points to a higher-dimensional feature space, but with a new twist: neural nets can learn features themselves.]

[I want to begin by reminding you of the story I told you at the beginning of the semester, about Frank Rosenblatt's invention of perceptrons in 1957. Remember that he held a press conference where he predicted that perceptrons would be "the embryo of an electronic computer that [the Navy] expects will be able to walk, talk, see, write, reproduce itself and be conscious of its existence."]

[Perceptron research continued, but something monumental happened in 1969. Marvin Minsky, one of the founding fathers of AI, and Seymour Papert published a book called "Perceptrons." Sounds good, right? Well, part of the book was devoted to things perceptrons can't do. And one of those things is XOR.]

		x_1	
	XOR	0	1
x_2	0	0	1
	1	1	0

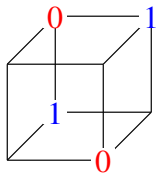
[Think of the four outputs here as sample points in two-dimensional space. Two of them are in class 1, and two of them are in class 0. We want to find a linear classifier that separates the 1's from the 0's. Can we do it? No.]

[So Minsky and Papert were basically saying, "Frank. You're telling us this machine is going to be conscious of its own existence but it can't do XOR?" That sounds like Taylor Swift; apparently she can't do exclusive or either.]

[The book had a devastating effect on the field. After its publication, almost no research was done on neural net-like ideas for a decade, a time we now call the "AI Winter." Shortly after the book was published, Frank Rosenblatt died. Officially, he died in a boating accident. But we all know he died of a broken heart.]

[One thing I don't understand is why the book was so fatal when there are some almost obvious ways to get around this problem. Here's the easiest.]

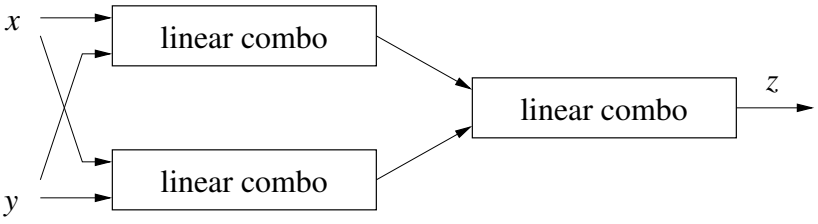
If you add one new quadratic feature, $x_1 x_2$, XOR is linearly separable in 3D.



[Draw this by hand. xorcube.pdf]

[Now we can find a plane that cuts through the cube obliquely and separates the 0's from the 1's.]

[However, there's an even more powerful way to do XOR. The idea is to design linear classifiers whose output is the input to other linear classifiers. That way, you should be able to emulate arbitrarily logical circuits. Suppose I put together some linear predictor functions like this.]



[Draw this by hand. lincombo.pdf]

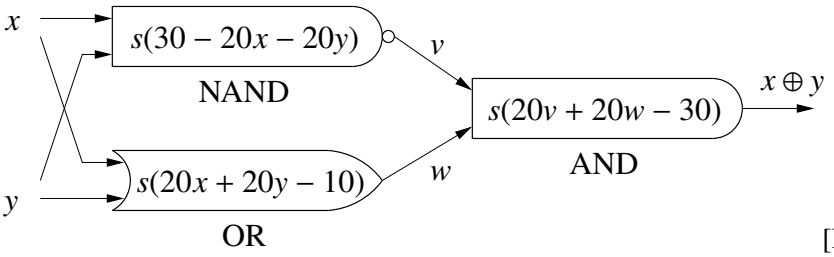
[If I interpret the output as 1 if z is positive or 0 is z is negative, can I do XOR with this?]

A linear combo of a linear combo is a linear combo . . . only works for linearly separable points.

[We need one more idea to make neural nets. We need to add some sort of nonlinearity between the linear combinations. Let's call these boxes that compute linear combinations "neurons." If a neuron runs the linear combination it computes through some nonlinear function before sending it on to other neurons, then the neurons can act somewhat like logic gates. The nonlinearity could be as simple as clamping the output so it can't go below zero. And that's actually used in practice sometimes.]

[The most popular traditional choice has been to use the logistic function. The logistic function can't go below zero or above one, which is nice because it can't ever get huge and oversaturate the other neurons it's sending information to. The logistic function is also smooth, which means it has well-defined gradients and Hessians we can use in gradient descent.]

[With logistic functions between the linear combinations, here's a two-level perceptron that computes the XOR function. Note that the logistic function at the output is optional; we could just take the sign of the output instead.]



[Draw this by hand. xorgates.pdf]

Network with 1 Hidden Layer

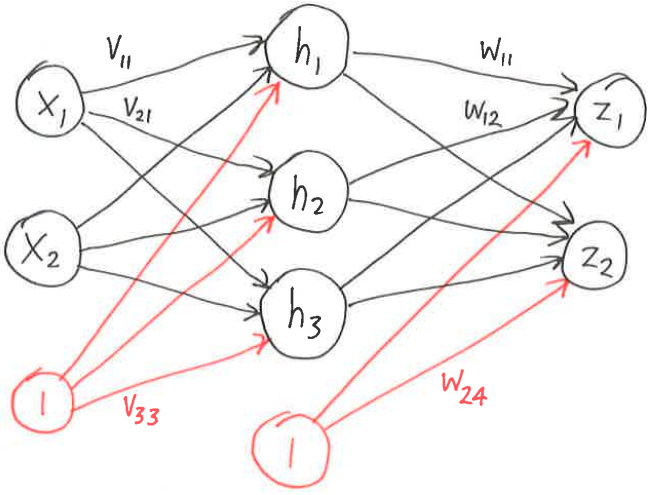
Input layer: $x_1, \ldots, x_d$; $x_{d+1} = 1$
Hidden units: $h_1, \ldots, h_m$; $h_{m+1} = 1$
Output layer: $z_1, \ldots, z_k$

[We might have more than one output so that we can build multiple classifiers that share hidden units. One of the interesting advantages of neural nets is that if you train multiple classifiers simultaneously, sometimes some of them come out better because they can take advantage of particularly useful hidden units that first emerged to support one of the other classifiers.]

Layer 1 weights: $m \times (d+1)$ matrix V $\quad$ V_i is row i
Layer 2 weights: $k \times (m+1)$ matrix W $\quad$ W_i is row i

Recall [logistic function] $s(\gamma) = \frac{1}{1+e^{-\gamma}}$. Other nonlinear fns can be used.

For vector v, $s(v) = \begin{bmatrix} s(v_1) \\ s(v_2) \\ \vdots \end{bmatrix}$. [We apply s to a vector component-wise.]



[Draw this by hand. 1hiddenlayer.pdf]

$$h = s(Vx) \qquad \ldots \text{that is, } h_i = s\left(\sum_{j=1}^{3} V_{ij} x_j\right)$$

$$z = s(Wh) = s(W s_1(Vx))$$

↑
add a 1 to end of vector

[We can add more hidden layers, and for some tasks it's common to have up to five hidden layers. There are many variations you can experiment with—for instance, you can have connections that go forward more than one layer.]

Training

Usually stochastic or batch gradient descent.

Pick loss fn $L(z, y)$ e.g. $L(z, y) = |z - y|^2$

$\uparrow\uparrow$

predictions true values (could be vectors)

Cost fn is $J(h) = \sum_{i=1}^{n} L(h(X_i), Y_i)$

[I'm using a capital Y here because now Y is a matrix with one row for each sample point and one column for each output of the neural net. Sometimes there is just one output, but many neural net applications have more.]

Usually there are many local minima!

[The cost function for a neural net is, generally, not even close to convex. For that reason, it's possible to wind up in a bad minimum. We'll talk later about some clever ways to coax neural nets into better minima.]

[Now let me ask you this. Suppose we start by setting all the weights to zero, and then we do gradient descent on the weights. What will go wrong?]

[This neural network has a symmetry: there's really no difference between one hidden unit and any other hidden unit. The gradient descent algorithm has no way to break the symmetry between hidden units. You can get stuck in a situation where all the weights in each layer have the same value, and they have no way to become different from each other. To avoid this problem, and in the hopes of finding a better minimum, we start with random weights.]

Rewrite all the weights in V & W as a vector w. Batch gradient descent:

$w \leftarrow$ vector of random weights
repeat
    $w \leftarrow w - \epsilon \nabla J(w)$

[It's important to make sure the random weights aren't too big, because if a unit's output gets too close to zero or one, it can get "stuck," meaning that it always has roughly the same output value regardless of the input. Stuck units tend to stay stuck because in that operating range, the gradient $s'(\cdot)$ of the logistic function is close to zero.]

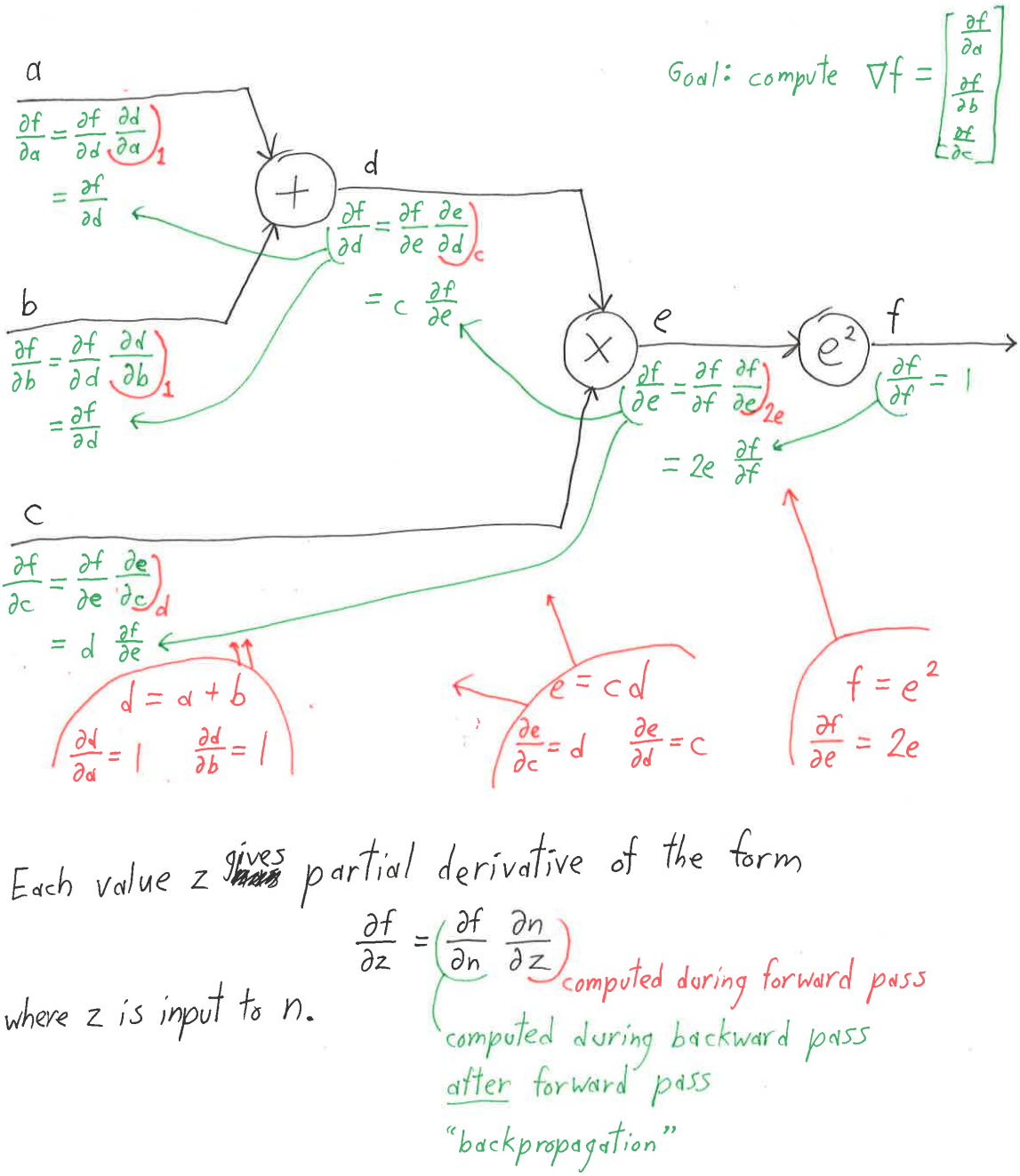
[Instead of batch gradient descent, we can use stochastic gradient descent, which means we use the gradient of one sample point's loss function at each step. Typically, we shuffle the points in a random order, or just pick one randomly at each step.]

[The hard part of this algorithm is computing the gradient. If you simply derive one derivative for each weight, you'll find that it takes time linear in the size of the neural network to compute a derivative for one weight in the first layer. Multiply that by the number of weights. We're going to spend some time learning to improve the running time to linear in the number of weights.]

Naive gradient computation: O(units $\times$ edges) time
Backpropagation: O(edges) time

Computing Gradients for Arithmetic Expressions

[Let's see what it takes to compute the gradient of an arithmetic expression. It turns into repeated applications of the chain rule from calculus.]



Each value z gives partial derivative of the form

$$\frac{\partial f}{\partial z} = \left(\frac{\partial f}{\partial n}\frac{\partial n}{\partial z}\right)$$

computed during forward pass

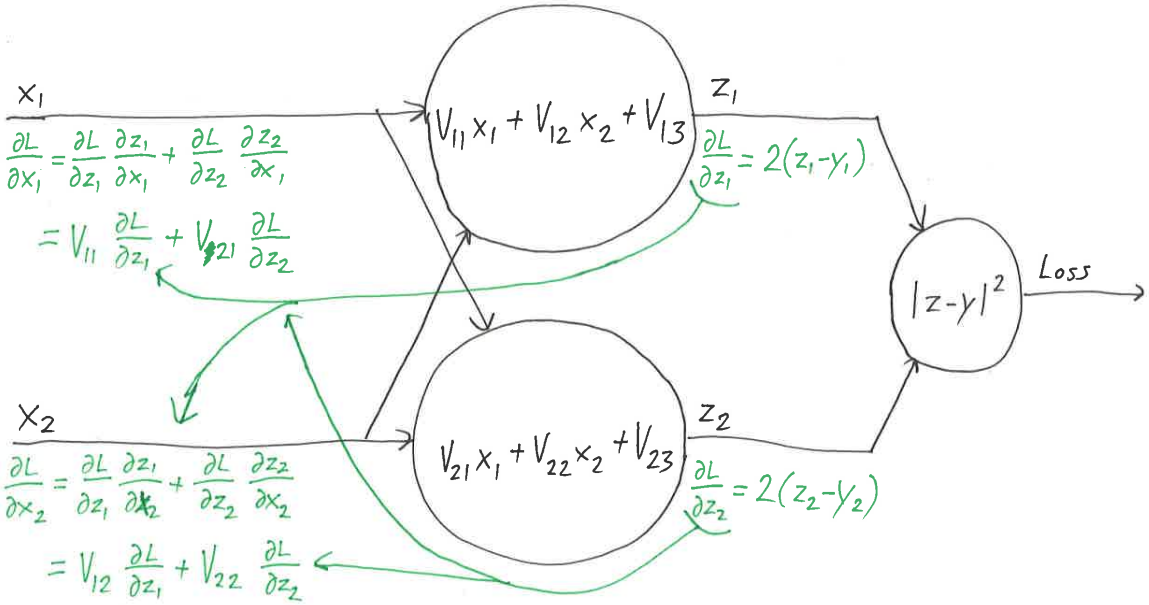
computed during backward pass after forward pass

"backpropagation"

where z is input to n.

[Draw this by hand. gradients.pdf Draw the black diagram first. Then the goal (upper right). Then the green and red expressions, from left to right, leaving out the green arrows. Then the green arrows, starting at the right side of the page and moving left. Lastly, write the text at the bottom. (Use the same procedure for the next two figures.)]

[What if a unit's output goes to more than one unit? Then we need to understand a more complicated version of the chain rule. Let's try it with an expression that's similar to what you'll encounter in a neural net.]



[Draw this by hand. gradientspartial.pdf]

[Observe that we're doing dynamic programming here. We're computing the solutions of subproblems, then using each solution to compute the solutions of several bigger problems.]

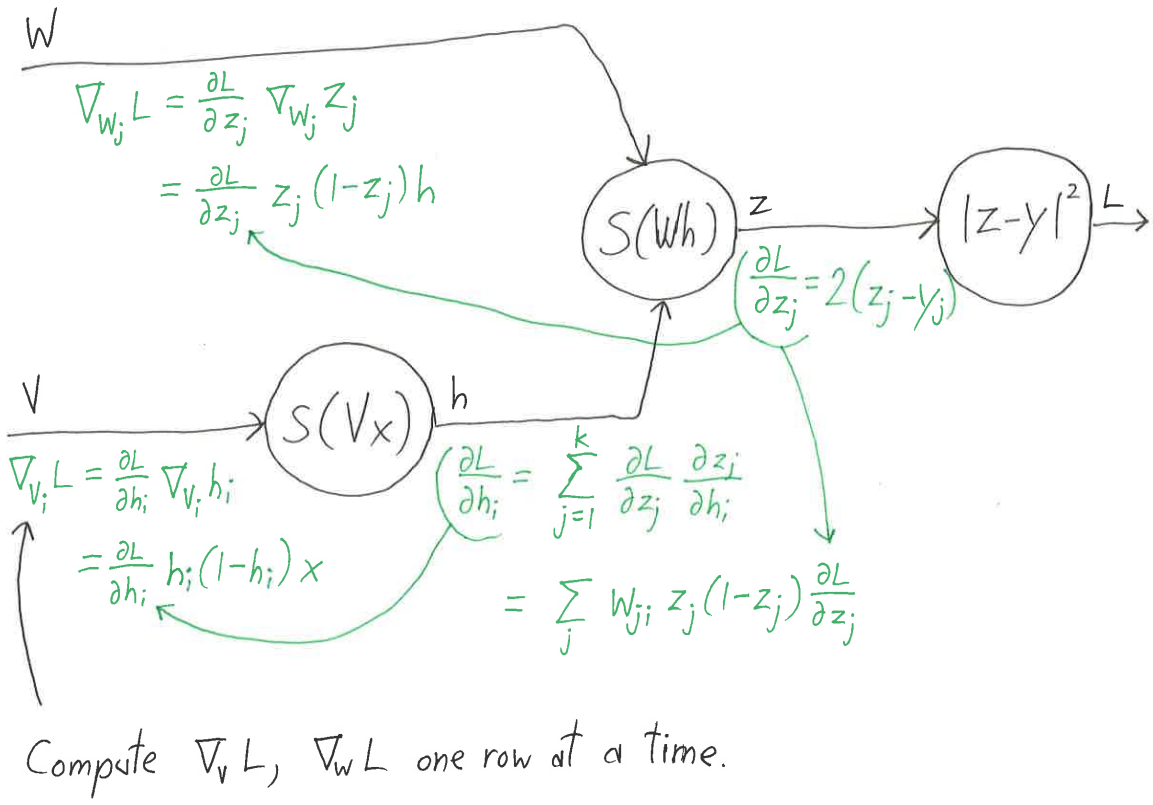
The Backpropagation Alg

[Backpropagation is a dynamic programming algorithm for computing the gradients we need to do neural net gradient descent in linear time.]

Recall $s'(\gamma) = s(\gamma)\,(1 - s(\gamma))$

$h_i = s(V_i \cdot x)$, so $\quad \nabla_{V_i} h_i = s'(V_i \cdot x)\, x = h_i\,(1 - h_i)\, x \qquad \nabla_{W_j} z_j = s'(W_j \cdot h)\, h = z_j\,(1 - z_j)\, h$

[Here is the arithmetic expression for the same neural network I drew for you three illustrations ago. It looks very different when you depict it like this, but don't be fooled; it's exactly the same network.]



[Draw this by hand. gradbackprop.pdf]

18 Neurons; Variations on Neural Networks

NEURONS

[The field of artificial intelligence started with some wrong premises. The early AI researchers attacked problems like chess and theory proving, because they thought those exemplified the essence of intelligence. They didn't pay much attention at first to problems like vision and speech understanding. Any four-year-old can do those things, and so researchers underestimated their difficulty. Today, we know better. Computers can effortlessly beat four-year-olds at chess, but they still can't play with toys nearly as well. We've come to realize that rule-based symbol manipulation is not the primary defining mark of intelligence. Even rats do computations that we're hard pressed to match with our computers. We've also come to realize that these are different classes of problems that require very different styles of computation. Brains and computers have very different strengths and weaknesses, which reflect these different computing styles.]

[Neural networks are partly inspired by the workings of actual brains. Let's take a look at a few things we know about biological neurons, and contrast them with both neural nets and traditional computation.]

- CPUs: largely sequential, nanosecond gates, fragile if gate fails
  superior for arithmetic, logical rules, perfect key-based memory
- Brains: very parallel, millisecond neurons, fault-tolerant

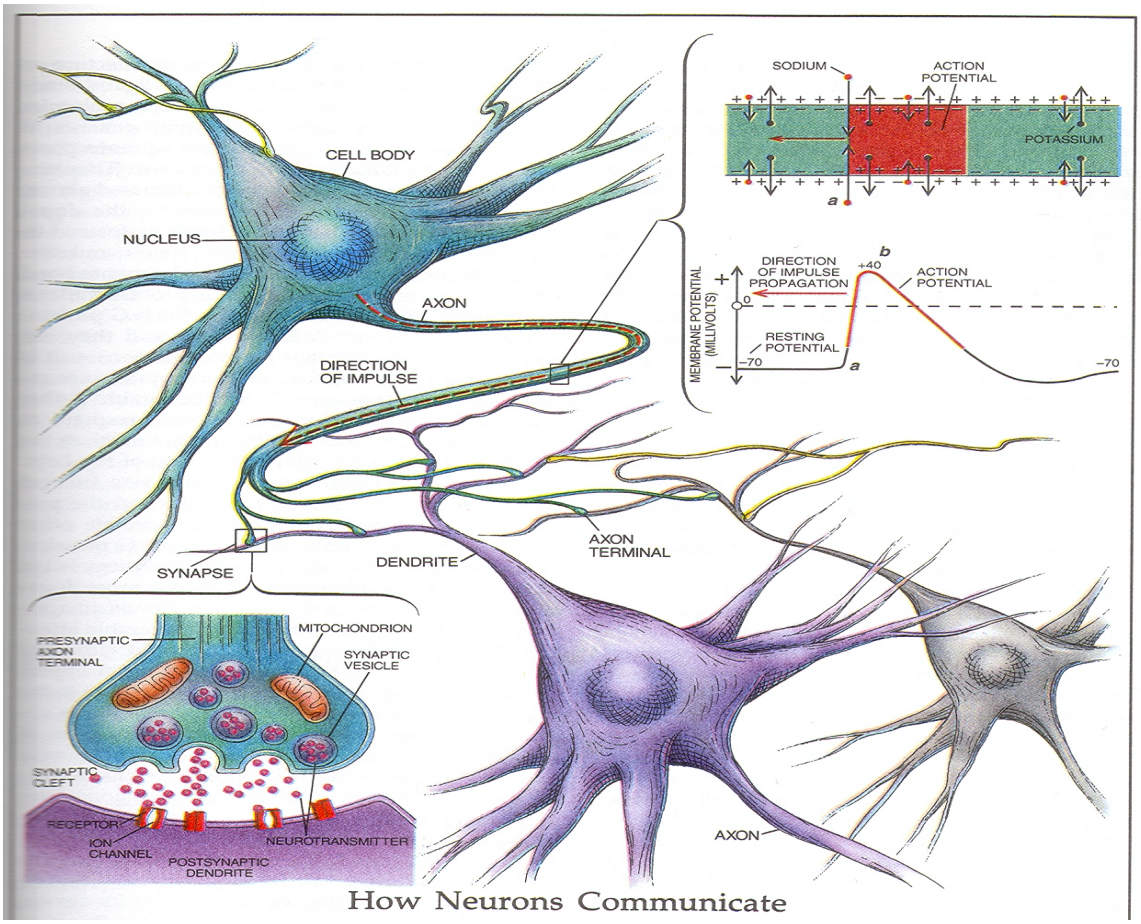
  [Neurons are continually dying. You've probably lost a few since this lecture started. But you probably didn't notice. And that's interesting, because it points out that our memories are stored in our brains in a diffuse representation. There is no one neuron whose death will make you forget that $2 + 2 = 4$. Artificial neural nets often share that resilience. Brains and neural nets seem to superpose memories on top of each other, all stored together in the same weights, sort of like a hologram.]

  superior for vision, speech, associative memory

  [By "associative memory," I mean noticing connections between things. One thing our brains are very good at is retrieving a pattern if we specify only a portion of the pattern.]

[It's impressive that even though a neuron needs a few milliseconds to transmit information to the next neurons downstream, we can perform very complex tasks like interpreting a visual scene in a tenth of a second. This is possible because neurons are parallel, but also because of their computation style.]

[Neural nets try to emulate the parallel, associative thinking style of brains, and they are among the best techniques we have for many fuzzy problems, including some problems in vision and speech. Not coincidentally, neural nets are also inferior at many traditional computer tasks such as multiplying numbers with lots of digits or compiling source code.]



neurons.pdf

- Neuron: A cell in brain/nervous system for thinking/communication
- Action potential or spike: An electrochemical impulse fired by a neuron to communicate w/other neurons
- Axon: The limb(s) along which the action potential propagates; "output"
  [Most axons branch out eventually, sometimes profusely near their ends.]
  [It turns out that giant squids have a very large axon they use for fast water jet propulsion. The mathematics of action potentials was first characterized in these giant squid axons, and that work won a Nobel Prize in Physiology in 1963.]
- Dendrite: Smaller limbs by which neuron receives info; "input"
- Synapse: Connection from one neuron's axon to another's dendrite
  [Some synapses connect axons to muscles or glands.]
- Neurotransmitter: Chemical released by axon terminal to stimulate dendrite

[When an action potential reaches an axon terminal, it causes tiny containers of neurotransmitter, called vesicles, to empty their contents into the space where the axon terminal meets another neuron's dendrite. That space is called the synaptic cleft. The neurotransmitters bind to receptors on the dendrite and influence the next neuron's body voltage. This sounds incredibly slow, but it all happens in 1 to 5 milliseconds.]

You have about 10^{11} neurons, each with about 10^4 synapses.
[Maybe 10^5 synapses after you pass CS 189.]

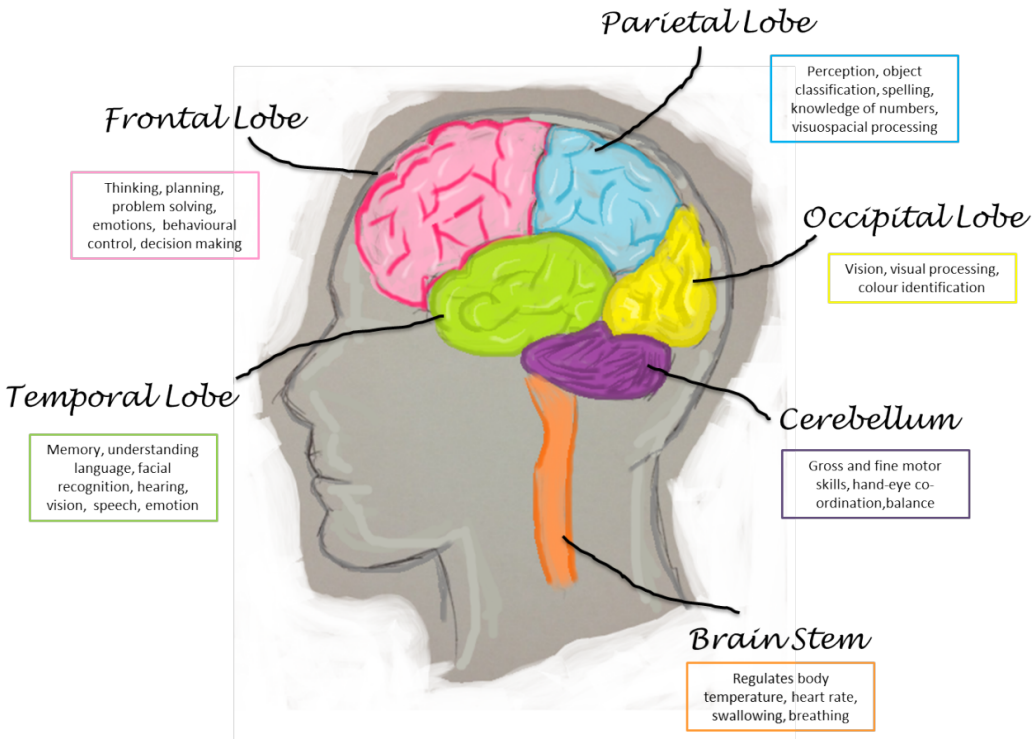
Analogies: [between artificial neural networks and brains]

- Output of unit ↔ firing rate of neuron
  [An action potential is "all or nothing"—all action potentials have the same shape and size. The output of a neuron is not signified by voltage like the output of a transistor. The output of a neuron is the frequency at which it fires. Some neurons can fire at nearly 1,000 times a second, which you might think of as a strong "1" output. Conversely, some types of neurons can go for minutes without firing. But some types of neurons never stop firing, and for those you might interpret a firing rate of 10 times per second as "0".]
- Weight of connection ↔ synapse strength
- Positive weight ↔ excitatory neurotransmitter (e.g. glutamine)
- Negative weight ↔ inhibitory neurotransmitter (e.g. GABA, glycine) [Gamma aminobutyric acid.]
  [A typical neuron is either excitatory at all its axon terminals, or inhibitory at all its terminals. It can't switch from one to the other. Artificial neural nets have an advantage here.]
- Linear combo of inputs ↔ summation
  [A neuron fires when the sum of its inputs, integrated over time, reaches a high enough voltage. However, the neuron body voltage also decays slowly with time, so if the action potentials are coming in slowly enough, the neuron might not fire at all.]
- Logistic/sigmoid fn ↔ firing rate saturation
  [A neuron can't fire more than 1,000 times a second, nor less than zero times a second. This limits its ability to be the sole determinant of whether downstream neurons fire. We accomplish the same thing with the sigmoid fn.]
- Weight change/learning ↔ synaptic plasticity
  Hebb's rule (1949): "Cells that fire together, wire together."
  [This doesn't mean that the cells have to fire at exactly the same time. But if one cell's firing tends to make another cell fire more often, their excitatory synaptic connection tends to grow stronger. There's a reverse rule for inhibitory connections. And there are ways for neurons that aren't even connected to grow connections.]
  [There are simple computer learning algorithms based on Hebb's rule. It can work, but it's generally not nearly as fast or effective as backpropagation.]

[Backpropagation is one part of artificial neural networks for which there is no analogy in the brain. Brains definitely do not do backpropagation.]

[Note to readers: Here I show Geoff Hinton's Hebbian learning slides, entitled "How to learn the weights" (hebbian.pdf). This part isn't very important, and I might skip it next time I teach the class, so if you don't have the slides, don't worry about it.]
[But this two-layer network is not flexible enough to do digit recognition well, especially when you have multiple writers with different handwriting. You can do much better with a three-layer network and backpropagation.]

[The brain is very modular.]



brain.png

[(The following items are all spoken, not written . . .)

- The part of our brain we think of as most characteristically human is the cerebral cortex, the seat of self-awareness, language, and abstract thinking.

But the brain has a lot of other parts that take the load off the cortex.

- Our brain stem regulates functions like heartbeat, breathing, and sleep.
- Our cerebellum governs fine coordination of motor skills. When we talk about "muscle memory," much of that is in the cerebellum, and it saves us from having to consciously think about how to walk or talk or brush our teeth, so the cortex can focus on where to walk and what to say and checking our phone.
- Our limbic system is the center of emotion and motivation, and as such, it makes a lot of the big decisions. I sometimes think that 90% of the job of our cerebral cortex is to rationalize decisions that have already been made by the limbic system. "Oh yeah, I made a careful, reasoned, logical decision to eat that fourth pint of ice cream."
- Our visual cortex performs a lot of processing on the input from your eyes to change it into a more useful form. Neuroscientists and computer scientists are particularly interested in the visual cortex for several reasons. Vision is an important problem for computers. The visual cortex is one of the easier parts of the brain to study in lab animals. The visual cortex is largely a feedforward network with few neurons going backward, so it's easier for us to train computers to behave like the visual cortex.]

[Although the brain has lots of specialized modules, one thing that's interesting about the cerebral cortex is that it seems to be made of general-purpose neural tissue that looks more or less the same everywhere, at least before it's trained. If you experience damage to part of the cortex early enough in life, while your brain is still growing, the functions will just relocate to a different part of the cortex, and you'll probably never notice the difference.]

[As computer scientists, our primary motivation for studying neurology is to try to get clues about how we can get computers to do tasks that humans are good at. But neurologists and psychologists have also been part of the study of neural nets from the very beginning. Their motivations are scientific: they're curious how humans think, and how we can do what we can do.]

NEURAL NET VARIATIONS

[I want to show you a few basic variations on the standard neural network I showed you last class, and how some of them change backpropagation.]

Regression: usually omit sigmoid fn from output unit(s).
[If you make that change, the gradient changes too, and you have to derive it for backprop. The gradient gets simpler, so I'll leave it as an exercise.]

Classification:

– Logistic loss fn (aka cross-entropy) often preferred to squared error.

$$L(z, y) = -\sum_i \Big(y_i \ln z_i + (1 - y_i) \ln(1 - z_i) \Big)$$

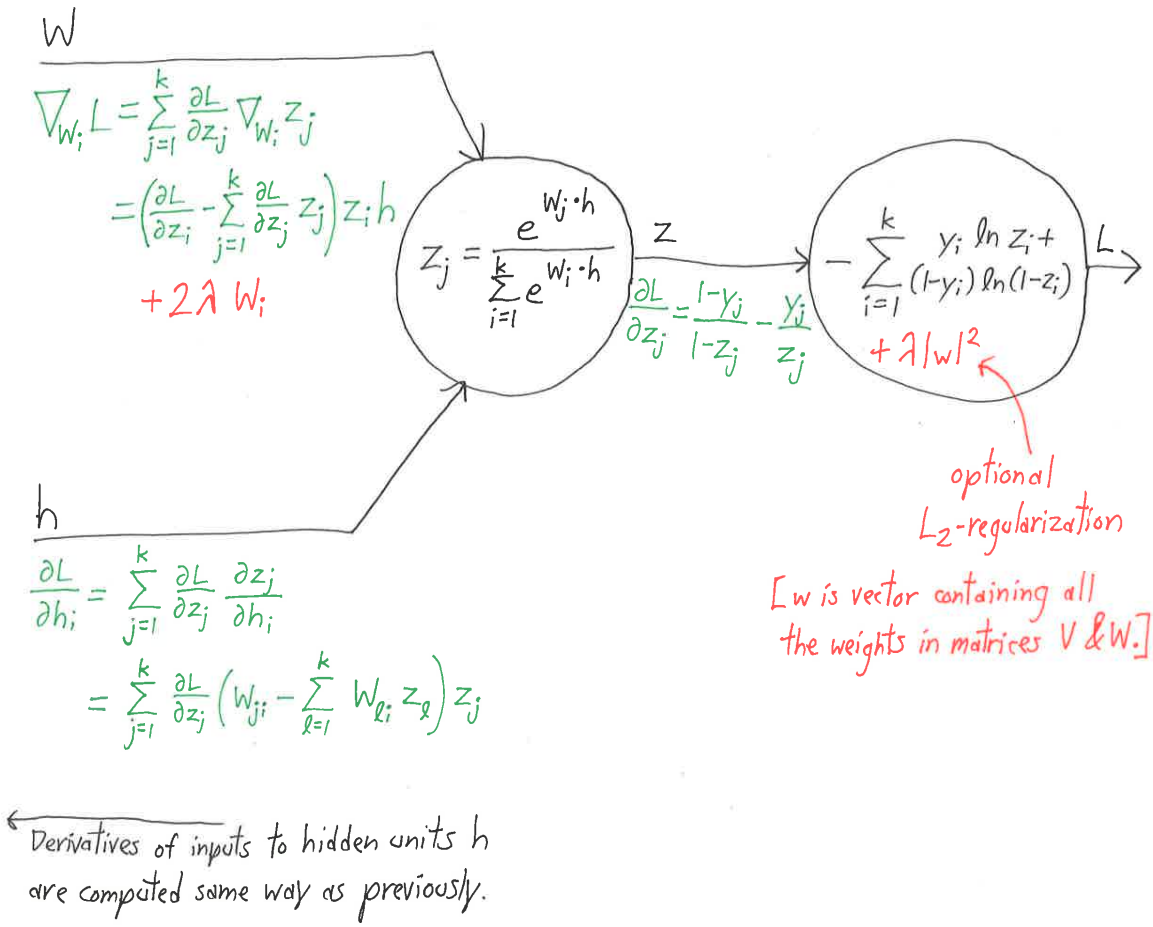
↑ true values
↑ prediction
} vectors

– For 2 classes, use one sigmoid output; for $k \geq 3$ classes, use softmax fn.
Let $t = Wh$ be k-vector of linear combos in final layer.

$$\text{Softmax output is } z_j(t) = \frac{e^{t_j}}{\sum_{i=1}^{k} e^{t_i}}. \qquad \frac{\partial z_j}{\partial t_j} = z_j (1 - z_j) \qquad \frac{\partial z_j}{\partial t_i} = -z_i z_j, i \neq j$$

[Each $z_j \in (0, 1)$, and their sum is 1.]

[Now I will show you how to derive the backprop equations for the softmax output, the logistic loss function, and L_2 regularization.]

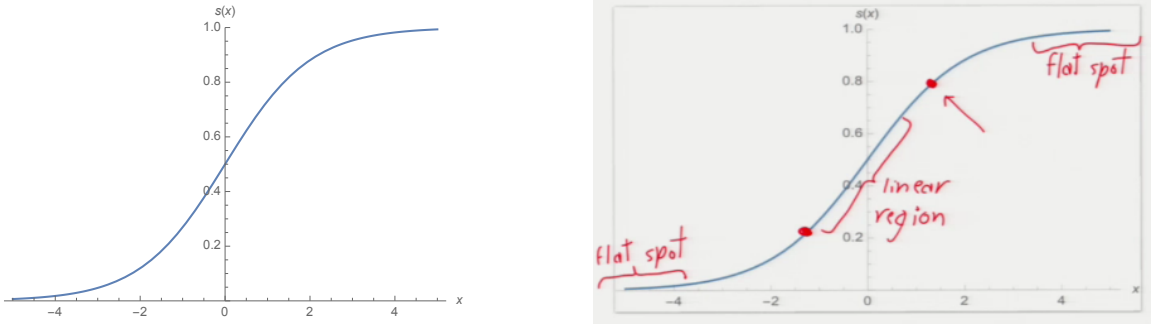


[Draw this by hand. gradsoftmax.pdf]

[Next I'm going to talk about a bunch of heuristics that make gradient descent faster, or make it find better local minima, or prevent it from overfitting. I suggest implementing vanilla stochastic backprop first, and experimenting with these other heuristics only after you get that working.]

Unit Saturation

Problem: When unit output s is close to 0 or 1 for all training points, $s' = s(1-s) \approx 0$, so gradient descent changes s very slowly. Unit is "stuck." Slow training & bad local minimum.



logistic.pdf [Draw flat spots, linear region, & max curvature points of the sigmoid function.]

[Wikipedia calls this the "vanishing gradient problem."]
[The more layers your network has, the more problematic this problem becomes. Most of the early attempts to train deep, many-layered neural nets failed.]

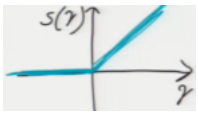
Mitigation: [None of these are complete cures.]

(1) Set target values to 0.15 & 0.85 instead of 0 & 1.
[Recall that the sigmoid function can never be 0 or 1; it can only come close. Relaxing the target values helps prevent the output units from getting saturated. The numbers 0.15 and 0.85 are reasonable because the sigmoid function achieves its greatest curvature when its output is near 0.21 or 0.79. But experiment to find the best values.]
[This helps to avoid stuck output units, but not stuck hidden units. So . . .]

(2) Modify backprop to add small constant (typically ~ 0.1) to s'.
[This hacks the gradient so a unit can't get stuck. We're not doing *steepest* descent any more, because we're not using the real gradient. But often we're finding a better descent direction that will get us to a minimum faster.]

(3) Initial weight of edge into unit with fan-in η:
random with mean zero, std. dev. $1/\sqrt{\eta}$.
[The bigger the fan-in of a unit, the easier it is to saturate it. So we choose smaller random initial weights for gates with bigger fan-in.]

(4) Replace sigmoid with ReLUs: rectified linear units.
ramp fn aka hinge fn: $s(\gamma) = \max\{0, \gamma\}$



$$s'(\gamma) = \begin{cases} 1 & \gamma \geq 0 \\ 0 & \gamma < 0 \end{cases}$$

[The derivative is not defined at zero, but we can just pretend it is.]
[Obviously, the gradient is sometimes zero, so you might wonder if ReLUs can get stuck too. Fortunately, it's rare for a ReLU's gradient to be zero for *all* the training data; it's usually zero for just some sample points. But yes, ReLUs sometimes get stuck too; just not as often as sigmoids.]

Popular for many-layer networks with large training sets.

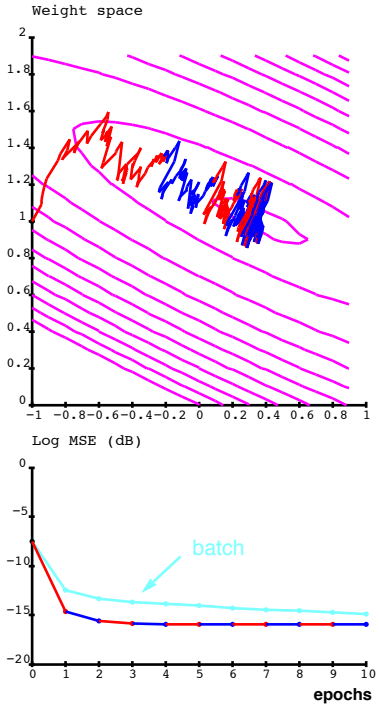
[One nice thing about ramp functions is that they and their gradients are very fast to compute. Computers compute exponentials slowly.]
[Even though ReLUs are linear in each half of their range, they're still nonlinear enough to easily compute functions like XOR.]

[Note that option (4) makes the first two options irrelevant.]

Heuristics for Avoiding Bad Local Minima

– (1) or (4) above.

– Stochastic gradient descent. A local minimum for batch descent is not a minimum for one typical training point.
[The idea is that instead of trying to optimize one risk function, we descend on one example's loss function and then we descend on another example's loss function. Every loss function has different local minima. It looks like a random walk or Brownian motion, and that random noise gets you out of shallow local minima.]



stochasticnnet.pdf (LeCun et al., "Efficient BackProp") [Stochastic gradient descent. Each red path and each blue path represents one epoch that presents every training point once.]

– Momentum. Gradient descent changes "velocity" slowly.
Carries us right through shallow local minima to deeper ones.

$\Delta w \leftarrow -\epsilon \nabla w$
repeat
    $w \leftarrow w + \Delta w$
    $\Delta w \leftarrow -\epsilon \nabla w + \beta \Delta w$

Good for both stochastic & batch descent. Choose hyperparameter $\beta < 1$.
[Think of Δw as the velocity. The hyperparameter β specifies how strongly momentum persists from iteration to iteration.]
[I've seen conflicting advice on β. Some researchers set it to 0.9; some set it close to zero.]
[If β is large, you should usually choose ϵ smaller to compensate, and you might also want to change the first line so the initial velocity is larger.]

19 More Variations on Neural Networks; Convolutional Neural Networks

Heuristics for Avoiding Bad Local Minima (continued)

- Train several nets; pick best.
  [If you train 10 neural nets on the same data, but each starting with different random weights, it's unlikely that any two of them will end up the same. Some might fall into bad local minima, but probably not all.]

- Layerwise pretraining.
  [We train one layer of the neural net at a time, starting from the input layer. The goal is to get the first layer to develop useful features, then for the second layer to develop even more useful features, and so on. This is a complicated topic on the forefront of research that I don't have time to do justice to. If you're interested, do a search.]

Heuristics for Faster Training

[One of the biggest disadvantages of neural nets is that they take a long, long time to train compared to most other classification methods we've studied. Here are some ideas for how to speed them up. Unfortunately, you often have to experiment with techniques and parameters for a while to figure out which ones will help with your particular application.]

– (1), (2), (3), (4) above.

– Stochastic gradient descent: faster than batch on large, redundant data sets.
[For example, if you have many different examples of the number "9", they contain some redundant information, and stochastic gradient descent learns the redundant information quickly.]

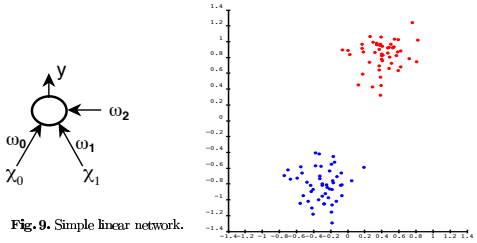
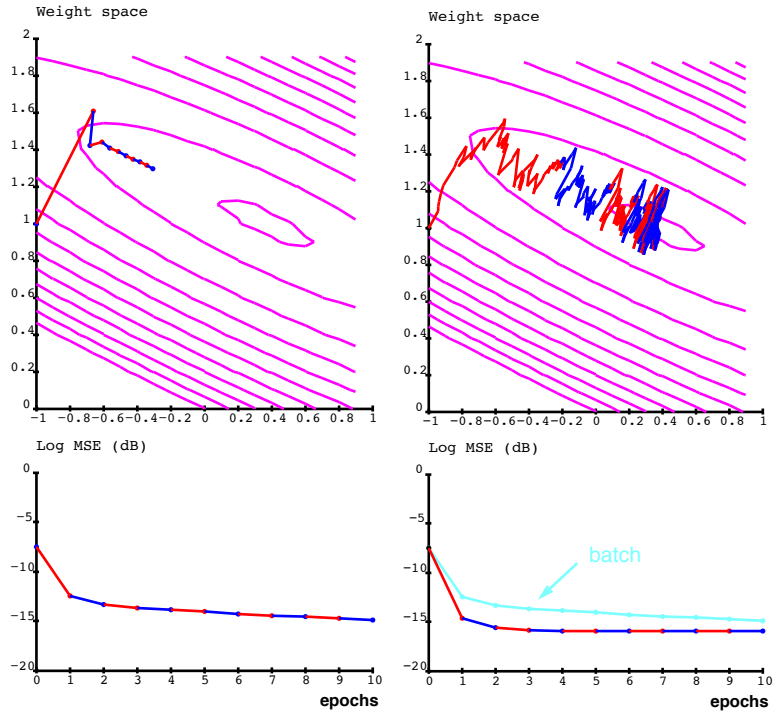


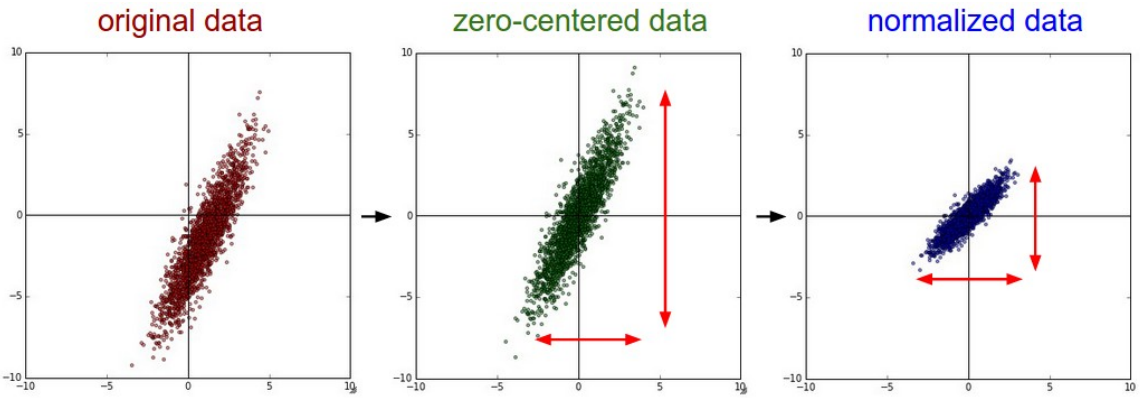
Fig. 9. Simple linear network.



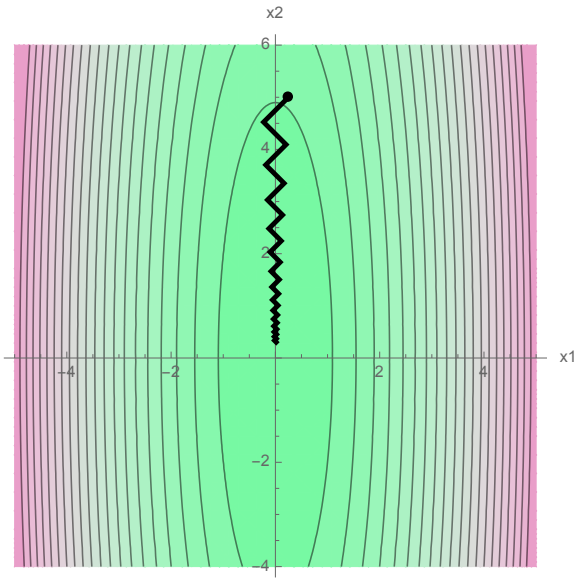
nnet2d.pdf, batchnnet.pdf, stochasticnnet.pdf (LeCun et al., "Efficient BackProp") [Top: a simple neural net with only three weights, and its 2D training data. Bottom left: batch gradient descent makes only a little progress each epoch. (Epochs alternate between red and blue.) Bottom right: stochastic descent decreases the error much faster than batch descent.]

One epoch presents every training example once. Training usually takes many epochs, but if data is huge it can take less than one.

– Normalizing the data:
  – First center each feature so mean is zero.
  – Then scale each feature so variance is constant (~ 1 is good).
    [The first step seems to make it easier for hidden units to get into a good operating region of the sigmoid. The second step makes the objective function better conditioned, so gradient descent converges faster.]



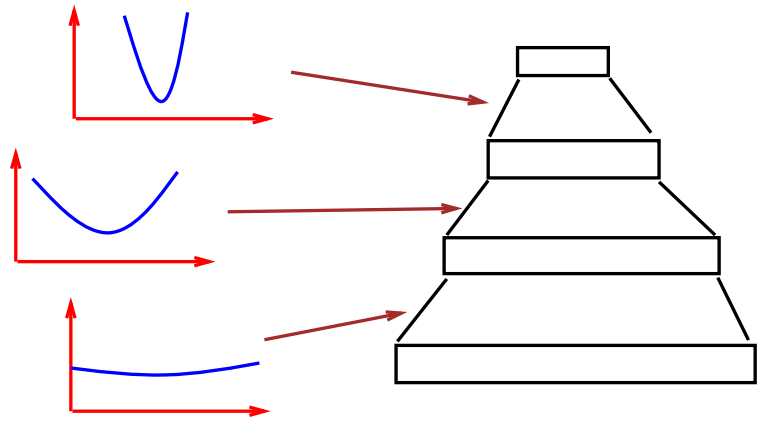
normalize.jpg [A 2D example of data normalization.]



illcondition.pdf [Skewed data often leads to an objective function with an ill-conditioned (highly eccentric) Hessian. Gradient descent in these functions can be painfully slow.]

– Centering the hidden units helps too.
  Replace sigmoids with $2\,s(\gamma) - 1$ or with $\tanh\gamma$.
  [These functions range from -1 to 1 instead of from 0 to 1.]
  [If you do this, don't forget that you also need to change s' in backprop. Also, good output target values change to roughly -0.7 and 0.7.]

– Use different learning rate for each layer of weights:
earlier layers have smaller gradients, need larger learning rate.



curvaturelayers.pdf [In this illustration, the inputs are at the bottom, and the outputs at the top. The derivatives tend to be smaller at the earlier layers.]

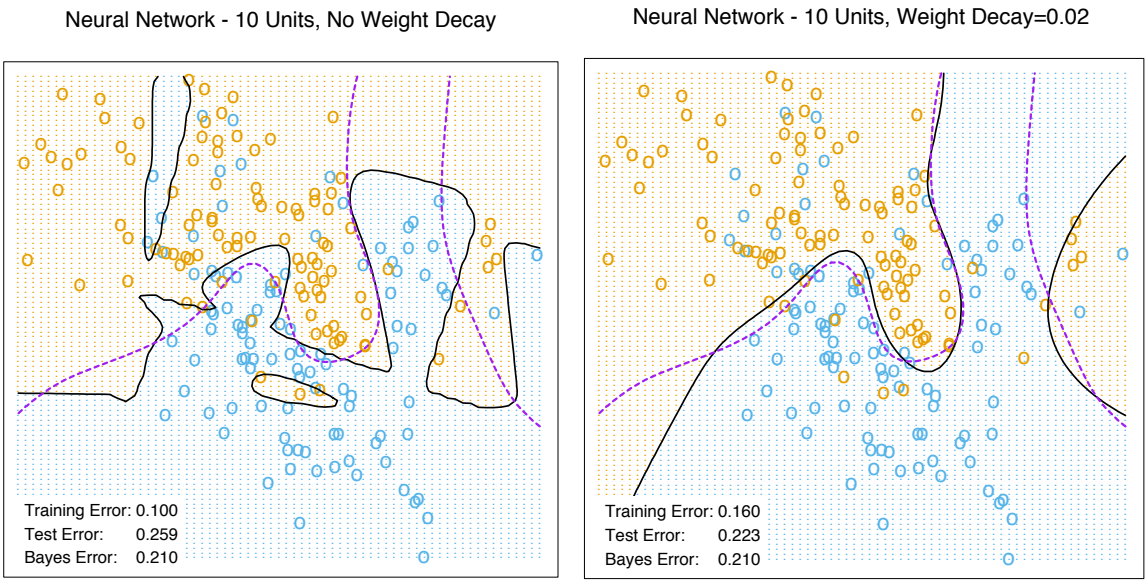
– Emphasizing schemes:
[Neural networks learn most quickly from the most unexpected examples. They learn most slowly from the most redundant examples. So we try to emphasize the uncommon examples.]
  – Shuffle so successive examples are never/rarely same class.
  – Present examples from rare classes more often, or w/bigger ϵ.
  – Warning: can backfire on bad outliers.

– Second-order optimization
  – No Newton's method; Hessian too expensive.
  – Nonlinear conjugate gradient; for small nets + small data + regression.
  Batch descent only! → Too slow with redundant data.
  – Stochastic Levenberg Marquardt; approximates a diagonal Hessian.
  [The authors claim convergence is typically three times faster than well-tuned stochastic gradient descent. The algorithm is complicated and I don't have time to cover it.]

Heuristics to Avoid Overfitting

– Ensemble of neural nets. Random initial weights; bagging.
[We saw how well ensemble learning works for decision trees. It works really well for neural nets too. The combination of random initial weights and bagging helps ensure that each neural net comes out different, even though they're trained on the same data. Obviously, this is slow.]

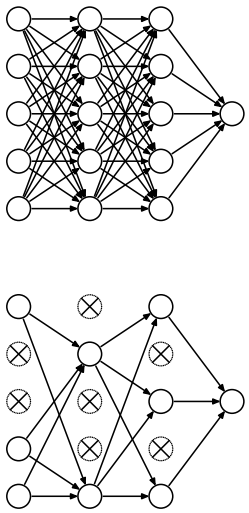
– L_2 regularization, aka weight decay.
Add $\lambda |w|^2$ to the cost/loss fn, where w is vector of all weights.
[Thus w includes all the weights in matrices V and W, rewritten as a vector.]
[We do this for the same reason we do it in ridge regression:
penalizing large weights reduces overfitting by reducing the variance.]
Effect: $-\frac{\partial J}{\partial w_i}$ has extra term $-2\lambda w_i$
Weight decays by factor $1 - 2\lambda\epsilon$ if not reinforced by training.



weightdecayoff.pdf, weightdecayon.pdf Write "softmax + logistic loss". [Examples of 2D classification without (left) and with (right) weight decay. Observe that the second example comes close to the Bayes optimal boundary.]

[One of the tricky parts of neural nets is deciding how many hidden units there should be. If there's too few, you can't learn very well, but if there's too many, they tend to overfit. L_2 regularization and weight decay make it safer to have too many hidden units, so it's less critical to find just the right number.]

– Dropout emulates an ensemble in one network. (Faster training too.)



dropout.pdf

[During training, we temporarily disable a random subset of the units, along with all the edges in and out of those units. It seems to work well to disable each hidden unit with probability 0.5, and to disable input units with a smaller probability. We do stochastic gradient descent and we frequently change which random subset of units is disabled. The authors claim that their method generalizes better than L_2 regularization. It gives some of the advantages of an ensemble, but it's faster to train.]

CONVOLUTIONAL NEURAL NETWORKS (ConvNets; CNNs)

[Convolutional neural nets have caused a big resurgence of interest in neural nets in the last few years. Often you'll hear the buzzword deep learning, which refers to neural nets with many layers.]

Vision: inputs are large images. 200×200 image = 40,000 pixels.
If we connect them all to 40,000 hidden units $\rightarrow$ 1.6 billion connections.
Neural nets are often overparametrized: too many weights, too little data.

[As a rule of thumb, you should have more data than you have weights to estimate. If you don't follow that rule, you usually overfit very badly. With images, it's impossible to get enough data to correctly train billions of weights. We could shrink the images, but then we're throwing away useful data. Another problem with having billions of weights is that the network becomes very slow to train or even to use.]

[Researchers have addressed these problems by taking inspiration from the neurology of the visual system. Remember that early in the semester, I told you that you can get better performance on the handwriting recognition task by using edge detectors. Edge detectors have two interesting properties. First, each edge detector looks at just one small part of the image. Second, the edge detection computation is the same no matter which part of the image you apply it to. So let's apply these two properties to neural net design.]

ConvNet ideas:

(1) Local connectivity: A hidden unit (in early layer) connects only to small patch of units in previous layer.

[This improves the overparametrization problem, and speeds up the network considerably.]

(2) Shared weights: Groups of hidden units share same set of input weights, called a mask aka filter aka kernel. We learn several masks.
[Each mask operates on every patch of image.]
Masks $\times$ patches = hidden units (in first hidden layer).
If one patch learns to detect edges, *every* patch has an edge detector.
[Because the mask that detects edges is applied to every patch.]
ConvNets automatically exploit repeated structure in images, audio.
Convolution: the same linear transformation applied to different parts of the input by shifting.

[Shared weights improve the overparametrization problem even more, because shared weights means fewer weights. It's a sort of regularization.]

[But shared weights have another big advantage. Suppose that gradient descent starts to develop an edge detector. That edge detector is being trained on *every* part of every image, not just on one spot. And that's good, because edges appear at different locations in different images. The location no longer matters; the edge detector can learn from edges in every part of the image.]

[In a neural net, you can think of hidden units as features that we learn, as opposed to features that you code up yourself. Convolutional neural nets take them to the next level by learning features from multiple patches simultaneously and then applying those features everywhere, not just in the patches where they were originally learned.]

[By the way, although local connectivity is inspired by our visual systems, shared weights obviously don't happen in biology.]

[Show slides on computing in the visual cortex and ConvNets (cnn.pdf), available from the CS 189 web page. Sorry, readers, there are too many images to include here. The narration is below.]

[Neurologists can stick needles into individual neurons in animal brains. After a few hours the neuron dies, but until then they can record its action potentials. In this way, biologists quickly learned how some of the neurons in the retina, called retinal ganglion cells, respond to light. They have interesting receptive fields, illustrated in the slides, which show that each ganglion cell receives excitatory stimulation from receptors in a small patch of the retina but inhibitory stimulation from other receptors around it.]

[The signals from these cells propagate to the V1 visual cortex in the occipital lobe at the back of your skull. The V1 cells proved much harder to understand. David Hubel and Torsten Wiesel of the Johns Hopkins University put probes into the V1 visual cortex of cats, but they had a very hard time getting any neurons to fire there. However, a lucky accident unlocked the secret and ultimately won them the 1981 Nobel Prize in Physiology.]

[Show video HubelWiesel.mp4, taken from YouTube: https://www.youtube.com/watch?v=IOHayh06LJ4]

[The glass slide happened to be at the particular orientation the neuron was sensitive to. The neuron doesn't respond to other orientations; just that one. So they were pretty lucky to catch that.]

[The simple cells act as line detectors and/or edge detectors by taking a linear combination of inputs from retinal ganglion cells.]
[The complex cells act as location-independent line detectors by taking inputs from many simple cells, which are location dependent.]

[Later researchers showed that local connectivity runs through the V1 cortex by projecting certain images onto the retina and using radioactive tracers in the cortex to mark which neurons had been firing. Those images show that the neural mapping from the retina to V1 is "retinatopic," i.e., local.]

[Unfortunately, as we go deeper into the visual system, layers V2 and V3 and so on, we know less and less about what processing the visual cortex does.]

[ConvNets were first popularized by the success of Yann LeCun's "LeNet 5" handwritten digit recognition software. LeNet 5 has six hidden layers! Layers 1 and 3 are convolutional layers in which groups of units share weights. Layers 2 and 4 are pooling layers that make the image smaller. These are just hardcoded max-functions with no weights and nothing to train. Layers 5 and 6 are just regular layers of hidden units with no shared weights. A great deal of experimentation went into figuring out the number of layers and their sizes. At its peak, LeNet 5 was responsible for reading the zip codes on 10% of US Mail. Another Yann LeCun system was deployed in ATMs and check reading machines and was reading 10 to 20% of all the checks in the US by the late 90's.]

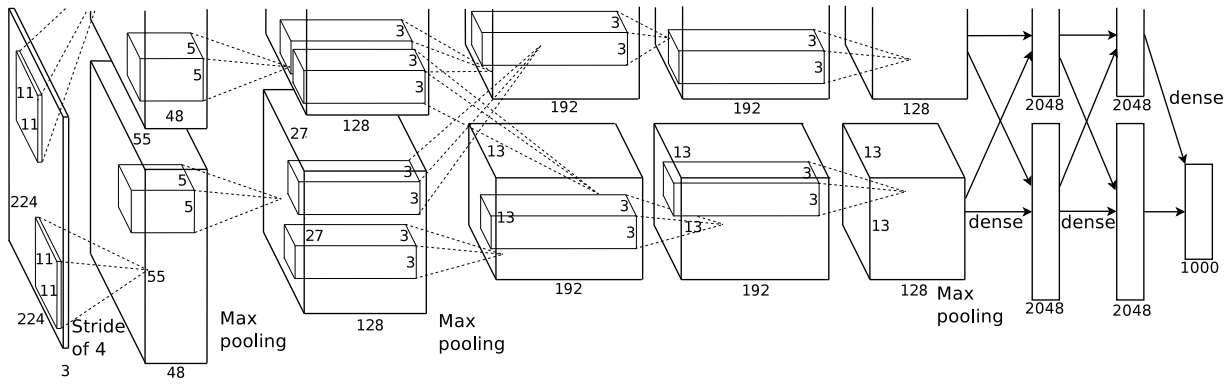
[Show Yann LeCun's video LeNet5.mov, illustrating LeNet 5.]

[When ConvNets were first applied to image analysis, researchers found that some of the learned masks are edge detectors or line detectors, similar to the ones that Hubel and Wiesel discovered! This created a lot of excitement in both the computer learning community and the neuroscience community. The fact that a neural net can naturally learn the same features as the mammalian visual cortex is impressive.]

[I told you last week that neural nets research was popular in the 60's, but the 1969 book *Perceptrons* killed interest in them throughout the 70's. They came back in the 80's, but interest was partly killed off a second time in the 00's by . . . guess what? By support vector machines. SVMs work well for a lot of tasks, they're faster to train, and they more or less have only one hyperparameter, whereas neural nets take a lot of work to tune.]

[Neural nets are now in their third wave of popularity. The single biggest factor in bringing them back is probably big data. Thanks to the internet, we now have absolutely huge collections of images to train neural nets with, and researchers have discovered that neural nets often give better performance than competing algorithms when you have huge amounts of data to train them with. In particular, convolutional neural nets are now learning better features than hand-tuned features. That's a recent change.]

[One event that brought attention back to neural nets was the ImageNet Image Classification Challenge in 2012.]
[Show ImageNet slide (imagenet.png).]
[The winner of that competition was a neural net, and it won by a huge margin, about 10%. It's called AlexNet, and it's surprisingly similarly to LeNet 5, in terms of how its layers are structured. However, there are some new innovations that led to their prize-winning performance, in addition to the fact that the training set had 1.4 million images: they used ReLUs, GPUs for training, and dropout.]



alexnet.pdf AlexNet.

20 Unsupervised Learning and Principal Components Analysis (PCA)

UNSUPERVISED LEARNING

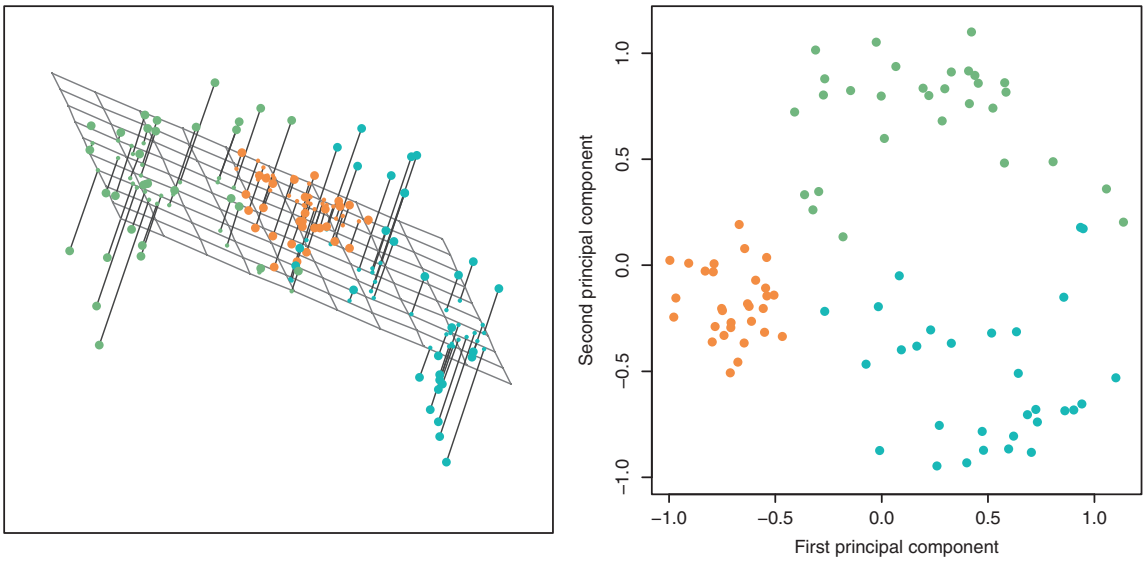
We have sample points, but no labels!
No classes, no y-values, nothing to predict.
Goal: Discover structure in the data.

Examples:

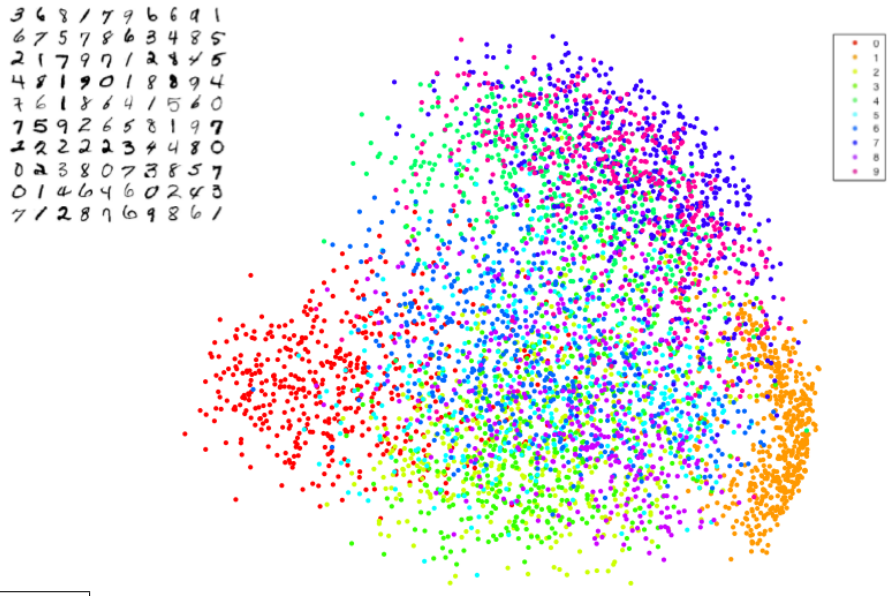
- Clustering: partition data into groups of similar/nearby points.
- Dimensionality reduction: data often lies near a low-dimensional subspace (or manifold) in feature space; matrices have low-rank approximations.
  [Whereas clustering is about grouping similar sample points, dimensionality reduction is more about identifying a continuous variation from sample point to sample point.]
- Density estimation: fit a continuous distribution to discrete data.
  [When we use maximum likelihood estimation to fit Gaussians to sample points, that's density estimation, but we can also fit functions more complicated than Gaussians, with more local variation.]

PRINCIPAL COMPONENTS ANALYSIS (PCA) (Karl Pearson, 1901)

Goal: Given sample points in $\mathbb{R}^d$, find k directions that capture most of the variation. (Dimensionality reduction.)



3dpca.pdf [Example of 3D points projected to 2D by PCA.]



pcadigits.pdf [The (high-dimensional) MNIST digits projected to 2D. Two dimensions aren't enough to fully separate the digits, but observe that the digits 0 (red) and 1 (orange) are well on their way to being separated.]

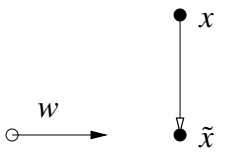
Why?

- Find a small basis for representing variations in complex things, e.g. faces, genes.
- Reducing # of dimensions makes some computations cheaper, e.g. regression.
- Remove irrelevant dimensions to reduce overfitting in learning algs.
  Like subset selection, but we can choose features that aren't axis-aligned,
  i.e. linear combos of input features.

[Sometimes PCA is used as a preprocess before regression or classification for the last two reasons.]

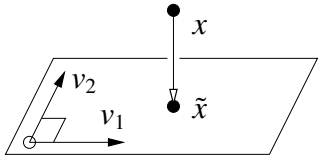
Let X be $n \times d$ design matrix. [No fictitious dimension.]
From now on, assume X is centered: mean X_i is zero.
[As usual, we can center the data by computing the mean x-value, then subtracting the mean from each sample point.]

[Let's start by seeing what happens if we pick just one principal direction.]
Let w be a unit vector.
The orthogonal projection of point x onto vector w is $\tilde{x} = (x \cdot w)\, w$
If w not unit, $\tilde{x} = \frac{x \cdot w}{|w|^2} w$



[The idea is that we're going to pick the best direction w, then project all the data down onto w so we can analyze it in a one-dimensional space. Of course, we lose a lot of information when we project down from d dimensions to just one. So, suppose we pick several directions. Those directions span a subspace, and we want to project points orthogonally onto the subspace. This is easy *if* the directions are orthogonal to each other.]

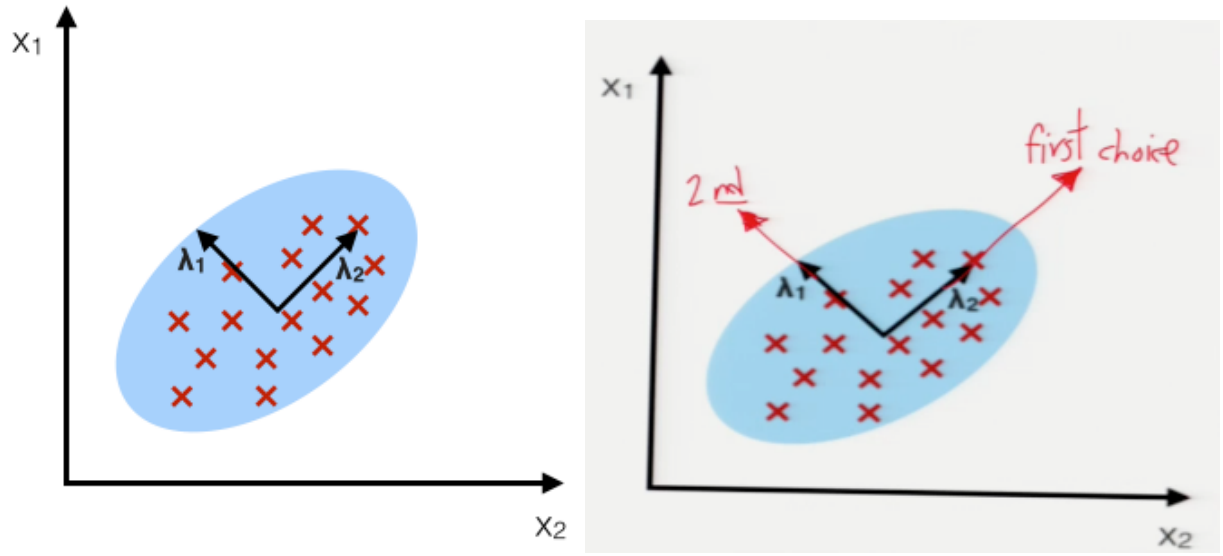
Given orthonormal directions $v_1, \ldots, v_k$, $\tilde{x} = \sum_{i=1}^{k} (x \cdot v_i)\, v_i$
[The word "orthonormal" implies they're mutually orthogonal and length 1.]



[Usually we don't actually want the projected point in $\mathbb{R}^d$;
usually we want the coordinates $x \cdot v_i$ in principal components space.]

$X^\top X$ is square, symmetric, positive semidefinite, $d \times d$ matrix.
Let $0 \le \lambda_1 \le \lambda_2 \le \ldots \le \lambda_d$ be its eigenvalues. [sorted]
Let $v_1, v_2, \ldots, v_d$ be corresponding orthogonal **unit** eigenvectors.
[It turns out that the principal directions will be these eigenvectors, and the most important ones will be the ones with the greatest eigenvalues. I will show you this in three different ways.]

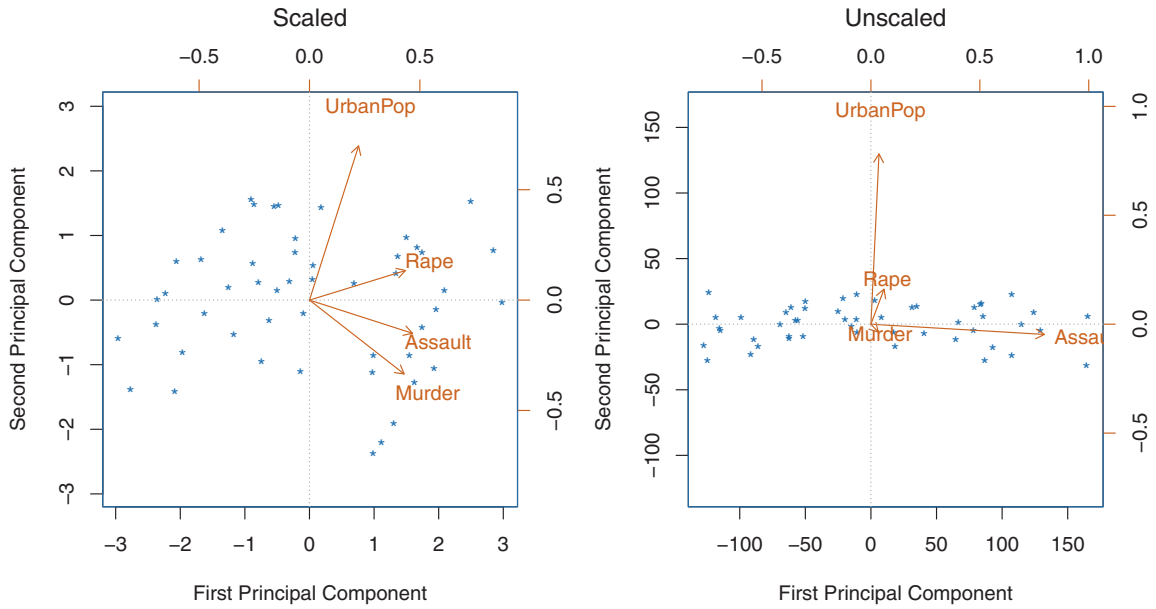
PCA derivation 1: Fit a Gaussian to data with maximum likelihood estimation.
Choose k Gaussian axes of greatest variance.



gaussfitpca.png [A Gaussian fitted to sample points.]

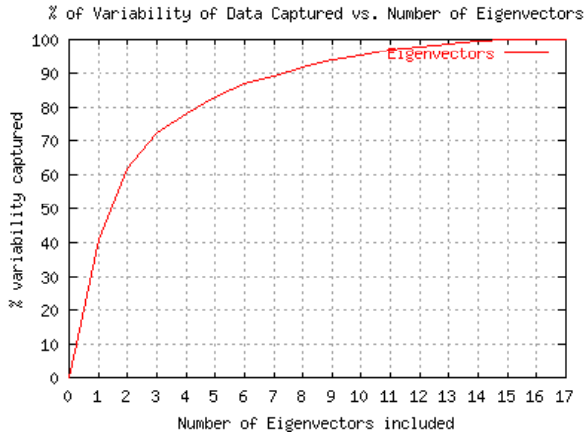
Recall that MLE estimates a covariance matrix $\hat{\Sigma} = \frac{1}{n} X^\top X$. [Presuming X is centered.]

PCA Alg:
- Center X.
- Optional: Normalize X. Units of measurement different?
  - Yes: Normalize.
    [Bad for principal components to depend on arbitrary choice of scaling.]
  - No: Usually don't.
    [If several features have the same unit of measurement, but some of them have smaller variance than others, that difference is usually meaningful.]



normalize.pdf [Normalized data at left; unnormalized data at right. The arrows show the original axes projected on the top two principal components. When the data are not normalized, rare occurrences like murder have little influence on the principal directions. Which is better? It depends on whether you think that low-frequency events like murder and rape should have a disproportionate influence.]

– Compute unit eigenvectors/values of $X^\top X$.
– Optional: choose k based on the eigenvalue sizes.
– For the best k-dimensional subspace, pick eigenvectors $v_{d-k+1}, \ldots, v_d$.
– Compute the coordinates $x \cdot v_i$ of training/test data in principal components space.
[When we do this projection, we have two choices: we can un-center the input training data before projecting it, OR we can translate the test data by the same vector we used to translate the training data when we centered it.]

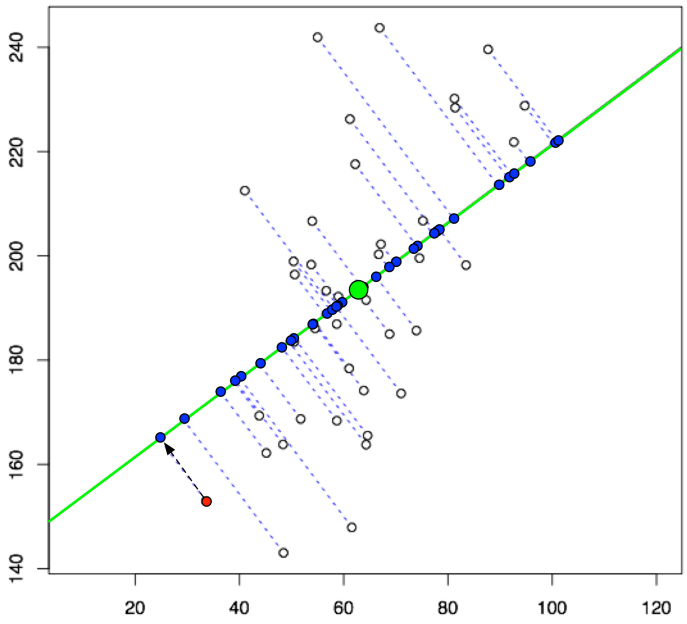


$$r_k = \sum_{i=1}^{k} \lambda_i^2 \Big/ \sum_{i=1}^{n} \lambda_i^2$$

variance.pdf [Plot of # of eigenvectors vs. percentage of variance captured . In this example, just 3 eigenvectors capture 70% of the variance.]

[If you are using PCA as a preprocess for a supervised learning algorithm, there's a more effective way to choose k: (cross-)validation.]

PCA derivation 2: Find direction w that maximizes variance of projected data
[In other words, when we project the data down, we don't want it all to bunch up; we want to keep it as spread out as possible.]

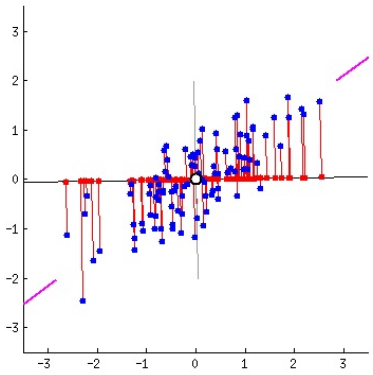


project.jpg [Points projected on a line. We wish to choose the orientation of the green line to maximize the variance of the blue points.]

$$\text{Maximize } \mathrm{Var}(\{\tilde{X}_1, \tilde{X}_2, \ldots, \tilde{X}_n\}) = \frac{1}{n}\sum_{i=1}^{n}\left(X_i \cdot \frac{w}{|w|}\right)^2 = \frac{1}{n}\frac{|Xw|^2}{|w|^2} = \frac{1}{n}\underbrace{\frac{w^\top X^\top X w}{w^\top w}}_{\text{Rayleigh quotient of } X^\top X \text{ and } w}$$

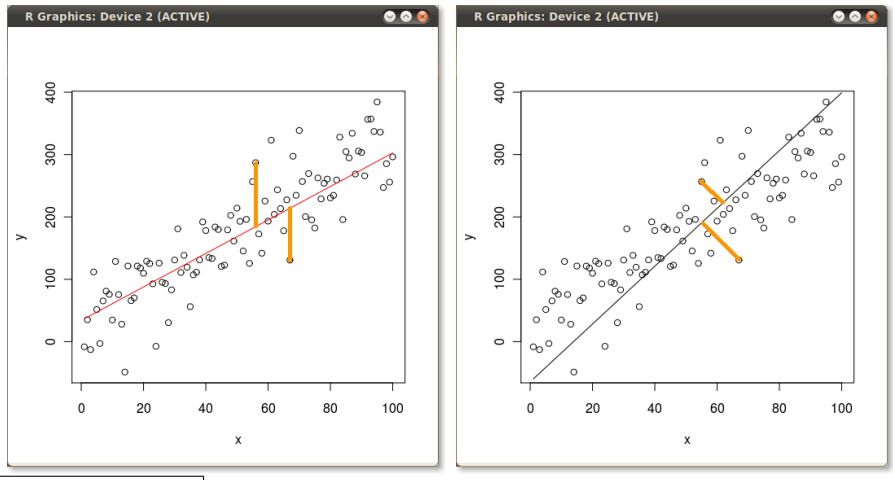
[This fraction is a well-known construction called the Rayleigh quotient. When you see it, you should smell eigenvectors nearby. How do we maximize this?]
If w is an eigenvector v_i, Ray. quo. $= \lambda_i$
$\rightarrow$ of all eigenvectors, v_d achieves maximum variance λ_d/n.
One can show v_d beats every other vector too.
[Because every vector w is a linear combination of eigenvectors, and so its Rayleigh quotient will be a convex combination of eigenvalues. It's easy to prove this, but I don't want to take the time. For the proof, look up "Rayleigh quotient" in Wikipedia.]
[So the top eigenvector gives us the best direction. But we typically want k directions. After we've picked one direction, then we have to pick a direction that's orthogonal to the best direction. But subject to that constraint, we again pick the direction that maximizes the variance.]
What if we constrain w to be orthogonal to v_d? Then pick v_{d-1}.

PCA derivation 3: Find direction w that minimizes "projection error"



PCAanimation.gif [This is an animated GIF; unfortunately, the animation can't be included in the PDF lecture notes. Find the direction of the black line for which the sum of squares of the lengths of the red lines is smallest.]

[You can think of this as a sort of least-squares linear regression, with one important change. Instead of measuring the error in a fixed vertical direction, we're measuring the error in a direction orthogonal to the principal component direction we choose.]

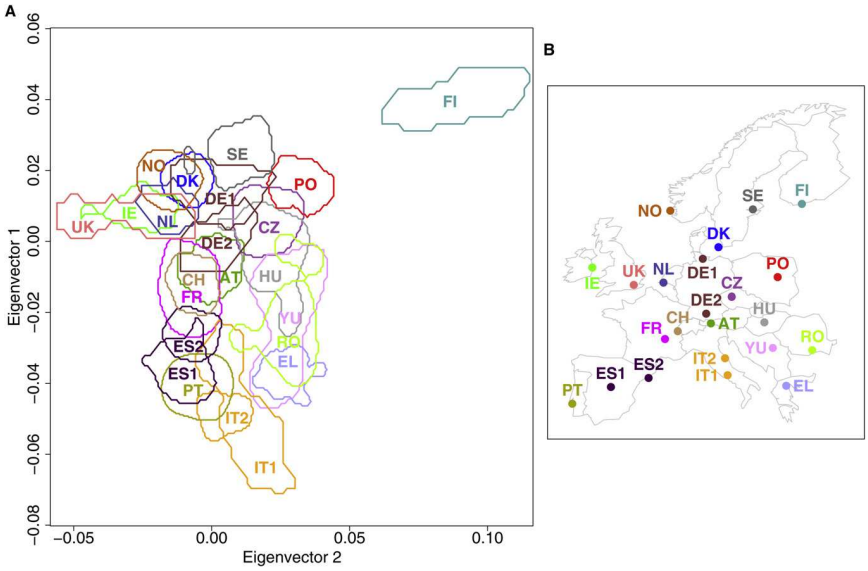


mylsq.png, mypca.png [Least-squares linear regression vs. PCA. In linear regression, the projection direction is always vertical; whereas in PCA, the projection direction is orthogonal to the projection hyperplane. In both methods, however, we minimize the sum of the squares of the projection distances.]

$$\text{Minimize} \sum_{i=1}^{n} \left|X_i - \tilde{X}_i\right|^2 = \sum_{i=1}^{n} \left|X_i - \frac{X_i \cdot w}{|w|^2} w\right|^2 = \sum_{i=1}^{n} \left(|X_i|^2 - \left(X_i \cdot \frac{w}{|w|}\right)^2\right)$$
$$= \text{constant} - n\,(\text{variance from derivation 2}).$$

Minimizing projection error = maximizing variance.

[From this point, we carry on with the same reasoning as derivation 2.]



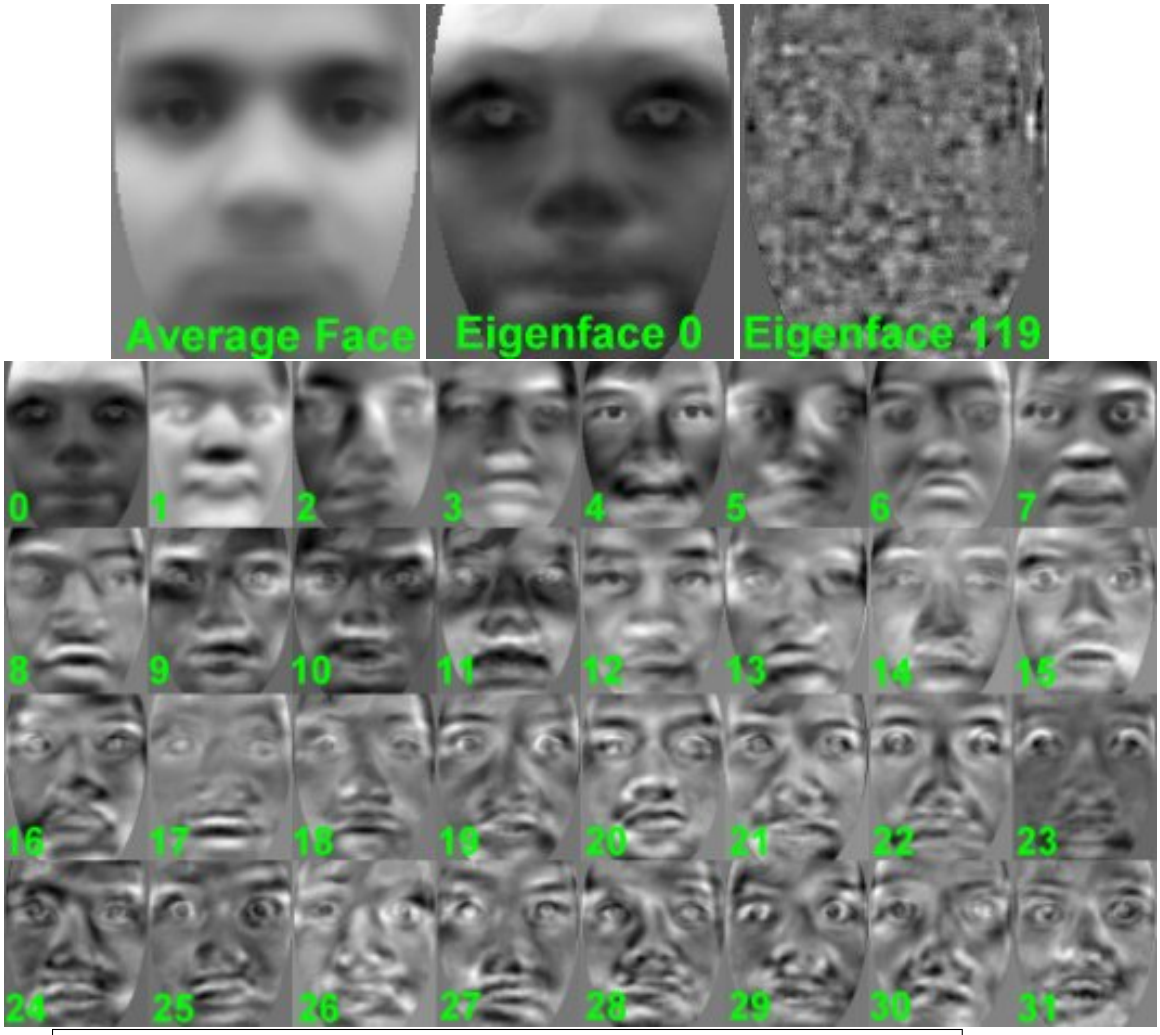
europegenetics.pdf [Illustration of the first two principal components of the single nucleotide polymorphism (SNP) matrix for the genes of various Europeans. The input matrix has 2,541 people from these locations in Europe, and 309,790 SNPs. Each SNP is binary, so think of it as 309,790 dimensions of zero or one. The output shows spots on the first two principal components where the projected people from a particular national type are denser than a certain threshold. What's amazing about this is how closely the projected genotypes resemble the geography of Europe. (From Lao et al., Current Biology, 2008.)]

Eigenfaces

X contains n images of faces, d pixels each.
[If we have a 200 × 200 image of a face, we represent it as a vector of length 40,000, the same way we represent the MNIST digit data.]
Face recognition: Given a query face, compare it to all training faces; find nearest neighbor in $\mathbb{R}^d$.
[This works best if you have several training photos of each person you want to recognize, with different lighting and different facial expressions.]

Problem: Each query takes $\Theta(nd)$ time.

Solution: Run PCA on faces. Reduce to much smaller dimension d'.
Now nearest neighbor takes $O(nd')$ time.
[Possibly even less. We'll talk about speeding up nearest-neighbor search at the end of the semester. If the dimension is small enough, you can sometimes do better than linear time.]



facerecaverage.jpg, facereceigen0.jpg, facereceigen119.jpg, facereceigen.jpg [Images of the average face and the eigenfaces.]

eigenfaceproject.pdf [Images of a face (left) projected onto the first 4 and 50 eigenvectors. The latter is blurry but good enough for face recognition.]

For best results, equalize the intensity distributions first.

facerecequalize.jpg [Image equalization.]

[If you have 500 stored faces with 40,000 pixels each, and you reduce them to 40 principal components, then each query face requires you to read 20,000 stored coordinates instead of 20 million pixels.]

[Eigenfaces are not perfect. They encode both face shape *and* lighting. Ideally, we would have some way to factor out lighting and analyze face shape only, but that's harder. Some people say that the first 3 eigenfaces are usually all about lighting, and you sometimes get better facial recognition by dropping the first 3 eigenfaces.]

[Show Blanz–Vetter face morphing video (morphmod.mpg).]

[Blanz and Vetter use PCA in a more sophisticated way for 3D face modeling. They take 3D scans of people's faces and find correspondences between peoples' faces and an idealized model. For instance, they identify the tip of your nose, the corners of your mouth, and other facial features, which is something the original eigenface work did not do. Instead of feeding an array of pixels into PCA, they feed the 3D locations of various points on your face into PCA. This works more reliably.]

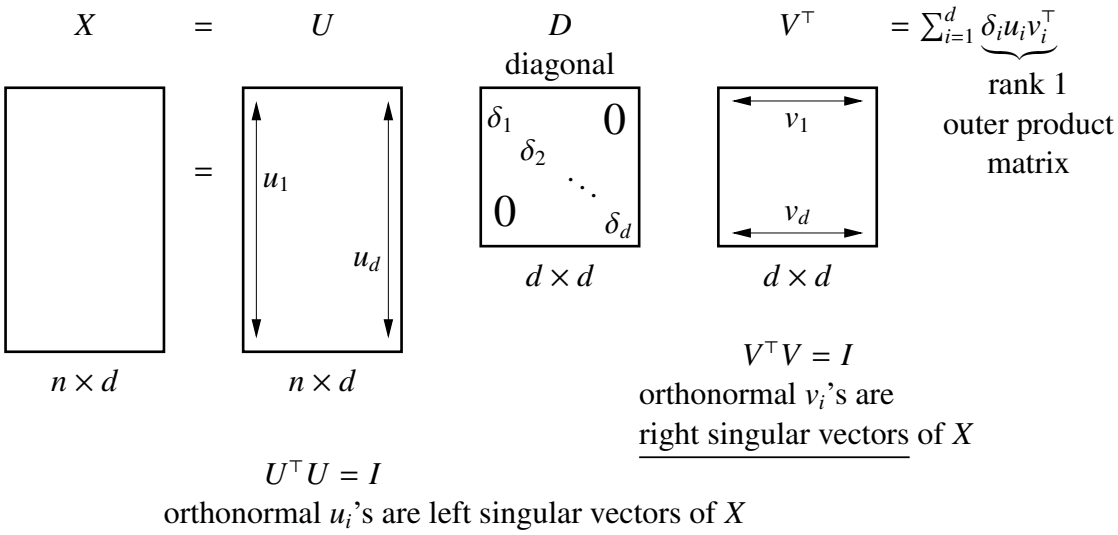
21 The Singular Value Decomposition; Clustering

The Singular Value Decomposition (SVD) [and its application to PCA]

Problems: Computing $X^\top X$ takes $\Theta(nd^2)$ time.
$X^\top X$ is poorly conditioned → numerically inaccurate eigenvectors.
[The SVD improves both these problems.]

[Earlier this semester, we learned about the eigendecomposition of a square, symmetric matrix. Unfortunately, nonsymmetric matrices don't decompose nearly as nicely, and non-square matrices don't have eigenvectors at all. Happily, there is a similar decomposition that works for all matrices, even if they're not symmetric and not square.]

Fact: If $n \geq d$, we can find a singular value decomposition $X = UDV^\top$



[Draw this by hand; write summation at right last. svd.pdf]

Diagonal entries $\delta_1, \ldots, \delta_d$ of D are nonnegative singular values of X.

[Some of the singular values might be zero. The number of nonzero singular values is equal to the rank of the centered design matrix X. If all the sample points lie on a line, there is only one nonzero singular value. If the points span a subspace of dimension r, there are r nonzero singular values.]

[If $n < d$, an SVD still exists, but now U is square and V is not.]

Fact: v_i is an eigenvector of $X^\top X$ w/eigenvalue δ_i^2.
Proof: $X^\top X = VDU^\top UDV^\top = VD^2V^\top$
which is an eigendecomposition of $X^\top X$.

[The columns of V are the eigenvectors of $X^\top X$, which is what we need for PCA. If $n < d$, V will omit some of the eigenvectors that have eigenvalue zero, but those are useless for PCA. The SVD also tells us the eigenvalues, which are the squares of the singular values. By the way, that's related to why the SVD is more numerically stable: because the ratios between singular values are smaller than the ratios between eigenvalues.]

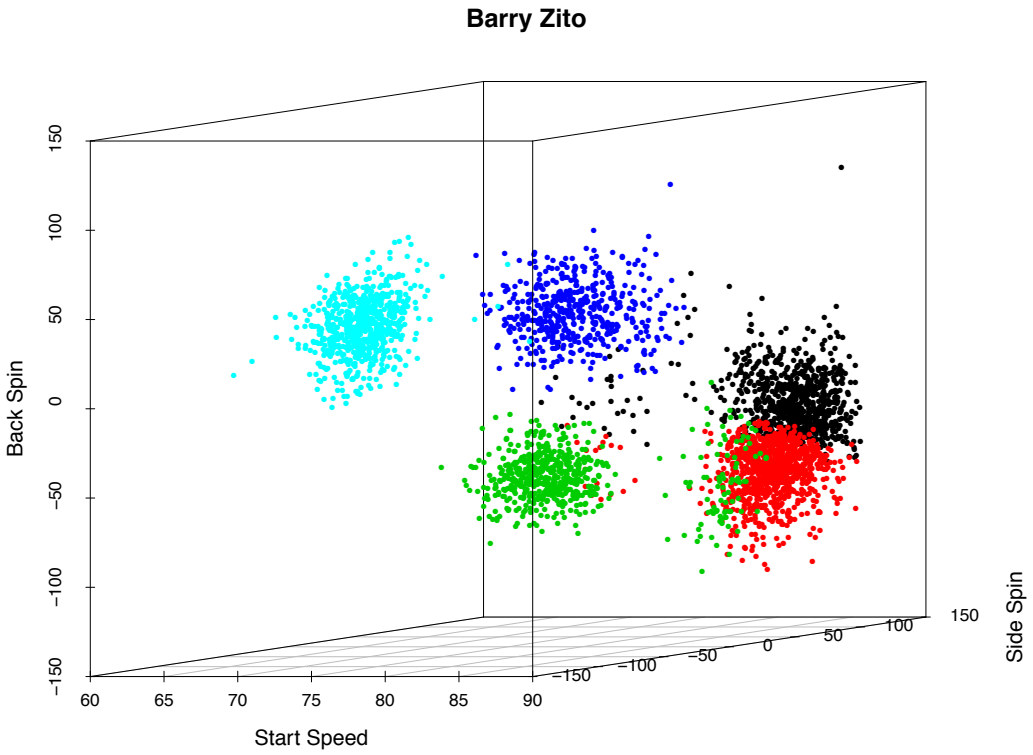
Fact: We can find the k greatest singular values & corresponding vectors in $O(ndk)$ time.
[So we can save time by computing some of the singular vectors without computing all of them.]
[There are approximate, randomized algorithms that are even faster, producing an approximate SVD in $O(nd \log k)$ time. These are starting to become popular in algorithms for very big data.]
[https://code.google.com/archive/p/redsvd/]

Row i of UD gives the coordinates of sample point X_i in principal components space (i.e. $X_i \cdot v_j$ for each j). [So we don't need to project the input points onto principal components space; the SVD has already done it for us.]

CLUSTERING

Partition data into clusters so points within a cluster are more similar than across clusters.
Why?

- Discovery: Find songs similar to songs you like; determine market segments
- Hierarchy: Find good taxonomy of species from genes
- Quantization: Compress a data set by reducing choices
- Graph partitioning: Image segmentation; find groups in social networks



4-Seam Fastball	2-Seam Fastball	Changeup	Slider	Curveball
Black	Red	Green	Blue	Light Blue

zito.pdf [Clusters that classify Barry Zito's baseball pitches. Here we discover that there really are distinct classes of baseball pitches.]

k-Means Clustering aka Lloyd's Algorithm (Stuart Lloyd, 1957)

Goal: Partition n points into k disjoint clusters.
Assign each input point X_i a cluster label $y_i \in [1, k]$.
Cluster i's mean is $\mu_i = \frac{1}{n_i} \sum_{y_j=i} X_j$, given n_i points in cluster i.

Find y that minimizes $\sum_{i=1}^{k} \sum_{y_j=i} |X_j - \mu_i|^2$ [Sum of the squared distances from points to their cluster means.]

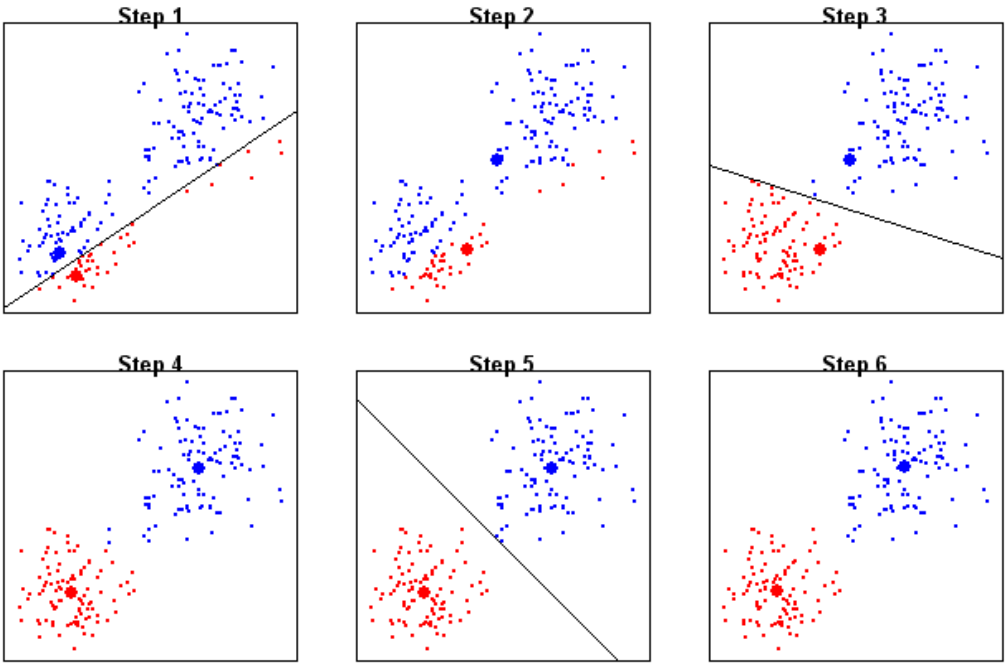
NP-hard. Solvable in $O(nk^n)$ time. [Try every partition.]

k-means heuristic: Alternate between
(1) y_j's are fixed; update μ_i's
(2) μ_i's are fixed; update y_j's
Halt when step (2) changes no assignments.

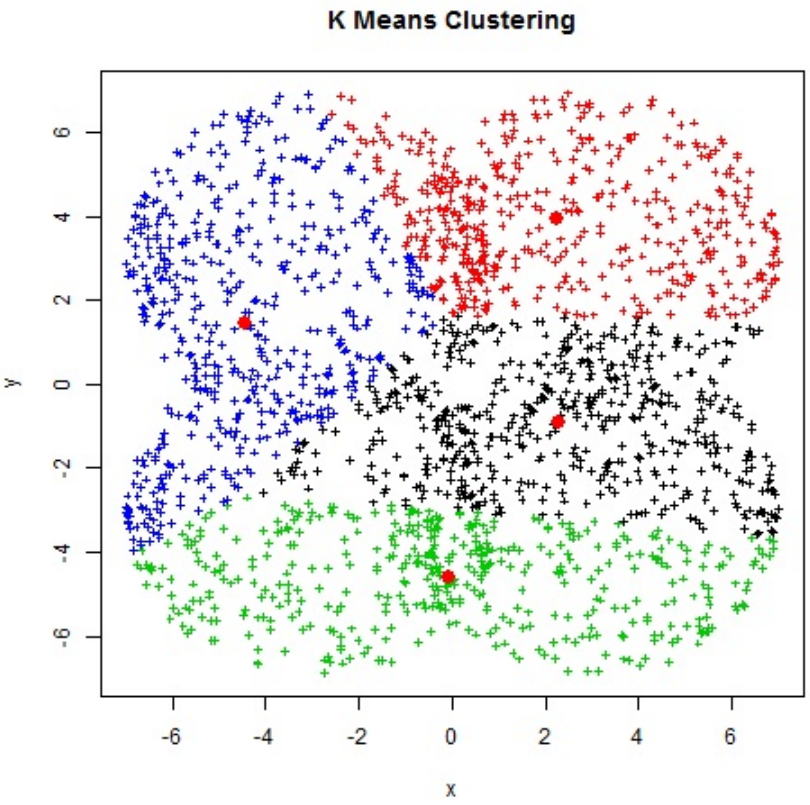
[So, we have an assignment of points to clusters. We compute the cluster means. Then we reconsider the assignment. A point might change clusters if some other's cluster's mean is closer than its own cluster's mean. Then repeat.]

Step (1): One can show (calculus) the optimal μ_i is the mean of the points in cluster i.
[This is easy calculus, so I leave it as a short exercise.]

Step (2): The optimal y assigns each point X_j to the closest center μ_i.
[This should be even more obvious than step (1).]
[If there's a tie, and one of the choices is for X_j to stay in the same cluster as the previous iteration, always take that choice.]

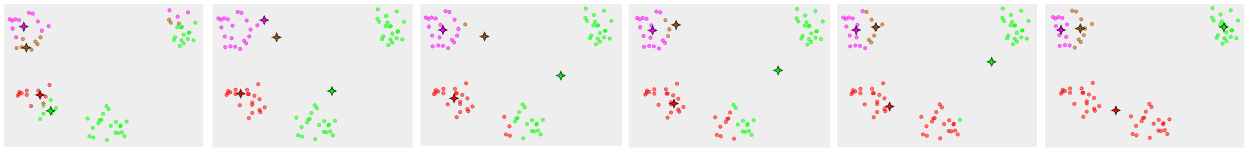


2means.png [An example of 2-means. Odd-numbered steps reassign the data points. Even-numbered steps compute new means.]



4meansanimation.gif [This is an animated GIF of 4-means with many points. Unfortunately, the animation can't be included in the PDF lecture notes.]

Both steps decrease objective fn *unless* they change nothing.
[Therefore, the algorithm never returns to a previous assignment.]
Hence alg. must terminate. [As there are only finitely many assignments.]
[This argument doesn't say anything optimistic about the running time, because we might see $O(k^n)$ different assignments before we halt. In theory, one can actually construct point sets in the plane that take an exponential number of iterations, but those never come up in practice.]
Usually very fast in practice. Finds a local minimum, often not global.
[. . . which is not surprising, as this problem is NP-hard.]



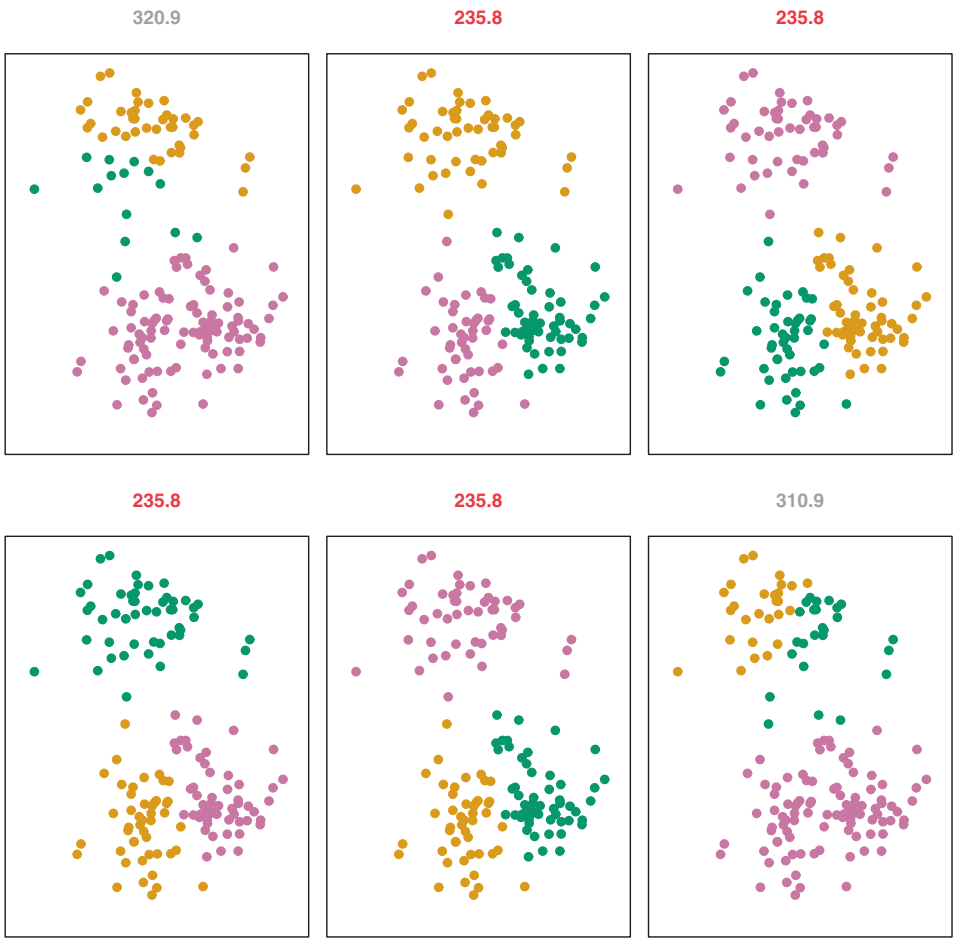
4meansbad.png [An example where 4-means clustering fails.]

Getting started:

- Forgy method: choose k random sample points to be initial μ_i's; go to (2).
- Random partition: randomly assign each sample point to a cluster; go to (1).

[Forgy seems to be better, but Wikipedia mentions some variants of k-means for which random partition is better.]

For best results, run k-means multiple times with random starts.



kmeans6times.pdf (ISL, Figure 10.7) [Clusters found by running 3-means 6 times on the same sample points.]

[Why did we choose that particular objective function to minimize? Partly because it is equivalent to minimizing the following function.]

Equivalent objective fn: the within-cluster variation

Find y that minimizes $\sum_{i=1}^{k} \frac{1}{n_i} \sum_{y_j=i} \sum_{y_m=i} |X_j - X_m|^2$

[This objective function is equal to twice the previous one. It's a worthwhile exercise to show that—it's harder than it looks. The nice thing about this expression is that it doesn't include the means; it's a direct function of the input points and the clusters we assign them to.]

Normalize the data? [before applying k-means]
Same advice as for PCA. Sometimes yes, sometimes no.
[If some features are much larger than others, they will tend to dominate the Euclidean distance. So if you have features in different units of measurement, you probably should normalize them. If you have features in the same unit of measurement, you probably shouldn't, but it depends on context.]

k-Medoids Clustering

Generalizes k-means beyond Euclidean distance.
Specify a distance fn $d(x, y)$ between points x, y, aka dissimilarity.
Can be arbitrary; ideally satisfies triangle inequality $d(x, y) \leq d(x, z) + d(z, y)$.
[Sometimes people use the ℓ_1 norm or the ℓ_∞ norm. Sometimes people specify a matrix of pairwise distances between the input points.]
[Suppose you have a database that tells you how many of each product each customer bought. You'd like to cluster together customers who buy similar products for market analysis. But if you cluster customers by Euclidean distance, you'll get one big cluster of all the customers who have only ever bought one thing. So Euclidean distance is not a good measure of dissimilarity. Instead, it makes more sense to treat each customer as a vector and measure the *angle* between two customers. If there's a large angle between customers, they're dissimilar.]
Replace mean computation with medoid, the sample point that minimizes total distance to other points in same cluster.
[So the medoid of a cluster is always one of the input points.]

[One difficulty with k-means is that you have to choose the number k of clusters before you start, and there isn't any reliable way to guess how many clusters will best fit the data. The next method, hierarchical clustering, has the advantage in that respect. By the way, there is a whole Wikipedia article on "Determining the number of clusters in a data set."]

Hierarchical Clustering

Creates a tree; every subtree is a cluster.
[So some clusters contain smaller clusters.]

Bottom-up, aka agglomerative clustering:
start with each point a cluster; repeatedly fuse pairs.
[Draw figure of points in the plane; pair clusters together until all points are in one top-level cluster.]

Top-down, aka divisive clustering:
start with all pts in one cluster; repeatedly split it.
[Draw figure of points in the plane; divide points into subsets hierarchically until each point is in its own subset.]

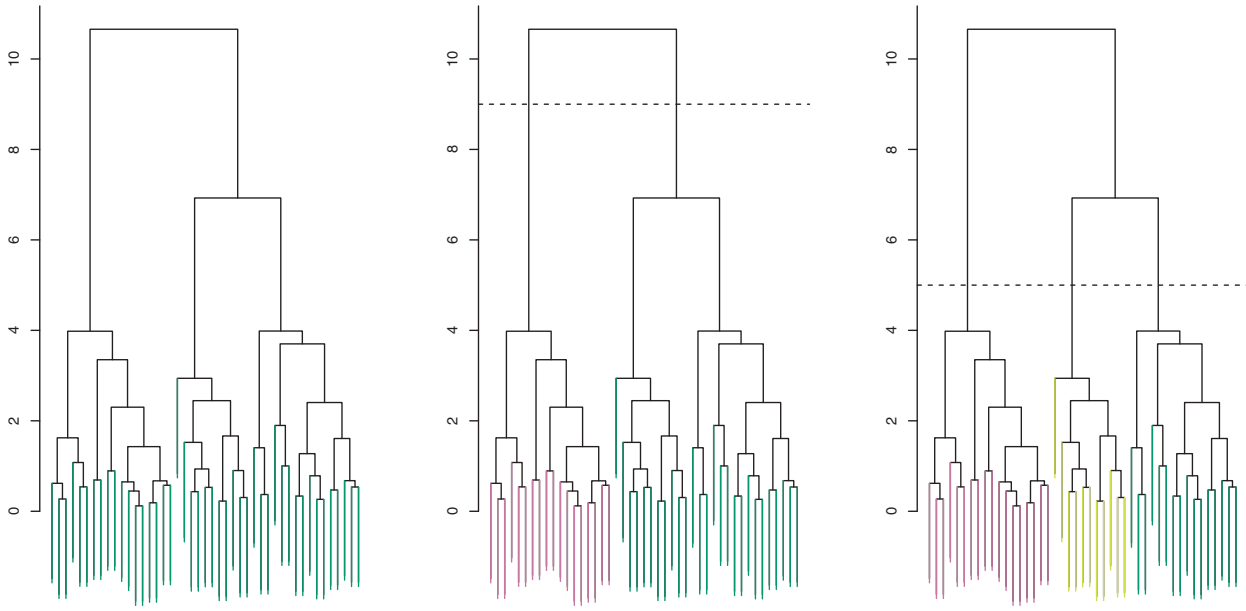
[When the input is a point set, agglomerative clustering is used much more in practice than divisive clustering. But when the input is a graph, it's the other way around: divisive clustering is more common.]

We need a distance fn for clusters A, B:

complete linkage: $d(A, B) = \max\{d(w, x) : w \in A, x \in B\}$
single linkage: $d(A, B) = \min\{d(w, x) : w \in A, x \in B\}$
average linkage: $d(A, B) = \frac{1}{|A||B|} \sum_{w \in A} \sum_{x \in B} d(w, x)$
centroid linkage: $d(A, B) = d(\mu_A, \mu_B)$ where μ_S is mean of S
[The first three of these linkages work for any distance function, even if the input is just a matrix of distances between all pairs of points. The centroid linkage only really makes sense if we're using the Euclidean distance. But there's a variation of the centroid linkage that uses the medoids instead of the means, and medoids are defined for any distance function. Moreover, medoids are more robust to outliers than means.]

Greedy agglomerative alg.:
Repeatedly fuse the two clusters that minimize $d(A, B)$
Naively takes $O(n^3)$ time.
[But for complete and single linkage, there are more sophisticated algorithms called CLINK and SLINK, which run in $O(n^2)$ time.]

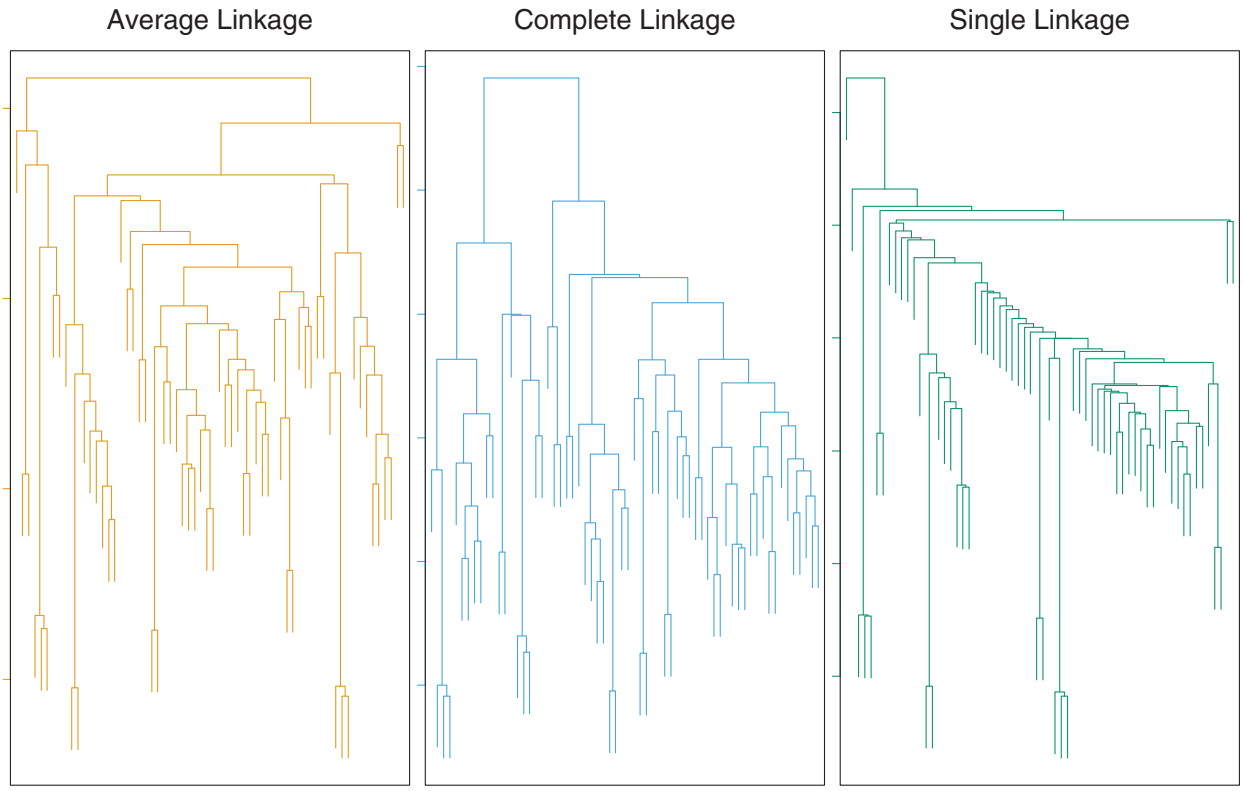
Dendrogram: Illustration of the cluster hierarchy (tree) in which the vertical axis encodes all the linkage distances.



dendrogram.pdf (ISL, Figure 10.9) [Example of a dendrogram cut into 1, 2, or 3 clusters.]

Cut dendrogram into clusters by horizontal line according to your choice of # of clusters OR intercluster distance.

[It's important to be aware that the horizontal axis of a dendrogram has no meaning. You could swap some node's left children and right children and it would still be the same dendrogram. It doesn't always mean anything that two leaves happen to be next to each other.]



linkages.pdf (ISL, Figure 10.12) [Comparison of average, complete (max), and single (min) linkages. Observe that the complete linkage gives the best-balanced dendrogram, whereas the single linkage gives a very unbalanced dendrogram that is sensitive to outliers (especially near the top of the dendrogram).]

[Probably the worst of these is the single linkage, because it's very sensitive to outliers. Notice that if you cut this example into three clusters, two of them have only one node. It also tends to give you a very unbalanced tree.]

[The complete linkage tends to be the best balanced, because when a cluster gets large, the furthest point in the cluster is always far away. So large clusters are more resistant to growth than small ones. If balanced clusters are your goal, this is your best choice.]

[In most cases you probably want the average or complete linkage.]

Warning: centroid linkage can cause inversions where a parent cluster is fused at a lower height than its children.

[So statisticians don't like it, but nevertheless, centroid linkage is popular in genomics.]

[As a final note, all the clustering algorithms we've studied so far are unstable, in the sense that deleting a few input points can sometimes give you very different results. But these unstable heuristics are still the most commonly used clustering algorithms. And it's not clear to me whether a truly stable clustering algorithm is even possible.]

22 Spectral Graph Clustering

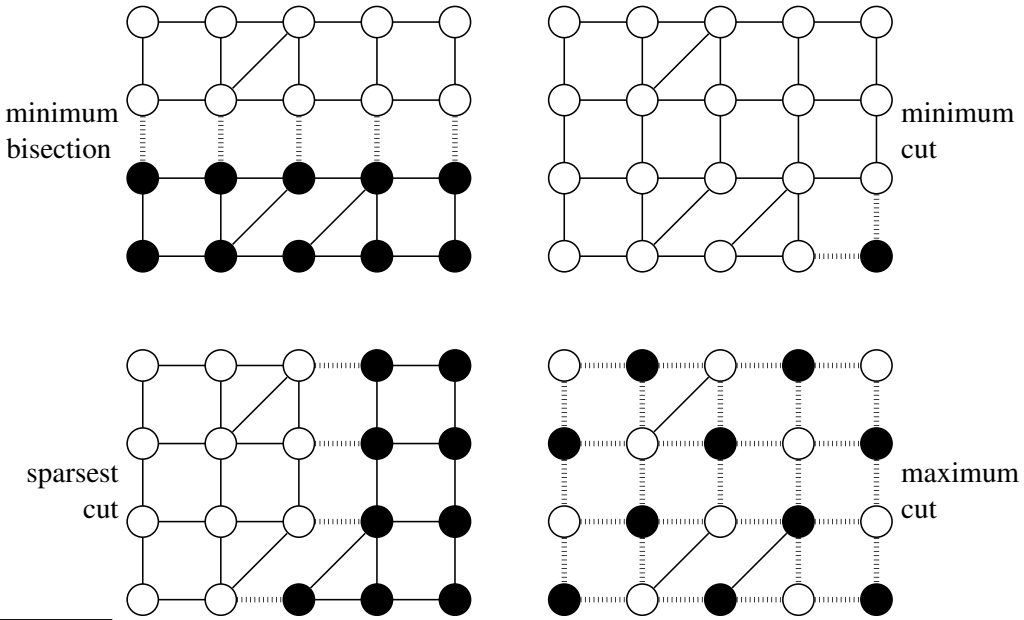
SPECTRAL GRAPH CLUSTERING

Input: Weighted, undirected graph $G = (V, E)$. No self-edges.
w_{ij} = weight of edge $(i, j) = (j, i)$; zero if $(i, j) \notin E$.

[Think of the edge weights as a similarity measure. A big weight means that the two vertices want to be in the same cluster. So the circumstances are the opposite of the last lecture on clustering. Then, we had a distance or dissimilarity function, so small numbers meant that points wanted to stay together. Today, big numbers mean that vertices want to stay together.]

Goal: Cut G into 2 (or more) pieces G_i of similar sizes,
but don't cut too much edge weight.
[That's a vague goal. There are many ways to make this precise.
Here's a typical goal, which we'll solve approximately.]
e.g. Minimize the sparsity $\frac{\text{Cut}(G_1,G_2)}{\text{Mass}(G_1)\,\text{Mass}(G_2)}$, aka cut ratio
where $\text{Cut}(G_1, G_2)$ = total weight of cut edges
$\text{Mass}(G_1)$ = # of vertices in G_1 OR assign masses to vertices

[The denominator "$\text{Mass}(G_1)\,\text{Mass}(G_2)$" penalizes imbalanced cuts.]



graph.pdf [Four cuts. All edges have weight 1.
Upper left: the minimum bisection; a bisection is perfectly balanced.
Upper right: the minimum cut. Usually very unbalanced; not what we want.
Lower left: the sparsest cut, which is good for many applications.
Lower right: the maximum cut; in this case also the maximum bisection.]

Sparsest cut, min bisection, max cut all NP-hard.
[We will look for an approximate solution to the sparsest cut problem.]

[We will turn this combinatorial graph cutting problem into algebra.]

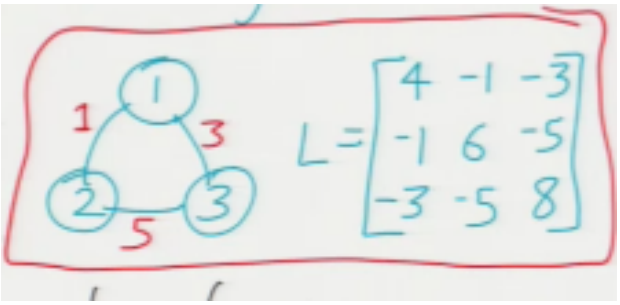
Let $n = |V|$. Let $y \in \mathbb{R}^n$ be an indicator vector:

$$y_i = \begin{cases} 1 & \text{vertex } i \in G_1, \\ -1 & \text{vertex } i \in G_2. \end{cases}$$

$$\text{Then } \frac{w_{ij}}{4}(y_i - y_j)^2 = \begin{cases} w_{ij} & (i,j) \text{ is cut}, \\ 0 & (i,j) \text{ is not cut}. \end{cases}$$

$$\begin{aligned} \text{Cut}(G_1, G_2) &= \sum_{(i,j)\in E} \frac{w_{ij}}{4}(y_i - y_j)^2 \\ &= \frac{1}{4} \sum_{(i,j)\in E} \left(w_{ij} y_i^2 - 2 w_{ij} y_i y_j + w_{ij} y_j^2 \right) \\ &= \frac{1}{4} \Bigg(\underbrace{\sum_{(i,j)\in E} -2 w_{ij} y_i y_j}_{\text{off-diagonal terms}} + \underbrace{\sum_{i=1}^{n} y_i^2 \sum_{k \neq i} w_{ik}}_{\text{diagonal terms}} \Bigg) \\ &= \frac{y^\top L y}{4}, \end{aligned}$$

$$\text{where } L_{ij} = \begin{cases} -w_{ij}, & i \neq j, \\ \sum_{k \neq i} w_{ik}, & i = j. \end{cases}$$



[Draw this by hand graphexample.png]

L is symmetric, $n \times n$ Laplacian matrix for G.

[L is effectively a matrix representation of G. For the purpose of partitioning a graph, there is no need to distinguish edges of weight zero from edges that are not in the graph.]
[We see that minimizing the weight of the cut is equivalent to minimizing the Laplacian quadratic form $y^\top L y$. This lets us turn graph partitioning into a problem in matrix algebra.]
[Usually we assume there are no negative weights, in which case $\text{Cut}(G_1, G_2)$ can never be negative, so it follows that L is positive semidefinite.]

If $y = \mathbf{1} = [1 \quad 1 \quad \ldots \quad 1]^\top$, then $\text{Cut}(G_1, G_2) = 0$, so
$\mathbf{1}$ is an eigenvector of L with eigenvalue 0.

[If G is connected and all the edge weights are positive, then this is the only zero eigenvalue. But if G is not connected, L has one zero eigenvalue for each connected component of G. It's easy to prove, but time prevents me.]

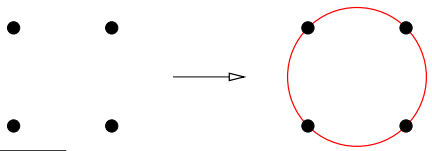
Bisection: exactly $n/2$ vertices in G_1, $n/2$ in G_2. Write $\mathbf{1}^\top y = 0$.
[So we have reduced graph bisection to this constrained optimization problem.]

Find y that minimizes $y^\top L y$	
subject to $\forall i,\ y_i = 1$ or $y_i = -1$	← binary constraint
and $\mathbf{1}^\top y = 0$	← balance constraint

Also NP-hard. We relax the binary constraint. → fractional vertices!

[A very common approach in combinatorial optimization algorithms is to relax some of the constraints so a discrete problem becomes a continuous problem. Intuitively, this means that you can put $1/3$ of vertex 7 in graph G_1 and the other $2/3$ of vertex 7 in graph G_2. You can even put $-1/2$ of vertex 7 in graph G_1 and $3/2$ of vertex 7 in graph G_2. This sounds crazy, but the continuous problem is much easier to solve than the combinatorial problem. After we solve it, we will round the vertex values to $+1/-1$, and we'll hope that our solution is still close to optimal.]
[But we can't just drop the binary constraint. We still need *some* constraint to rule out the solution $y = 0$.]

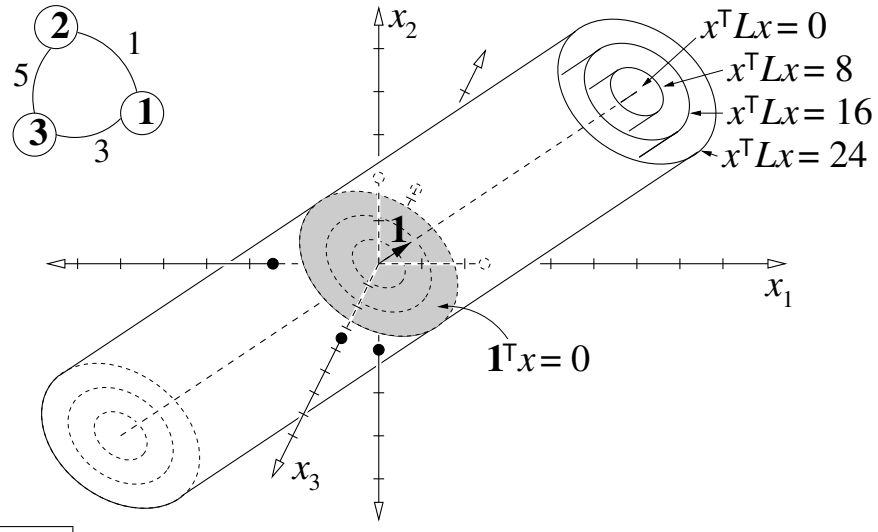
New constraint: y must lie on sphere of radius $\sqrt{n}$.



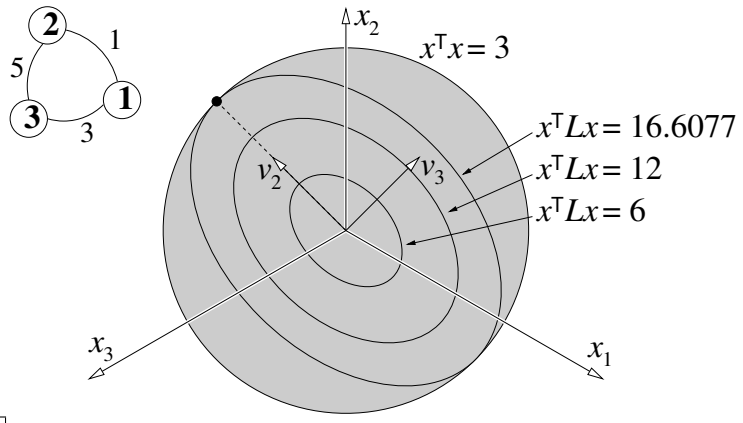
[Draw this by hand. circle.pdf] [Instead of constraining y to lie at a vertex of the hypercube, we constrain y to lie on the sphere through those vertices.]

Relaxed problem:

Minimize $y^\top L y$	
subject to $y^\top y = n$	} = Minimize $\dfrac{y^\top L y}{y^\top y}$ = Rayleigh quotient of L & y
and $\mathbf{1}^\top y = 0$	



cylinder.pdf [The isosurfaces of $y^\top L y$ are elliptical cylinders. The gray cross-section is the hyperplane $\mathbf{1}^\top = 0$. We seek the point that minimizes $y^\top L y$, subject to the constraints that it lies on the gray cross-section and that it lies on a sphere centered at the origin.]

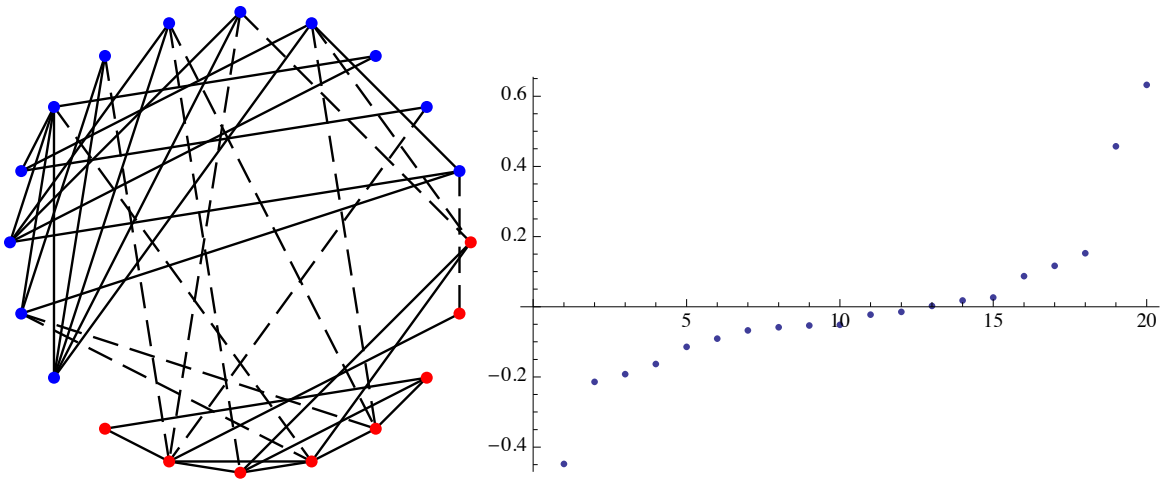


endview.pdf [The same isosurfaces restricted to the hyperplane $\mathbf{1}^T = 0$. The solution is constrained to lie on the outer circle.]

[You should remember this Rayleigh quotient from the lecture on PCA. As I said then, when you see a Rayleigh quotient, you should smell eigenvalues nearby. The y that minimizes this Rayleigh quotient is the eigenvector with the smallest eigenvalue. We already know what that eigenvector is: it's $\mathbf{1}$. But that violates our balance constraint. As you should recall from PCA, when you've used the most extreme eigenvector and you need an orthogonal one, the next-best optimizer of the Rayleigh quotient is the next eigenvector.]

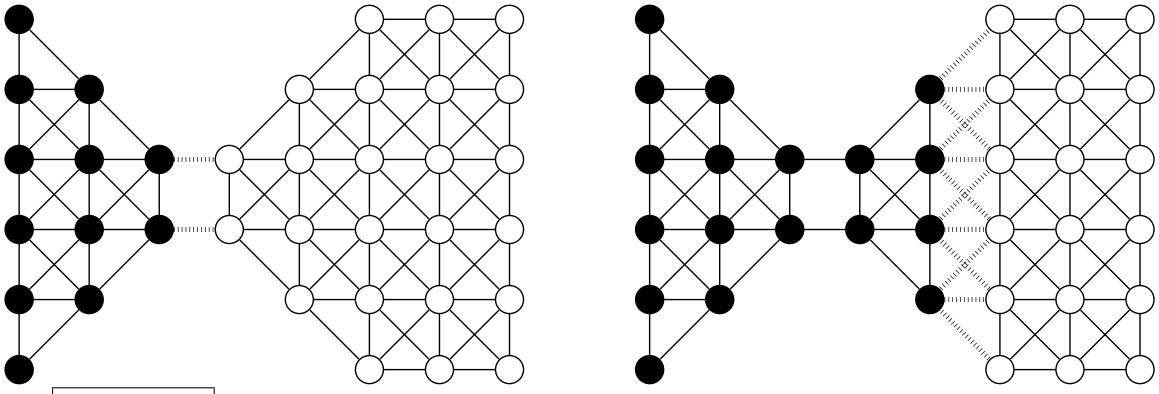
Let λ_2 = second-smallest eigenvalue of L.
Eigenvector v_2 is the Fiedler vector.
[It would be wonderful if every component of the Fiedler vector was 1 or -1, but that happens more or less never. So we round it. The simplest way is to round all positive entries to 1 and all negative entries to -1. But in both theory and practice, it's better to choose the threshold as follows.]

Spectral partitioning alg.:
- Compute Fiedler vector v_2 of L
- Round v_2 with a sweep cut:
  - = Sort components of v_2.
  - = Try the $n - 1$ cuts between successive components. Choose min-sparsity cut.
    [If we're clever about it, we can try all these cuts in time linear in the number of edges in G.]



specgraph.pdf, specvector.pdf [Left: example of a graph partitioned by the sweep cut. Right: what the un-rounded Fiedler vector looks like.]

[One consequence of relaxing the binary constraint is that the balance constraint no longer forces an exact bisection. But that's okay; we're cool with a slightly off-balance constraint if it means we cut fewer edges.]

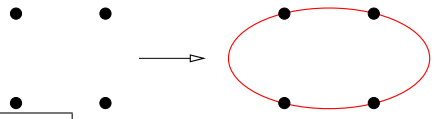


lopsided.pdf [A graph for which an off-balance cut (left) is better than a balanced one (right).]

Vertex Masses

[Sometimes you want the notion of balance to accord more prominence to some vertices than others. We can assign masses to vertices.]

Let M be diagonal matrix with vertex masses on diagonal.
New balance constraint: $\mathbf{1}^\top My = 0$.
[This new balance constraint says that G_1 and G_2 should each have the same total mass. It turns out that this new balance constraint is easier to satisfy if we also revise the sphere constraint a little bit.]
New ellipsoid constraint: $y^\top My = \text{Mass}(G) = \sum M_{ii}$.
[Instead of a sphere, now we constrain y to lie on an axis-aligned ellipsoid.]



[Draw this by hand. ellipse.pdf] [The constraint ellipsoid passes through the points of the hypercube.]

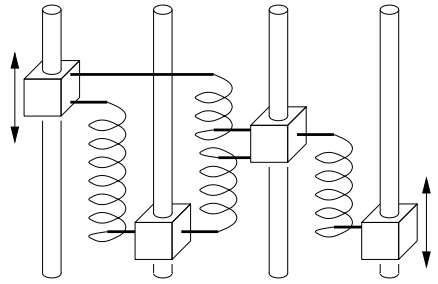
Now we want Fiedler vector of generalized eigensystem $Lv = \lambda Mv$.
[Most algorithms for computing eigenvectors and eigenvalues of symmetric matrices can easily be adapted to compute eigenvectors and eigenvalues of symmetric generalized eigensystems like this too.]

[Here's an interesting theorem for the grad students.]

Fact: Sweep cut finds a cut w/sparsity $\leq \sqrt{2\lambda_2 \max_i \frac{L_{ii}}{M_{ii}}}$;
Cheeger's inequality.
The optimal cut has sparsity $\geq \lambda_2/2$.

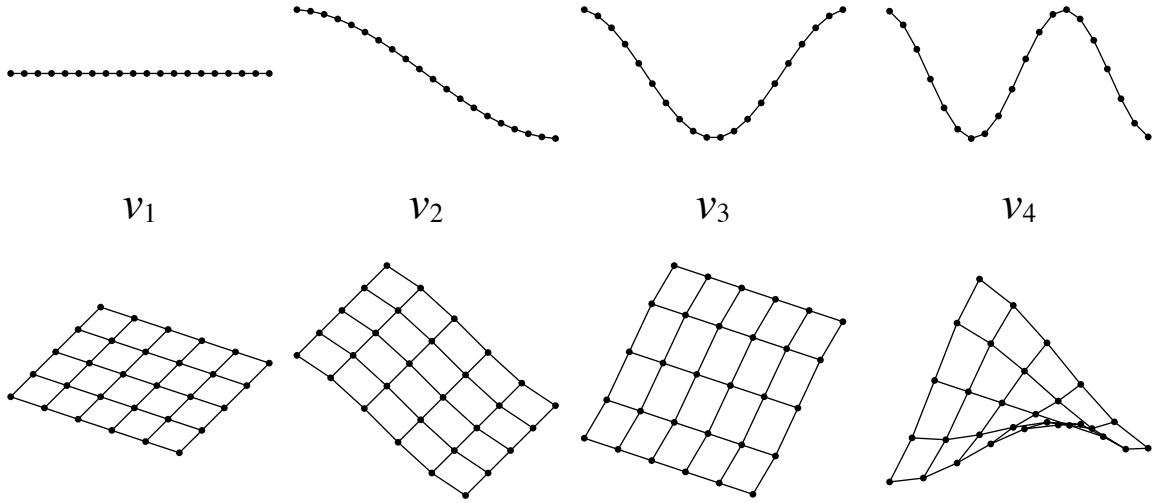
[So the spectral partitioning algorithm is an approximation algorithm, albeit not one with a constant factor of approximation. Cheeger's inequality is a very famous result in spectral graph theory, because it's one of the most important cases where you can relax a combinatorial optimization problem to a continuous optimization problem, round the solution, and still have a provably decent solution to the original combinatorial problem.]

Vibration Analogy



vibrate.pdf

[For intuition about spectral partitioning, think of the eigenvectors as vibrational modes in a physical system of springs and masses. Each vertex models a point mass that is constrained to move freely along a vertical rod. Each edge models a vertical spring with rest length zero and stiffness proportional to its weight, pulling two point masses together. The masses are free to oscillate sinusoidally on their rods. The eigenvectors of the generalized eigensystem $Lv = \lambda Mv$ are the vibrational modes of this physical system, and their eigenvalues are proportional to their frequencies.]



grids.pdf [Vibrational modes in a path graph and a grid graph.]

[These illustrations show the first four eigenvectors for two simple graphs. On the left, we see that the first eigenvector is the eigenvector of all 1's, which represents a vertical translation of all the masses in unison. That's not really a vibration, which is why the eigenvalue is zero. The second eigenvector is the Fiedler vector, which represents the vibrational mode with the lowest frequency. Each component indicates the amplitude with which the corresponding point mass oscillates. At any point in time as the masses vibrate, roughly half the mass is moving up while half is moving down. So it makes sense to cut between the positive components and the negative components. The third eigenvector also gives us a nice bisection of the grid graph, entirely different from the Fiedler vector. Some more sophisticated graph clustering algorithms use multiple eigenvectors.]

[I want to emphasize that spectral partitioning takes a global view of a graph. It looks at the whole gestalt of the graph and finds a good cut. By comparison, the clustering algorithms we saw last lecture were much more local in nature, so they're easier to fool.]

Greedy Divisive Clustering

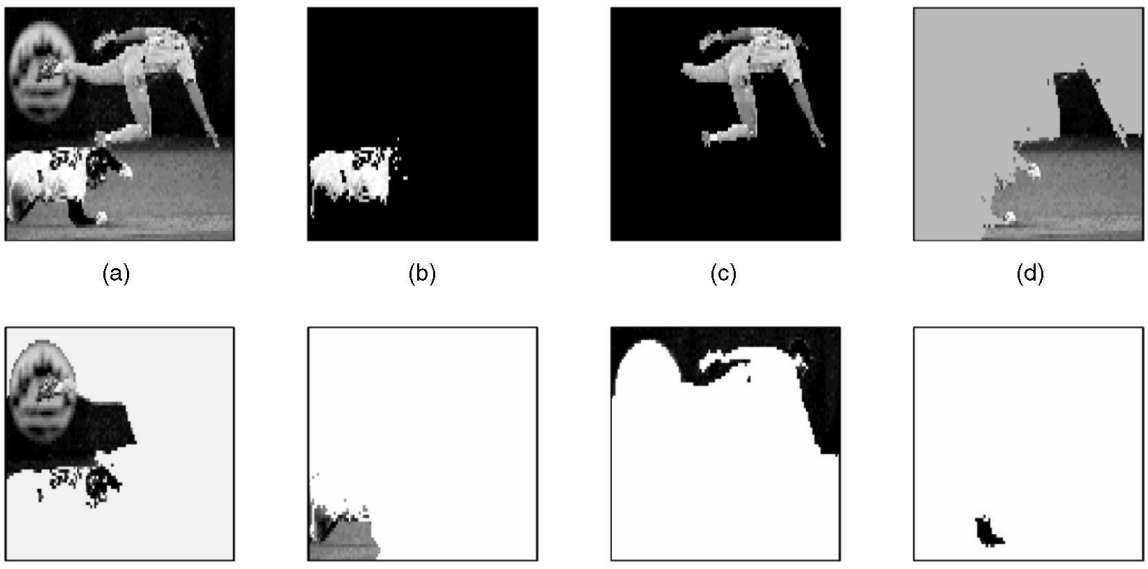
Partition G into 2 subgraphs; recursively cluster them.
[The sparsity is a good criterion for graph clustering. Use G's sparsest cut to divide it into two subgraphs, then recursively cut them. You can stop when you have the right number of clusters, or you could keep going until each subgraph is a single vertex and create a dendrogram.]
Can form a dendrogram, but it may have inversions.
[There's no reason to expect that the sparsity of a subgraph is smaller than the sparsity of the parent graph, so the dendrogram can have inversions. But it's still useful for getting an arbitrary number of clusters on demand.]

The Normalized Cut

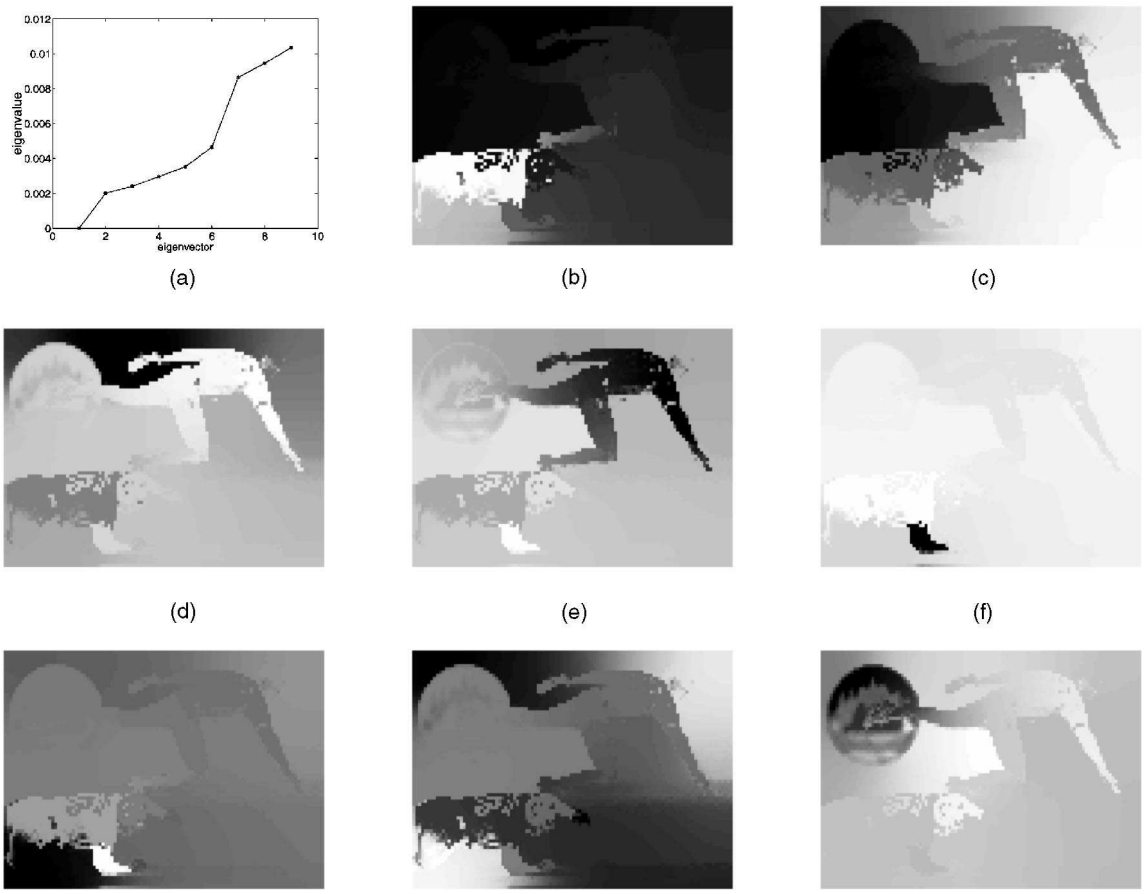
Set vertex i's mass $M_{ii} = L_{ii}$. [Sum of edge weights adjoining vertex i.]
[That is how we define a "normalized cut," which turns out to be a good choice for many different applications.]
Popular for image segmentation.
[Image segmentation is the problem of looking at a photograph and separating it into different objects. To do that, we define a graph on the pixels.]
For pixels with location w_i, brightness b_i, use graph weights

$$w_{ij} = \exp\left(-\frac{|w_i - w_j|^2}{\alpha} - \frac{|b_i - b_j|^2}{\beta}\right) \qquad \text{or zero if } |w_i - w_j| \text{ large.}$$

[We choose a distance threshold, typically less than 4 to 10 pixels apart. Pixels that are far from each other aren't connected. α and β are empirically chosen constants. It often makes sense to choose β proportional to the variance of the brightness values.]



baseballsegment.pdf (Shi and Malik, "Normalized Cut and Image Segmentation")
[A segmentation of a photo of a scene from a baseball game (upper left). The other figures show segments of the image extracted by recursive spectral partitioning.]



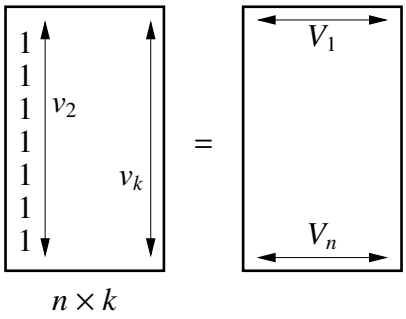
baseballvectors.pdf (Shi and Malik) [Eigenvectors 2–9 from the baseball image.]

Invented by [our own] Prof. Jitendra Malik and his student Jianbo Shi.

23 Multiple Eigenvectors; Latent Factor Analysis; Nearest Neighbors

Clustering w/Multiple Eigenvectors

For k clusters, compute first k eigenvectors $v_1 = \mathbf{1}, v_2, \ldots, v_k$ of generalized eigensystem $Lv = \lambda Mv$.

$V =$ 

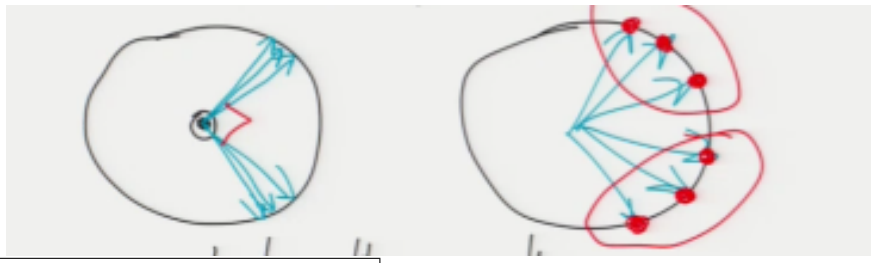
[V's columns are the eigenvectors with the k largest eigenvalues.]

[Yes, we do include the all-1's vector v as one of the columns of V.]

[Draw this by hand. eigenvectors.pdf]

Row V_i is spectral vector [my name] for vertex i. [These are vectors in a k-dimensional space I'll call the "spectral space." When we were using just one eigenvector, it made sense to cluster vertices together if their components were close together. When we use more than one eigenvector, it turns out that it makes sense to cluster vertices together if their spectral vectors point in similar directions.]
Normalize each row V_i to unit length.

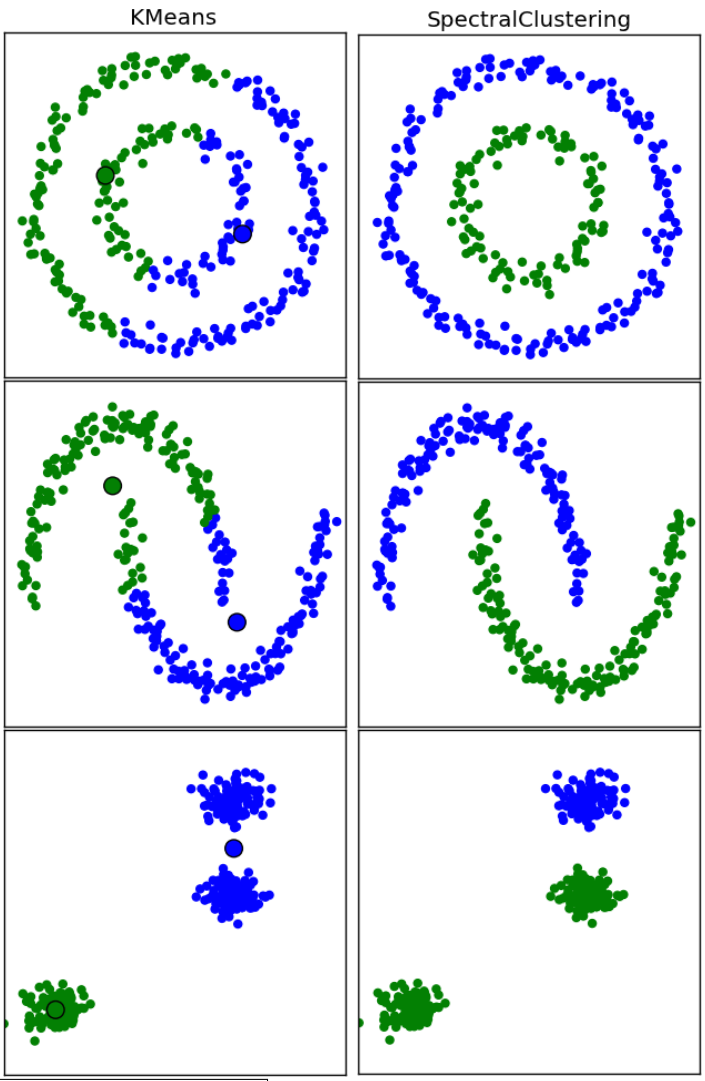
[Now you can think of the spectral vectors as points on a unit sphere centered at the origin.]



Draw this by hand (vectorclusters.png) [Draw a 2D example showing two clusters on a circle. If the input graph has k components, the points in each cluster will have identical spectral vectors that are exactly orthogonal to all the other components' spectral vectors (left). If we modify the graph by connecting these components with small-weight edges, we get vectors more like those at right—not exactly orthogonal, but still tending toward distinct clusters.]

k-means cluster these vectors.

[Because all the spectral vectors lie on the sphere, k-means clustering will cluster together vectors that are separated by small angles.]



compkmeans.png, compspectral.png [Comparison of point sets clustered by k-means—just k-means by itself, that is—vs. a spectral method. To create a graph for the spectral method, we use an exponentially decaying function to assign weights to pairs of points, like we used for image segmentation but without the brightnesses.]

Invented by [our own] Prof. Michael Jordan, Stanford Prof. Andrew Ng [when he was still a student at Berkeley], Yair Weiss.

[This wasn't the first algorithm that uses multiple eigenvectors for spectral clustering, but it has become one of the most popular.]

LATENT FACTOR ANALYSIS [aka latent semantic indexing]

[You can think of this as dimensionality reduction for matrices.]

Suppose X is a term-document matrix: [aka bag-of-words model]
row i represents document i; column j represents term j. [Term = word.]
[Term-document matrices are usually sparse, meaning most entries are zero.]
X_{ij} = occurrences of term j in doc i
better: log (1+ occurrences) [So frequent words don't dominate.]
[Better still is to weight the entries so rare words give big entries and common words like "the" give small entries. I'll omit the details.]

Recall SVD $X = UDV^\top = \sum_{i=1}^{d} \delta_i u_i v_i^\top$. Suppose $\delta_i \le \delta_j$ for $i \ge j$.

(Unlike PCA, we usually don't center X.)
For greatest δ_i, each v_i lists terms in a genre/cluster of documents
each u_i " docus using similar/related terms
E.g. u_1 might have large components for the romance novels,
v_1 " " " " for terms "passion," "ravish," "bodice" ...
[... and δ_1 would give us an idea how much bigger the romance novel market is than the markets for every other genre of books.]

[v_1 and u_1 tell us that there is a large subset of books that tend to use the same large subset of words. We can read off the words by looking at the larger components of v_1, and we can read off the books by looking at the larger components of u_1.]

[The property of being a romance novel is an example of a latent factor. So is the property of being the sort of word used in romance novels. There's nothing in X that tells you explicitly that romance novels exist, but the genre is a hidden connection between them that gives them a large singular value. The vector u_1 reveals which books have that genre, and v_1 reveals which words are emphasized in that genre.]

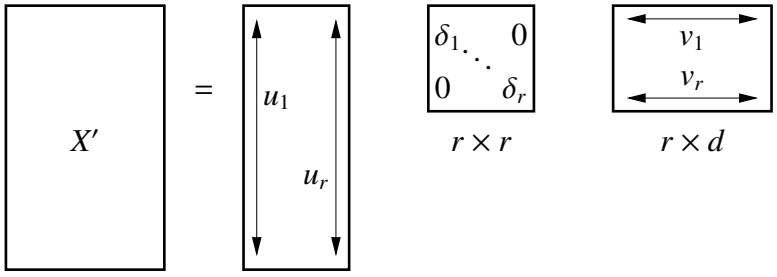
Like clustering, but clusters overlap: if u_1 picks out romances &
u_2 picks out histories, they both pick out historical romances.

[So you can think of latent factor analysis as a sort of clustering that permits clusters to overlap. Another way in which it differs from traditional clustering is that the u-vectors contain real numbers, and so some points have stronger cluster membership than others. One book might be just a bit romance, another a lot.]

Application in market research:
identifying consumer types (hipster, soccer mom) & items bought together.
[For applications like this, the first few singular vectors are the most useful. Most of the singular vectors are mostly noise, and they have small singular values to tell you so.]

Truncated sum $X' = \sum_{i=1}^{r} \delta_i u_i v_i^\top$ is low-rank approximation (rank r) of X.

[We choose the singular vectors with the largest singular values, because they carry the most/best information.]



[Draw this by hand. truncate.pdf]

X' is the rank-r matrix that minimizes the [squared] Frobenius norm

$$\|X - X'\|_F^2 = \sum_{i,j} \left(X_{ij} - X'_{ij}\right)^2$$

Applications:

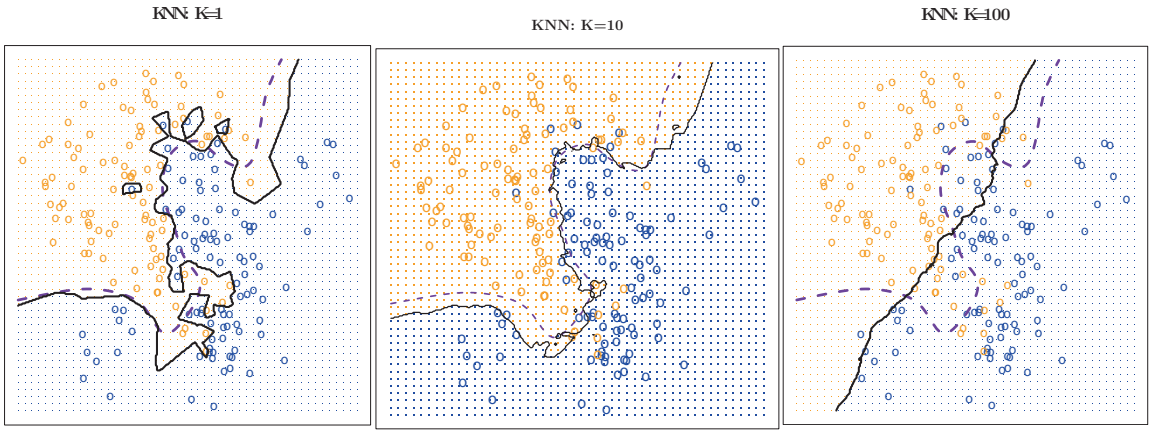
- Fuzzy search. [Suppose you want to find a document about gasoline prices, but the document you want doesn't have the word "gasoline"; it has the word "petrol." One cool thing about the reduced-rank matrix X' is that it will probably associate that document with "gasoline," because the SVD tends to group synonyms together.]
- Denoising. [The idea is to assume that X is a noisy measurement of some unknown matrix which probably has low rank. If that assumption is partly true, then the reduced-rank matrix X' might be better than the input X.]
- Collaborative filtering: fills in unknown values, e.g. user ratings.
  [Suppose the rows of X represents Netflix users and the columns represent movies. The entry X_{ij} is the review score that user i gave to movie j. But most users haven't reviewed most movies. Just as the rank reduction will associate "petrol" with "gasoline," it will tend to associate users with similar tastes in movies, so the reduced-rank matrix X' can predict ratings for users who didn't supply any. You'll try this out in the last homework.]

NEAREST NEIGHBOR CLASSIFICATION

[We're done with unsupervised learning. Now I'm going back to classifiers, and I saved one of the simplest for the end of the semester.]

Idea: Given query point v, find the k input points nearest v.
Distance metric of your choice.
Regression: Return average value of the k points.
Classification: Return class with the most votes from the k points OR
return histogram of class probabilities.

[Obviously, the histogram of class probabilities has limited precision. If $k = 3$, then the only probabilities you'll ever return are 0, 1/3, 2/3, or 1. You can improve the precision by making k larger, but you might underfit. It works best when you have a huge amount of data.]



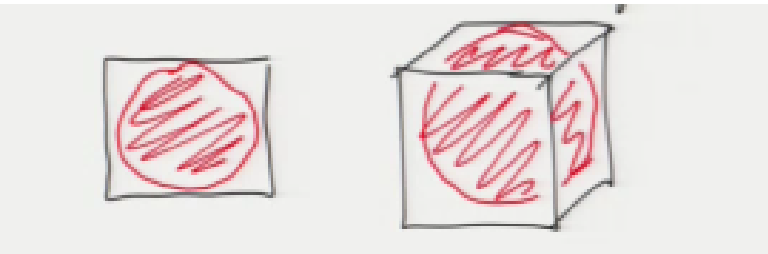
allnn.pdf (ISL, Figures 2.15, 2.16) [Examples of 1-NN, 10-NN, and 100-NN. A larger k smooths out the boundary. In this example, the 1-NN classifier is badly overfitting the data, and the 100-NN classifier is badly underfitting. The 10-NN classifier does well: it's reasonably close to the Bayes decision boundary. Generally, the ideal k depends on how dense your data is. As your data gets denser, the optimal k increases.]

[There are theorems showing that if you have a lot of data, nearest neighbors can work quite well.]
Theorem (Cover & Hart, 1967):
As $n \to \infty$, the 1-NN error rate is $< B(2 - B)$ where B = Bayes risk.
if only 2 classes, $\leq 2B(1 - B)$
[There are a few technical requirements of this theorem. The most important is that the training points and the test points all have to be drawn independently from the same probability distribution. The theorem applies to any separable metric space, so it's not just for the Euclidean metric.]

Theorem (Fix & Hodges, 1951):
As $n \to \infty$, $k \to \infty$, $k/n \to 0$, k-NN error rate converges to B. [Which means optimal.]

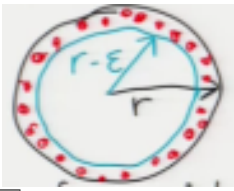
The Geometry of High-Dimensional Spaces

Consider unit ball $B = \{p \in \mathbb{R}^d : |p| \leq 1\}$
& hypercube $H = \{p \in \mathbb{R}^d : |p_i| \leq 1\}$



Draw this by hand (unitball.png) [2D circle in square, 3D ball in cube.]

[In two dimensions, it looks like the circle fills most of the square. But in 100 dimensions, the ball takes almost no volume compared to the hypercube, and the corners of the cube are a distance of 10 away from the center. And since there are 2^{100} corners, there's a lot of volume out toward the corners.]
Consider a shell of the sphere.



Draw this by hand (concentric.png) [Ball of radius r, and concentric ball of radius $r - \epsilon$ inside. In high dimensions, almost every point chosen uniformly at random in the outer hypersphere lies outside the inner hypersphere.]

Volume of outer ball $\propto r^d$
Volume of inner ball $\propto (r - \epsilon)^d$
Ratio of inner ball volume to outer =

$$\frac{(r-\epsilon)^d}{r^d} = \left(1 - \frac{\epsilon}{r}\right)^d \approx \exp\left(-\frac{\epsilon d}{r}\right) \quad \text{for large } d \to \text{small!}$$

E.g. if $\frac{\epsilon}{r} = 0.1$ & $d = 100$, inner ball has 0.0027% of volume.

Random points from uniform distribution in ball: nearly all are in outer shell.
" " " Gaussian " " " " " " " some "

[In other words, if the dimension is very high, the majority of the random points generated from an isotropic Gaussian distribution are approximately at the same distance from the center. So they lie in a thin shell. As the dimension grows, the *standard deviation* of a random point's distance to the center gets smaller and smaller compared to the distance itself.]

[This is one of the things that makes machine learning hard in high dimensions. Sometimes the farthest points aren't much farther away than the nearest ones.]

Exhaustive k-NN alg.

Given query point v:

- Scan through all n input points, computing (squared) distances to v.
- Maintain max-heap with the k shortest distances seen so far.
  [Whenever you encounter an input point closer to v than the point at the top of the heap, you remove the heap-top point and insert the better point. Obviously you don't need a heap if $k = 1$ or even 3, but if $k = 100$ a heap will substantially speed up keeping track of the distance to beat.]

Time to construct the classifier: 0
[This is the only $O(0)$-time algorithm we'll learn this semester.]
Query time: $O(nd + n \log k)$
expected $O(nd + k \log^2 k)$ if random point order
[It's a cute theoretical observation that you can slightly improve the expected running time by randomizing the point order so that only $O(k \log k)$ heap operations occur. But in practice I don't recommend it; you'll probably lose more from the cache than you'll gain from randomization.]

Speeding Up NN

Can we preprocess the training points to obtain sublinear query time?

Very low dimensions: Voronoi diagrams
Medium dim (up to ~ 30): k-d trees
Larger dim: locality sensitive hashing [still researchy, not widely adopted]
Largest dim: no [stick with exhaustive k-NN.]

Can use PCA or other dimensionality reduction as preprocess

[Most fast nearest-neighbor algorithms in more than a few dimensions are *approximate* nearest neighbor algorithms; we don't necessarily expect to find the exact nearest neighbors. That's usually okay, as machine learning classifiers are rarely perfect. If we use PCA as a preprocess, then it's even more approximate, but it's much faster.]

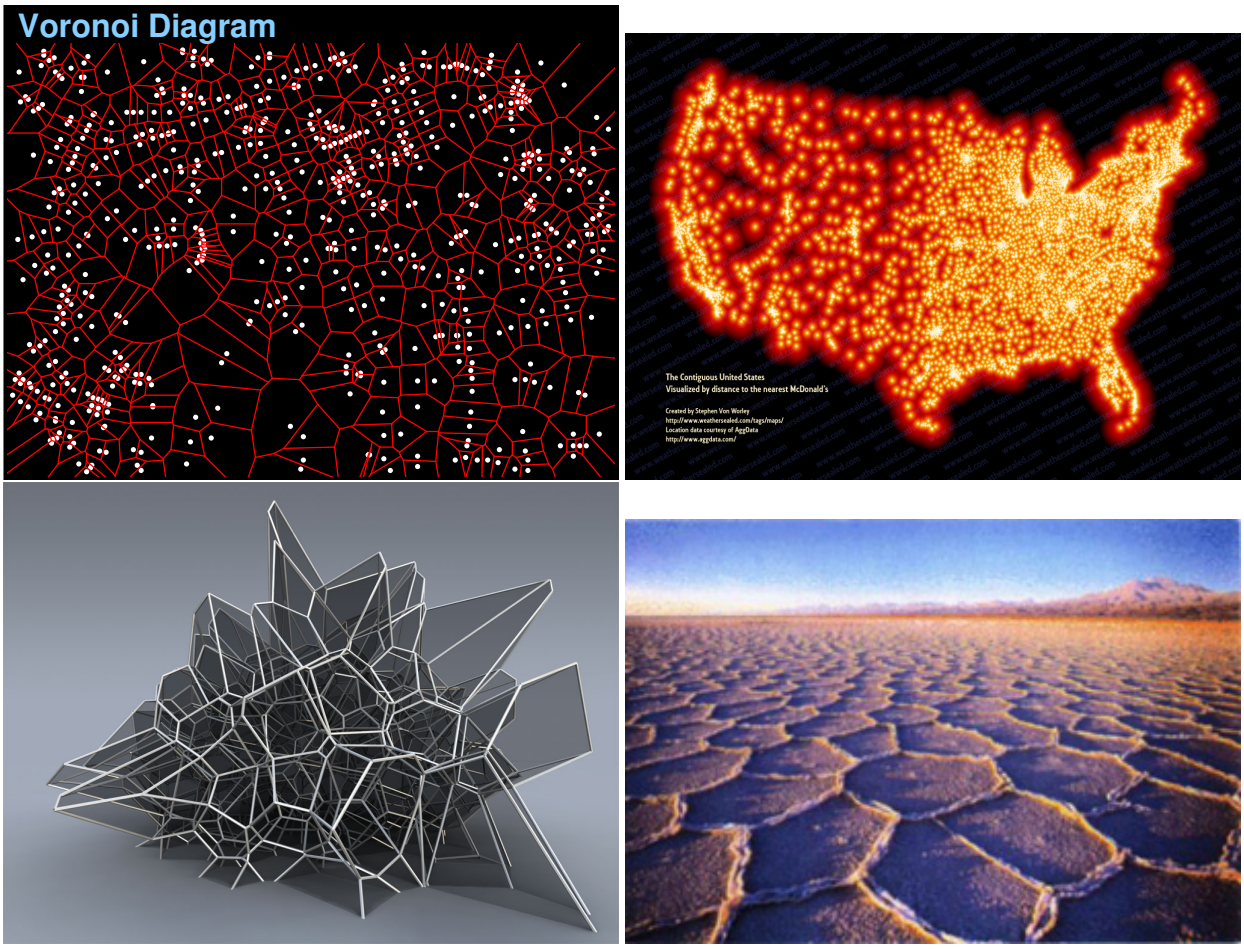
24 More Nearest Neighbors: Voronoi Diagrams and k-d Trees

SPEEDING UP NN (continued)

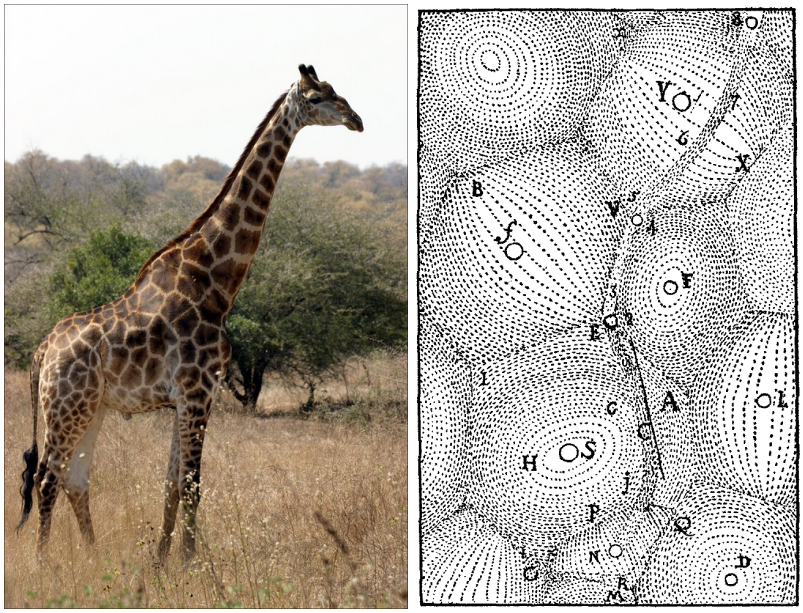
[Recall from last lecture that Voronoi diagrams can be used in very low dimensions, 2 through perhaps 5; and k-d trees are used for moderate dimensions, up to about 30. In high-dimensional spaces, people usually use exhaustive search or they apply dimensionality reduction so they can use k-d trees. Occasionally people use locality-sensitive hashing, and it will be interesting to see if it develops to the point where it reliably beats exhaustive search on many applications.]

Voronoi Diagrams

Let P be a point set. The Voronoi cell of $w \in P$ is
Vor $w = \{p \in \mathbb{R}^d : |pw| \leq |pv| \quad \forall v \in P\}$
[A Voronoi cell is always a convex polyhedron or polytope.]
The Voronoi diagram of P is the set of P's Voronoi cells.



voronoi.pdf, vormcdonalds.jpg, voronoiGregorEichinger.jpg, saltflat-1.jpg
[Voronoi diagrams sometimes arise in nature (salt flats, giraffe).]



giraffe-1.jpg, vortex.pdf

[Believe it or not, the first published Voronoi diagram dates back to 1644, in the book "Principia Philosophiae" by the famous mathematician and philosopher René Descartes. He claimed that the solar system consists of vortices. In each region, matter is revolving around one of the fixed stars (vortex.pdf). His physics was wrong, but his idea of dividing space into polyhedral regions has survived.]

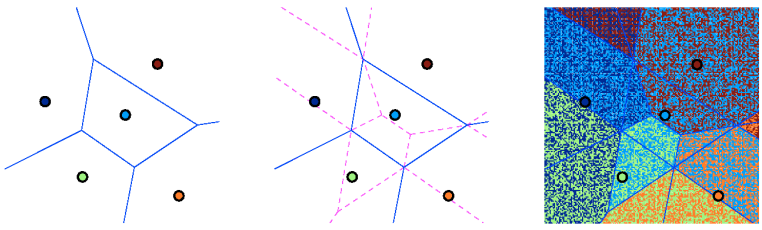
Size (e.g. # of vertices) $\in O(n^{\lceil d/2 \rceil})$
[This upper bound is tight when d is a small constant. As d grows, the tightest upper bound is somewhat smaller than this, but the complexity still grows exponentially with d.]
. . . but often in practice it is $O(n)$.
[Here I'm leaving out a constant that grows exponentially with d.]

Point location: Given query point v, find the point $w \in P$ for which $v \in \text{Vor } w$.

2D: $O(n \log n)$ time to compute V.d. and a trapezoidal map for pt location
$O(\log n)$ query time [because of the trapezoidal map]
[That's a pretty great running time compared to the linear query time of exhaustive search.]

dD: Use binary space partition tree (BSP tree) for pt location
[Unfortunately, it's difficult to characterize the running time of this strategy, although it is likely to be reasonably fast.]

1-NN only! What about k-NN?
order-k Voronoi diagram has a cell for each possible k nearest neighbors.



2ordervoronoi.pdf [An order-2 Voronoi diagram.]

In 2D, size $\in O(k^2 n)$
[Order-k Voronoi diagrams are rarely used in practice for, partly because the size grows rapidly with k; partly because there's no software available. Voronoi diagrams are good for 1-nearest neighbor in 2 or 3 dimensions, maybe 4 or 5, but for anything beyond that, k-d trees are much simpler and probably faster.]

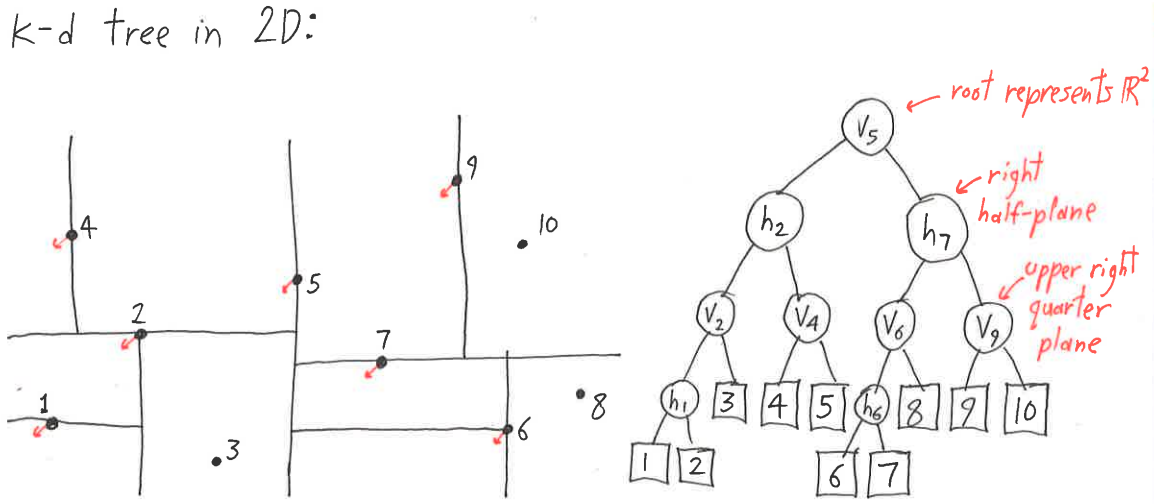
[There are also Voronoi diagrams for other distance metrics, like the L_1 and L_∞ norms.]

[If you want to know how to compute Voronoi diagrams and point location data structures for them, consider my course CS 274, Computational Geometry, which I'll probably teach next spring.]

k-d Trees

Decision trees for NN search. Differences: [compared to decision trees]

- No entropy. Split dimension w/greatest variance or width (max − min).
  [With nearest neighbor search, we don't care about the entropy. Instead, what we want is that if we draw a sphere around the query point, it won't intersect very many boxes of the decision tree. So it helps if the boxes are nearly cubical, rather than long and thin.]
- Each internal node stores a sample point. [...that lies in the node's box.]
  [We don't have points only at the leaves; we have them at internal nodes too, because when we search the tree for a query point, we want to stop searching as early as possible.]



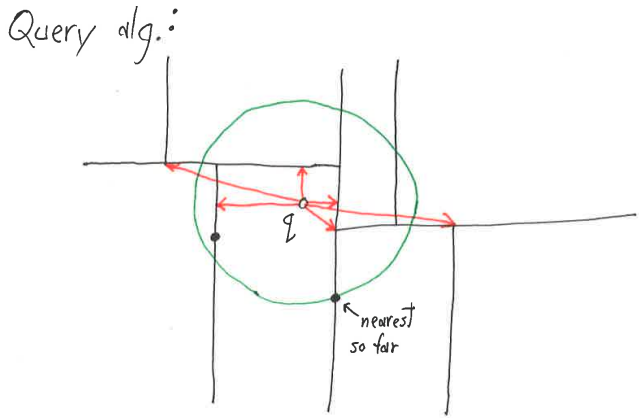
[Draw this by hand. kdtreestructure.pdf]

Given query pt q, find a sample pt p such that $|qp| \leq (1+\epsilon)\,|qs|$, where s is the closest sample pt.
[If $\epsilon = 0$, we're asking for an exact nearest neighbor; if ϵ is positive, we're asking for an approximate nearest neighbor.]

The alg. maintains:

- Nearest neighbor found so far (or k nearest). goes down ↓
- Heap of unexplored subtrees, keyed by distance from q. goes up ↑

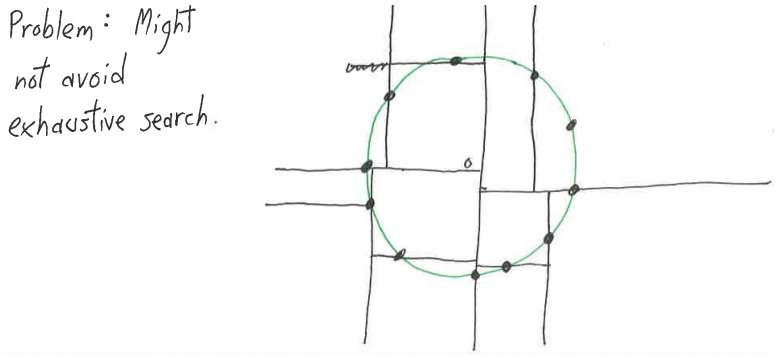
[Each subtree represents a box, and we measure the distance from q to the nearest point in that box and use it as a key for the subtree in the heap. The search stops when the distance to the kth-nearest neighbor found so far and the distance to the nearest unexplored box meet in the middle.]



[Draw this by hand. kdtreequery.pdf] [A search in progress.]

$Q \leftarrow$ heap containing root node of tree
$r \leftarrow \infty$
while Q has a cell closer to q than $\frac{r}{1+\epsilon}$
    $C \leftarrow$ removemin(Q)
    $p \leftarrow C$'s sample point
    $r \leftarrow \min\{r, |qp|\}$ [optimization: store squares of distances instead.]
    $C', C'' \leftarrow$ child cells of C
    insert(Q, C') [optimization: check whether
    insert(Q, C'') distance $\geq \frac{r}{1+\epsilon}$ before inserting.]
return point that determined r

[This procedure tries to avoid searching most of the subtrees by searching the boxes close to q first.]



[Draw this by hand. kdtreeproblem.pdf] [Example of a worst-case exact search query.]

[In the worst case, we may have to look at every node in the k-d tree to find the nearest neighbor. In that case, the k-d tree is slower than simple exhaustive search. This is an example where the *approximate* nearest neighbor algorithm can be much faster. In practice, settling for an approximate nearest neighbor sometimes improves the speed by a factor of 10 or even 100, because you don't need to look at most of the tree to do a query.]

For k-NN, replace "r" with a max-heap holding the k nearest neighbors
[. . . just like in the exhaustive search algorithm I discussed last lecture.]

Works with any L_p norm for $p \in [1, \infty]$.
[k-d trees are not limited to the Euclidean (L_2) norm.]

Software:
- ANN (David Mount & Sunil Arya, U. Maryland)
- FLANN (Marius Muja & David Lowe, U. British Columbia)
- GeRaF (Georgios Samaras, U. Athens) [random forests!]

[I want to emphasize the fact that exhaustive nearest neighbor search really is one of the first classifiers you should try in practice, even if it seems too simple. So here's an example of a modern research paper that uses 1-NN and 120-NN search to solve a problem . . .]

Example: im2gps

Paper by James Hays and [our own] Prof. Alexei Efros.
[Goal: given a query photograph, determine where on the planet the photo was taken. Called geolocalization. They evaluated both 1-NN and 120-NN with a complex set of features. What they did not do, however, is treat each photograph as one long vector. That's okay for tiny digits, but too expensive for online travel photographs. Instead, they reduced each photo to a small descriptor made up of a variety of features that extract the essence of each photo.]
[Show slides (im2gps.pdf). Sorry, images not included here.]

[Features, in rough order from most effective to least:
1. GIST: A compact "image descriptor" based on oriented edge detection (Gabor filters) + histograms.
2. Textons: A histogram of textures, created after assembling a dictionary of common textures.
3. A shrunk 16×16 image.
4. A color histogram.
5. Another histogram of edges, this one based on the Canny edge detector, invented by our own Prof. John Canny.
6. A geometric descriptor that's particularly good for identifying ground, sky, and vertical lines.]

[Bottom line: With 120-NN, their most sophisticated implementation came within 64 km of the correct location about 50% of the time.]

RELATED CLASSES

[If you like machine learning and you'll still be here next year, here are some courses you might want to take.]

CS C281A: Statistical Learning Theory (fall) [C281A is the most direct continuation of CS 189/289A.]
EE 127 (spring), EE 227BT (fall): Numerical optimization [a core part of ML]
[It's hard to overemphasize the importance of numerical optimization to machine learning, as well as other fields of CS like graphics and theory.]
EE 126 (both): Random Processes [Markov chains, expectation maximization, PageRank]
EE C106A/B (fall/spring): Intro to Robotics [dynamics, control, sensing]
Math 110: Linear Algebra [but the real gold is in Math 221]
Math 221: Matrix Computations (fall?) [how to compute SVDs, eigenvectors, etc.]
CS C280 (spring): Computer Vision
CS C267 (spring): Scientific Computing [parallelization, practical matrix algebra, some graph partitioning]
CS 298-115 (fall): Interactive Robotics; Anca Dragan
CS 194-26 (fall): Computational Photography; Alyosha Efros
CS 274 (spring): Computational Geometry; me [k-d trees, Voronoi diagrams, geometric search structures]