Concise Machine Learning

Jonathan Richard Shewchuk
May 5, 2025

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, including five and a half lectures on neural networks, and a few methods for clustering and dimensionality reduction. It is concise because nothing is included that cannot be written or spoken in a single semester's lectures (with whiteboard lectures and almost no 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 Awards CCF-1423560 and CCF-1909204, 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, density estimation, dimensionality reduction, clustering, perceptrons, support vector machines (SVMs), Gaussian discriminant analysis, linear discriminant analysis (LDA), quadratic discriminant analysis (QDA), logistic regression, decision trees, random forests, ensemble learning, bagging, boosting, AdaBoost, neural networks, convolutional neural networks (CNNs, ConvNets), residual neural networks (ResNets), batch normalization, AdamW, nearest neighbor search, least-squares linear regression, logistic regression, polynomial regression, ridge regression, Lasso, bias-variance decomposition, maximum likelihood estimation (MLE), principal components analysis (PCA), singular value decomposition (SVD), random projection, latent factor analysis, latent semantic indexing, k-means clustering, hierarchical clustering, spectral graph clustering, the kernel trick, learning theory

Contents

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 include almost nothing that I do not have time to write or speak during one fourteen-week semester of twice-weekly 80-minute lectures. (After holidays and the midterm exam, that amounts to 25 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 one brief segment during the semester (to discuss the V1 visual cortex).

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 think that the 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 need additional readings or discussion sections for greater detail. My class web page lists additional readings for most of the lectures, many of them from three textbooks that have been kindly made available for free on the web: *An Introduction to Statistical Learning with Applications in R*,[2] second edition, by Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani, Springer, New York, 2021, ISBN # 978-1-0716-1417-4; *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; and *Deep Learning*[4] by Christopher M. Bishop with Hugh Bishop, Springer, 2024. Readers wanting the verbose kind of "lecture notes" should consider the fine ones written by Stanford University's Andrew Ng.[5] 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 said, "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.

[1] https://people.eecs.berkeley.edu/~jrs/189/
[2] https://www.statlearning.com
[3] https://hastie.su.domains/ElemStatLearn/
[4] https://www.bishopbook.com
[5] http://cs229.stanford.edu/notes2020spring/

1 Introduction; Classification; Train, Validate, Test

CS 189 / 289A [Spring 2025]
Machine Learning
Jonathan Shewchuk

https://people.eecs.berkeley.edu/~jrs/189/

Homework 1 due next Wednesday.

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

Discussion sections (Tue & Wed):
Attend any section. [We'll put up a list on Ed Discussion.]
[We might have a few advanced sections, including research discussion or exam problem preparation.]
Sections start Tuesday. [Next week.]

[Enrollment: 736 students max. 349 waitlisted. Expecting many drops. EECS grads have highest priority; CD/DS undergrads second; non-EECS grads third; a few concurrent enrollment students will be admitted.]

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

Prerequisites

Vector calculus: Math 53 [or another vector calculus course]
Linear algebra: Math 54, Math 110, or EE 16A+16B [or another linear algebra course]
Probability: CS 70, EECS 126, or Stat 134 [or another probability course]
Plentiful programming experience [TAs have no obligation to look at your code.]
NOT CS 188

Grading: 189

40% 7 Homeworks. Late policy: 5 slip days total
20% Midterm: Monday, March 17, 7:00–9:00 PM
40% Final Exam: Friday, May 16, 3–6 PM (not on Berkeley time)

Grading: 289A

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

Cheating

- Discussion of HW problems is encouraged. Showing other students *small* amounts of code is okay.
- All homeworks, including programming, must be written individually.
- All code must be typed by you. Do not use LLMs, Autopilot, or chatbots to autocomplete or write code, nor to answer math problems.
- You may use LLMs or chatbots to help debugging or understanding, but you MUST include a complete transcript of the conversation in an appendix at the end of your homework.
- 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 year, we had to punish 12 people for cheating. It was not fun. Please don't make me do it again.]

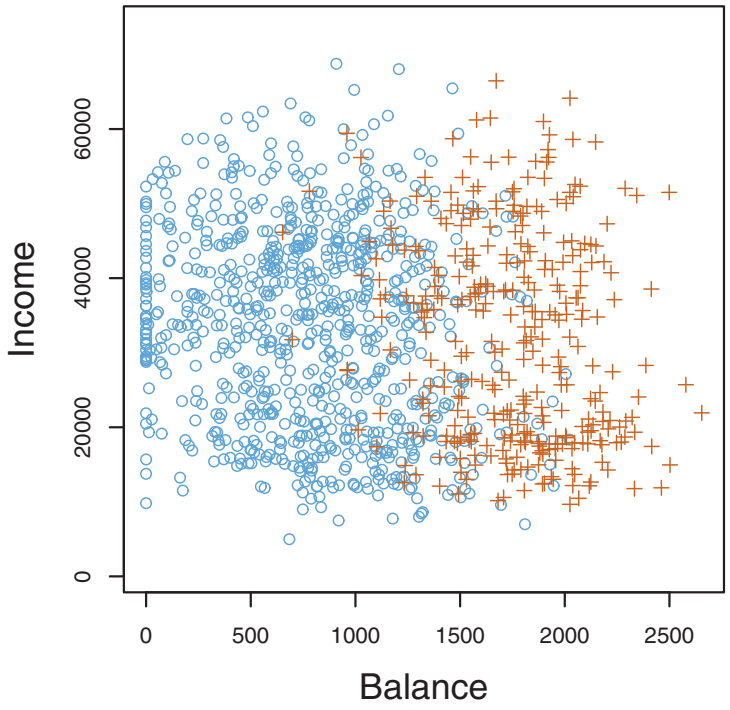
CORE MATERIAL

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

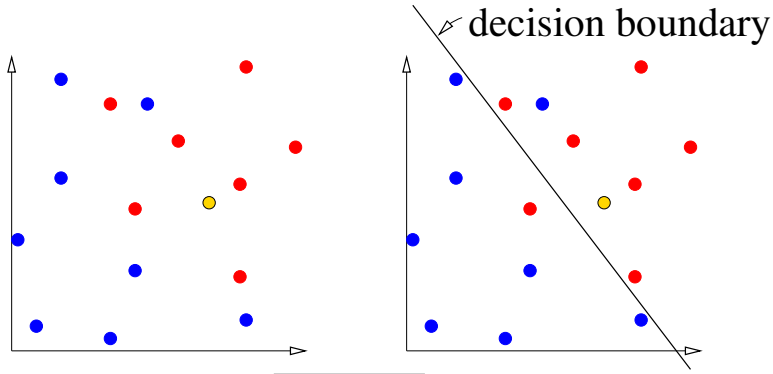
[The most important part of this is the data. Data drives everything else.
You cannot learn much if you don't have enough data.
You cannot learn much if your data has bad quality.
But it's amazing what you can do if you have lots of good data.
Machine learning has changed a lot in the last two decades 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.
Neural networks had fallen out of favor early this millennium, but they came back big around 2012 because researchers found that they work so much better when you have vast quantities of data.]

CLASSIFICATION

- Collect training points with class labels: reliable debtors & defaulted debtors
- Evaluate new applicants—predict their class

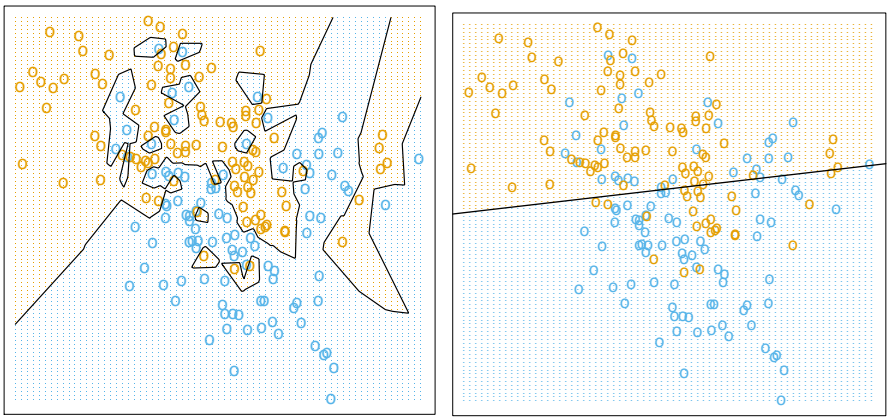


creditcardscrop.pdf (ISL, Figure 4.1) [The problem of classification. We are given data points, each belonging to one of two classes: orange crosses represent people who defaulted on their credit cards, and blue circles represent those who didn't. Then we are given additional points whose class is unknown, and we are asked to predict what class each new point is in. Given the credit card balance and annual income of new applicants, predict whether they will default on their debt.]



[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.]
[Those are two different *models* for the nature of this data.]

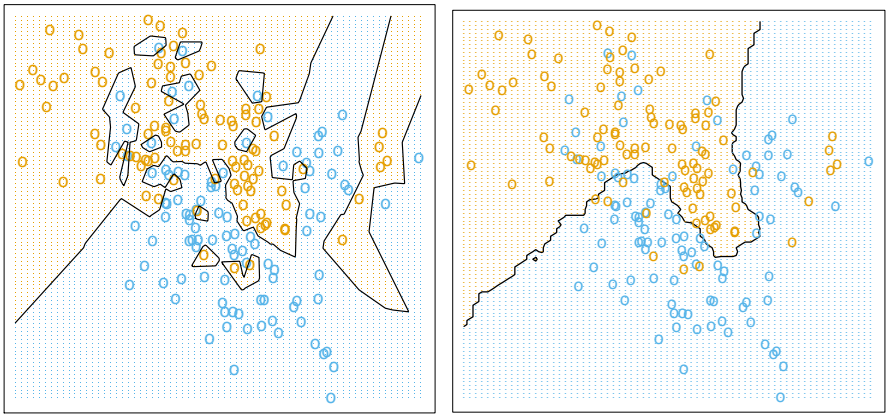
[We'll learn some ways to compute linear decision boundaries in the next several lectures. But for now, let's compare these two methods.]



classnear.pdf, classlinear.pdf (ESL, Figures 2.3 & 2.1) [Two examples of classifiers for the same data: a nearest neighbor classifier (left) and a linear classifier (right). The decision boundaries are in black.]

[At left we have a *nearest neighbor classifier*, which classifies a new point by finding the nearest point in the training 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. At right, the linear decision boundary—the black line—is explicitly computed by the classifier. At left, the decision boundary is *not* computed; the classifier just takes a new point and computes the distances to all the training points.]

[The neighbor classifier at left has a big advantage: it classifies all the training data correctly, whereas the linear classifier does not. But the linear classifier has an advantage too. Somebody please tell me what.]

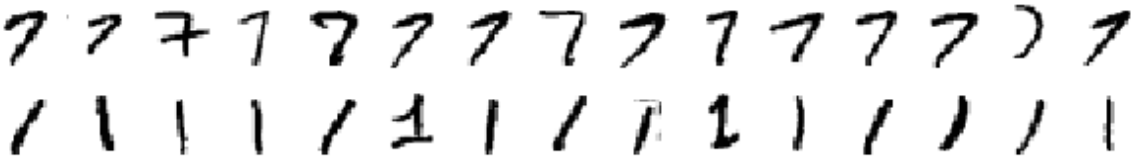


classnear.pdf, classnear15.pdf (ESL, Figures 2.3 & 2.2) [A 1-nearest neighbor classifier and a 15-nearest neighbor classifier.

[The *15-nearest neighbor classifier* classifies a new point by looking at its 15 nearest neighbors and letting them vote for the correct class.]

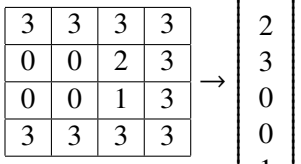
[The left figure is an example of what's called overfitting. In the left figure, observe how intricate the decision boundary is that separates the positive examples from the negative examples. It's a bit too intricate to reflect reality. In the right figure, the decision boundary is smoother. Intuitively,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

Classifying Digits



sevensones.pdf [In the MNIST digit recognition problem, we are given handwritten digits, and we are asked to learn to distinguish them. See Homework 1.]

Express these images as vectors



$$\begin{bmatrix} 3 \\ 3 \\ 3 \\ 3 \\ 0 \\ 0 \\ 2 \\ 3 \\ 0 \\ 0 \\ 1 \\ 3 \\ 3 \\ 3 \\ 3 \\ 3 \end{bmatrix}$$

Images are points in 16-dimensional space. Linear decision boundary is a hyperplane.

TRAIN, VALIDATE, TEST

How we classify:

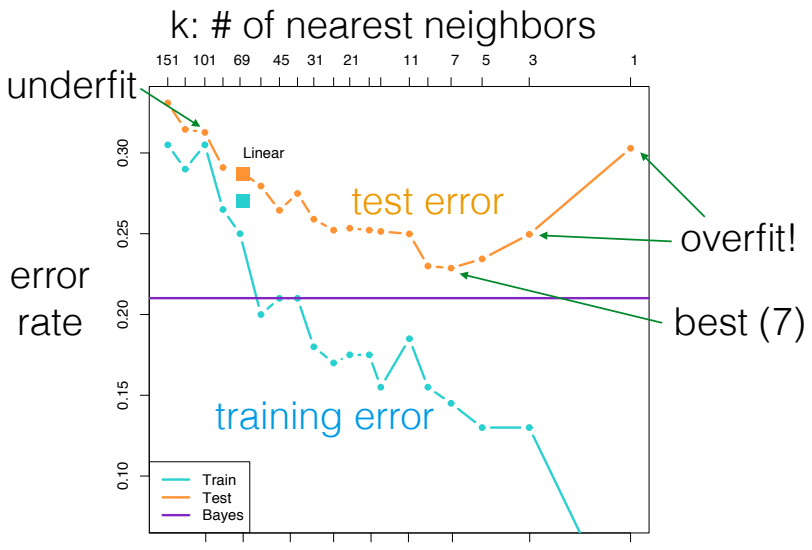
- We are given labeled data—sample points with class labels.
- Hold back a subset of the labeled points, called the validation set. Maybe 20%. The other 80% is the training set. [Warning: the term *training data* is not used consistently. Often "training data" refers to *all* the labeled data. You have to judge from context.]
- Train one or more classifiers: they learn to distinguish 7 from not 7. Use training set to learn model weights. Do NOT use validation set to train!!!
- Usually, train multiple learning algorithms, or one algorithm with multiple hyperparameter settings, or both [using the same training set for each].
- Validate the trained classifiers on the validation set. Choose classifier/hyperparameters with lowest validation error. Called validation. [When we do validation, we are not learning any more. We are checking what classes our trained classifiers assign to our validation set, and counting how often they're right. We use this to judge our models—not how well they remember the training set labels.]
- Optional: Test the best classifier on a test set of NEW data. Final evaluation. Typically you do NOT have the labels. [But somebody else might have them, and assign you a score!]

[When I underline a word or phrase, that usually means it's a definition. My advice to you is to memorize the definitions I cover in class.]

3 kinds of error:

- Training error: fraction of training set not classified correctly. [This is zero with the 1-nearest neighbor classifier, but nonzero with the 15-nearest neighbor and linear classifiers. But that doesn't mean the 1-nearest neighbor classifier is always better. Remember that you cannot include the validation data in this calculation, even if somebody calls it "training data."]
- Validation error: fraction of validation set misclassified. Use this to choose classifier/hyperparameters. [You didn't use the validation set to train, so even the 1-nearest neighbor classifier can classify these points wrong. Validation error is almost always higher than training error.]
- Test error: fraction of test set misclassified. Used to evaluate **you**.

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



overfitlabeled.pdf (modified from ESL, Figure 2.4)

- overfitting: when the validation/test error deteriorates because the classifier becomes too sensitive to outliers or other spurious patterns.
- underfitting: when the validation/test error deteriorates because the classifier is not flexible enough to fit patterns.
- outliers: points with atypical labels (e.g., rich borrower who defaulted anyway). Increase risk of overfitting.

[In machine learning, the goal is to create a classifier that generalizes to new examples we haven't seen yet. Overfitting and underfitting are both counterproductive to that goal. So we're always seeking a compromise: we want decision boundaries that make fine distinctions without being downright superstitious.]

Kaggle.com:

- Runs ML competitions, including our HWs
- We may use 2 test sets:
  public set: test scores available during competition
  private set: test scores available after competition
  [The private test set prevents you from "cheating" by throwing lots of models at the public test set until you find a lucky one.]

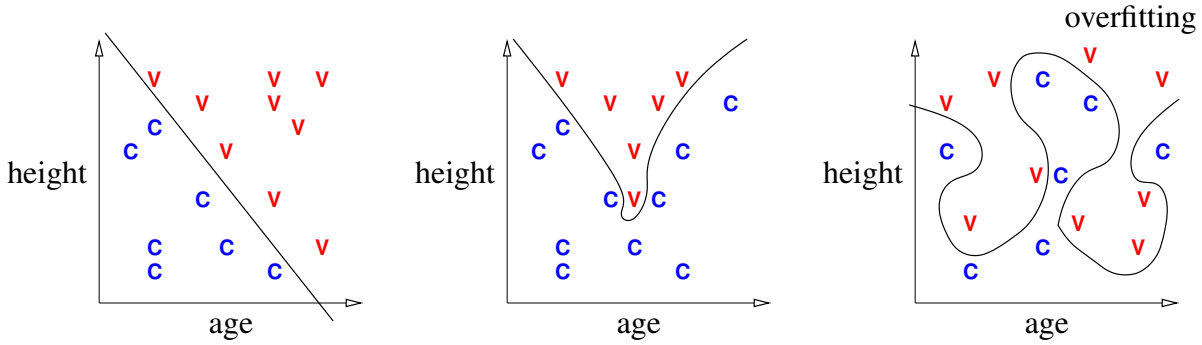
2 Linear Classifiers, the Centroid Method, and Perceptrons

CLASSIFIERS

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

Example: Observations are ice cream lovers
Features are height & age ($d = 2$)
Some are in class "chocolate," some prefer vanilla
Goal: Predict preferred flavor based on their height & age.

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



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

[We draw these lines/curves separating C's from V'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.

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

[A reminder that underlined phrases are definitions, worth memorizing.]

Some (not all) classifiers work by computing a

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

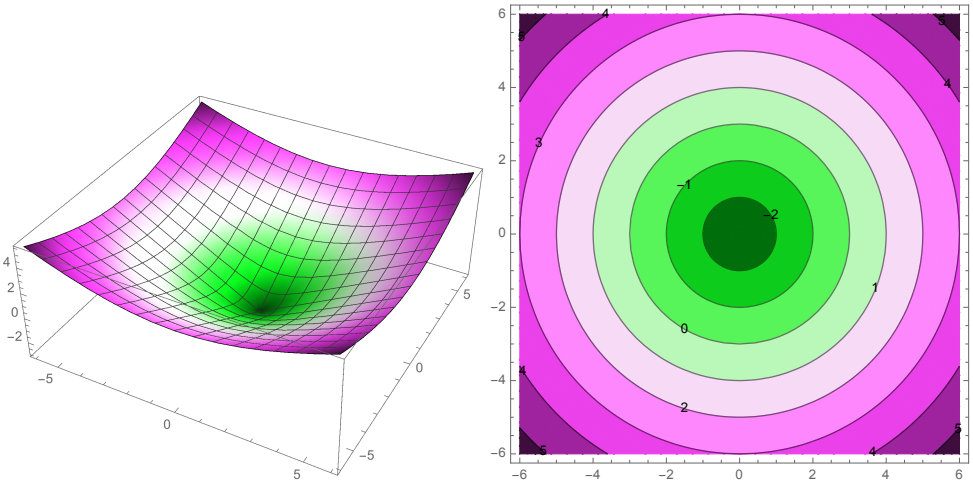
$f(x) > 0$ if $x \in$ class C;
$f(x) \leq 0$ if $x \notin$ class C.

Aka predictor 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 decision function is zero.]
Usually, this set is a $(d-1)$-dimensional surface in $\mathbb{R}^d$.

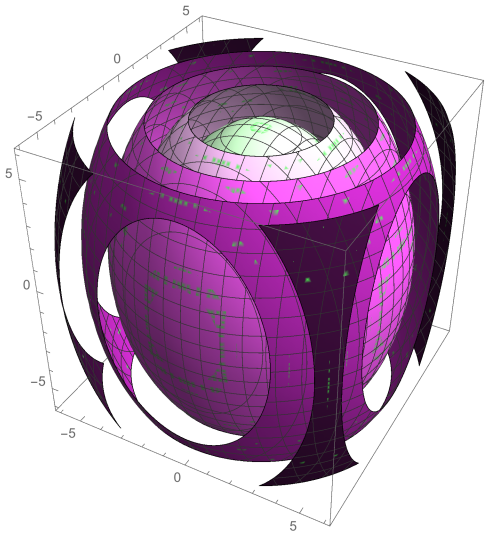
$\{x : f(x) = 0\}$ is also called an isosurface of f for the isovalue 0.

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



radiusplot.pdf, radiusiso.pdf [3D plot and isocontour plot of the cone] $f(x, y) = \sqrt{x^2 + y^2} - 3$.

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



radiusiso3d.pdf

[One of these spheres could be the decision boundary.]

linear classifier: The decision boundary is a line/plane.
Usually uses a linear decision function.

Linear Classifier Math

[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 = real scalar	α
Some integers:	n = # of sample points
	d = # of features (per point)
	= dimension of sample points
	i j k = indices
function (often scalar)	$f(\,)$, $s(\,)$, ...

Euclidean inner product (aka dot product): $x \cdot y = x_1y_1 + x_2y_2 + ... + x_dy_d$

also written $x^\top y$

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

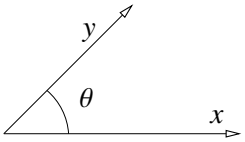
Euclidean norm: $\|x\| = \sqrt{x \cdot x} = \sqrt{x_1^2 + x_2^2 + ... + x_d^2}$

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

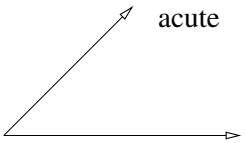
Given a vector $x \neq 0$, $\frac{x}{\|x\|}$ is a unit vector (length 1).

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

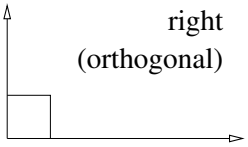
Use dot products to compute angles:



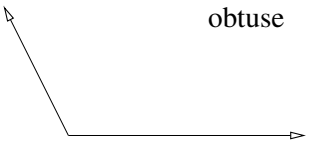
$$\cos\theta = \frac{x \cdot y}{\|x\|\,\|y\|} = \underbrace{\frac{x}{\|x\|}}_{\text{length 1}} \cdot \underbrace{\frac{y}{\|y\|}}_{\text{length 1}}$$



$x \cdot y > 0$



$x \cdot y = 0$



$x \cdot y < 0$

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

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

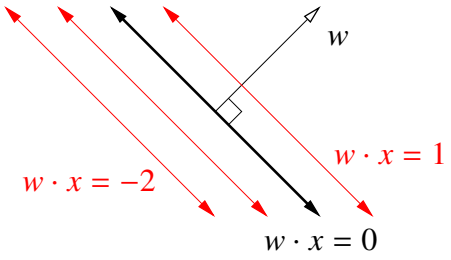
The set H is called a hyperplane. (A line in 2D, a plane in 3D.)

[A hyperplane is what you get when you generalize the idea of a plane to higher dimensions. The three most important things to understand about a hyperplane is (1) it has dimension $d-1$ and it cuts the d-dimensional space into two halves; (2) it's flat; and (3) it's infinite.]

Theorem: Let x, y be 2 points that lie on H. Then $w \cdot (y - x) = 0$.

Proof: $w \cdot (y - x) = -\alpha - (-\alpha) = 0$. [Therefore, w is orthogonal to any line segment that lies on H.]

w is called the normal vector of H,
because (as the theorem shows) w is normal (perpendicular) to H.
[I.e., w is perpendicular to every line on H.]



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

If w is a unit vector, then $f(x) = w \cdot x + \alpha$ is the signed distance from x to H. [See Discussion 1.]
I.e., positive on w's 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 the signed distance times some real.
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 parameters or regression coefficients).

[That's why we call the vector w; "w" stands for "weights."]

The training points are linearly separable if there exists a hyperplane that correctly classifies all the training points.

[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 and some that do a decent job with non-separable data. Obviously, if your data are 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.]

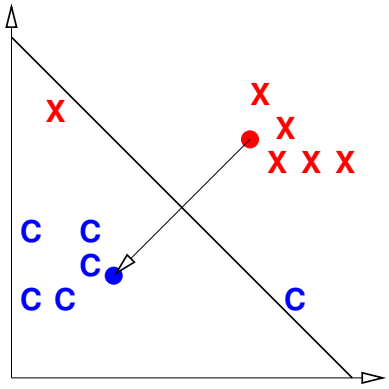
A Simple Classifier

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

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



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

[In this example, there's clearly a linear classifier that classifies every training point correctly, and the centroid method isn't it.
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 training points, where the centroid method misclassifies every training point but two.]

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

[We can sometimes improve this classifier by adjusting the scalar term α to minimize the number of misclassified points. Then the hyperplane has the same normal vector, but a different position.]

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 gradient descent is?
How many of you know what the backpropagation algorithm is?
How many of you know what a linear program is?
How many of you know what a quadratic program is?

We're going to learn what these things are. As machine learning people, we will be heavy users of optimization methods. Unfortunately, I won't have time to teach you *algorithms* for many optimization problems, but we'll learn a few. To learn more, take EECS 127.]

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 in a matrix X.]

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

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. [That's how we train a perceptron classifier.]

Define the loss function

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

[Here, $\hat{y}$ is the classifier's prediction, and y_i is the correct answer, called the label.]

If $\hat{y}$ has the same sign as y_i, the loss function is zero (happiness).
But if $\hat{y}$ has the wrong sign, the loss function is positive.

[For each training 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 your classifier is.]

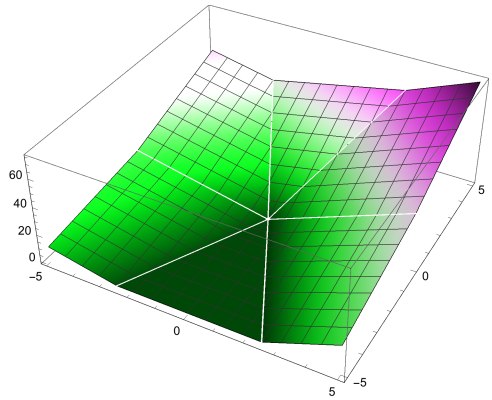
Define risk function (aka objective function or cost function)

$$\begin{aligned} R(w) &= \frac{1}{n} \sum_{i=1}^{n} L(X_i \cdot w, y_i) \\ &= \frac{1}{n} \sum_{i \in V} -y_i X_i \cdot w \qquad \text{where } V \text{ is the set of indices } i \text{ for which } y_i X_i \cdot w < 0. \end{aligned}$$

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 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 decision fn $f(x) = w \cdot x$ (for simplicity, no α)
- decision boundary $\{x : f(x) = 0\}$ (a hyperplane through the origin)
- sample points $X_1, X_2, \ldots, X_n \in \mathbb{R}^d$; class labels $y_1, \ldots, y_n = \pm 1$
- goal: find weights w such that $y_i X_i \cdot w \geq 0$
- goal, revised: find w that minimizes $R(w) = \sum_{i \in V} -y_i X_i \cdot w$ [risk function]

  where $V = \{i : y_i X_i \cdot w < 0\}$.

[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:	$\{z : x \cdot z = 0\}$

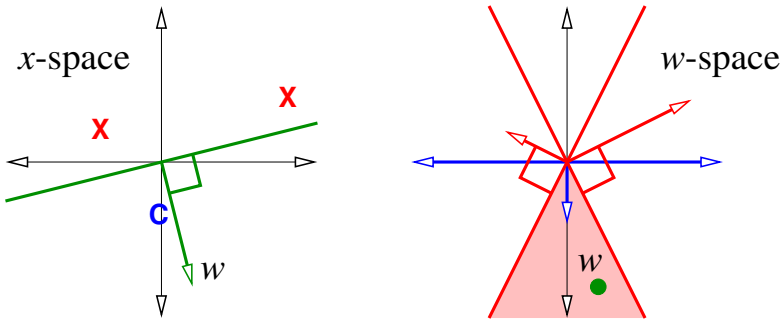
Point x lies on hyperplane $\{z : w \cdot z = 0\} \Leftrightarrow w \cdot x = 0 \Leftrightarrow$ point w lies on hyperplane $\{z : x \cdot z = 0\}$ in w-space.

[So a hyperplane transforms to a point that represents its normal vector. And a sample point transforms to the hyperplane whose normal vector is the sample point.]

[In this algorithm, the transformations happen to be 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 decision boundaries we use this semester.]

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

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

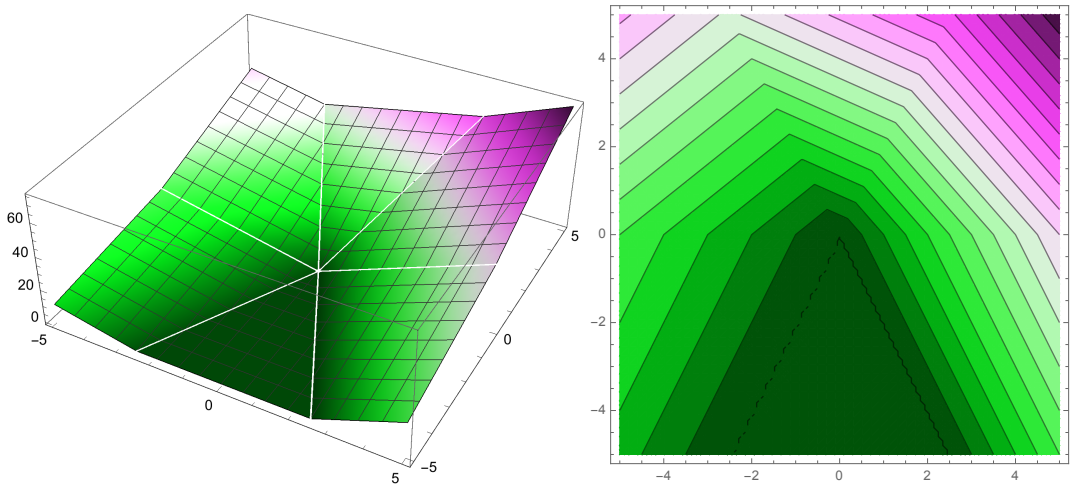


[Draw this by hand. xwspace.pdf]
[Observe that the x-space sample points are the normal vectors for the w-space lines. We can choose w to be anywhere in the shaded region.]

[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 (marked by an 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.]

[Again, what have we accomplished? We have switched from the problem of finding a hyperplane in x-space to the problem of finding a point in w-space. That's a better fit to how we think about optimization algorithms.]

[Let's take a look at the risk function these three sample points create.]



riskplot.pdf, riskiso.pdf [Plot & isocontours of risk $R(w)$. Note how R's creases match the lines in the w-space drawn 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.

[Draw the typical steps of gradient descent on the plot of 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]

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

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

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

$\epsilon > 0$ is the step size aka learning rate, chosen empirically. [Best choice depends on input problem!]

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

[Stochastic gradient descent is quite popular and we'll see it several times more this semester, especially for neural networks. However, 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.]

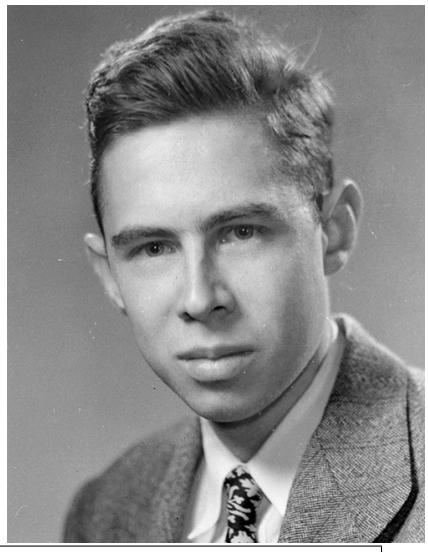
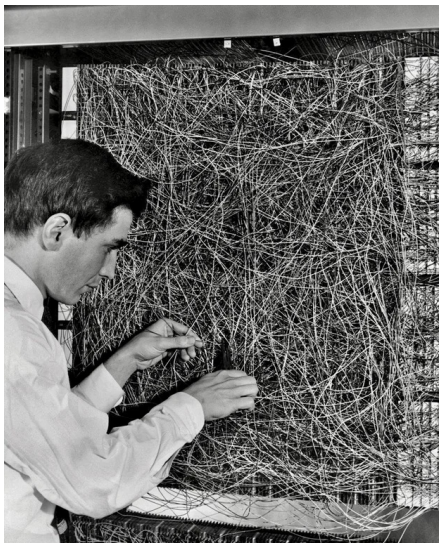
What if separating hyperplane doesn't pass through origin?
Add a fictitious dimension. Decision fn is

$$f(x) = w \cdot x + \alpha = [w_1 \;\; w_2 \;\; \alpha] \cdot \begin{bmatrix} x_1 \\ x_2 \\ 1 \end{bmatrix}$$

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

Run perceptron algorithm in $(d+1)$-dimensional space. [We are simulating a general hyperplane in d dimensions by using a hyperplane through the origin in $d+1$ dimensions.]

[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.]

frankrosenblatt.jpg, perceptron.jpg [Frank Rosenblatt (from *Cornell Chronicle*) and his 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.]

[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.]

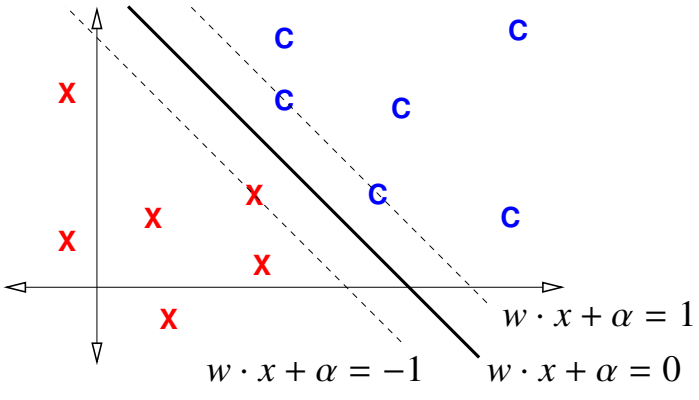
[We're not going to prove this, because perceptrons are obsolete.]

[Although the step size/learning rate ϵ 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 a 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 training point. What if we make the margin as wide as possible?



[Draw this by hand. maxmargin.pdf]

We enforce the constraints

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

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

Recall: if $\|w\| = 1$, signed distance from hyperplane to X_i is $w \cdot X_i + \alpha$.
Otherwise, it's $\frac{w}{\|w\|} \cdot X_i + \frac{\alpha}{\|w\|}$. [We've normalized the expression to get a unit weight vector.]

Hence the margin is $\min_i \frac{1}{\|w\|} \underbrace{|w \cdot X_i + \alpha|}_{\geq 1} \geq \frac{1}{\|w\|}$. [We get the inequality by substituting the constraints.]

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! [If the points are linearly separable; otherwise, it has no solution.]

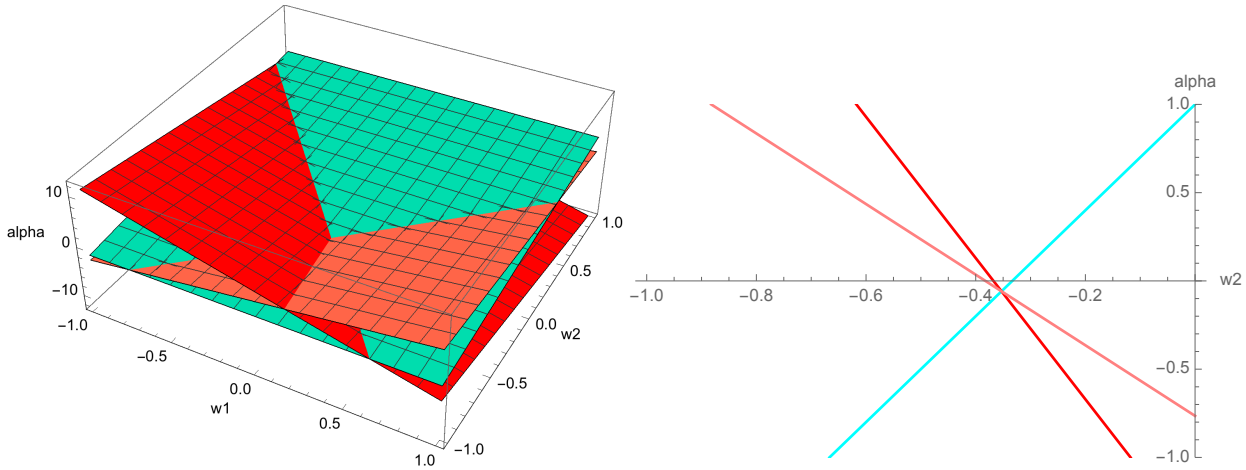
[A reason we use $\|w\|^2$ as an objective function, instead of $\|w\|$, is that the length function $\|w\|$ is not smooth at $w = 0$, whereas $\|w\|^2$ is smooth everywhere. This makes optimization easier.]

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 kernels.]

At the optimal solution, the margin is exactly $\frac{1}{\|w\|}$. [Because at least one constraint holds with equality.]
There is a slab of width $\frac{2}{\|w\|}$ containing no sample points [with the hyperplane running along its middle].

[Let's see what these constraints look like in weight space.]



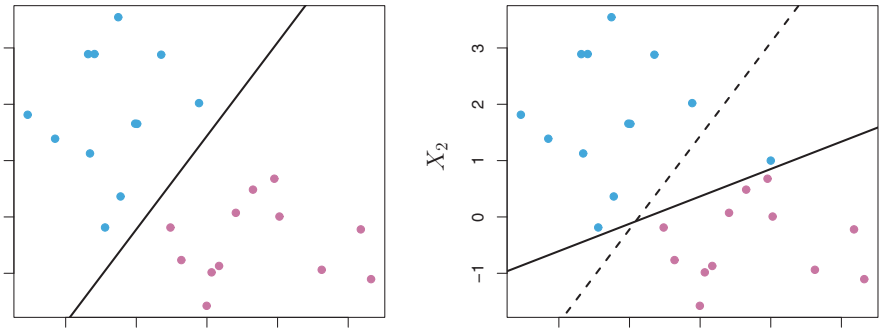
weight3d.pdf, weightcross.pdf [This is an example of what the linear constraints look like in the 3D weight space (w_1, w_2, α) for the SVM we've been studying with three training points. The SVM is looking for the point nearest the α-axis that lies above the blue plane (representing an in-class training 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.]

[Like the perceptron algorithm, a hard-margin SVM works only with linearly separable point sets. We'll fix that in the next lecture.]

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 hard-margin SVM decision boundary a lot.]

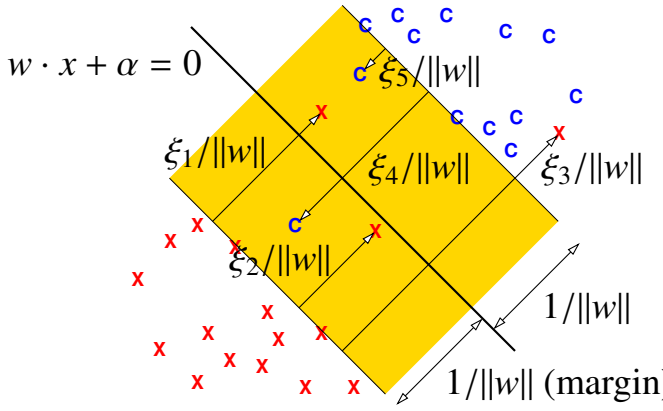
Idea: Allow some points to violate the margin, with slack variables.
Modified constraint for point i:

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

[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.]



slacker+.pdf [A margin where some points have slack.]

Re-define "margin" to be $1/||w||$. [For soft-margin SVMs, the margin is no longer the distance from the decision boundary to the nearest training point; instead, it's $1/||w||$.]

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 $i \in [1, n]$

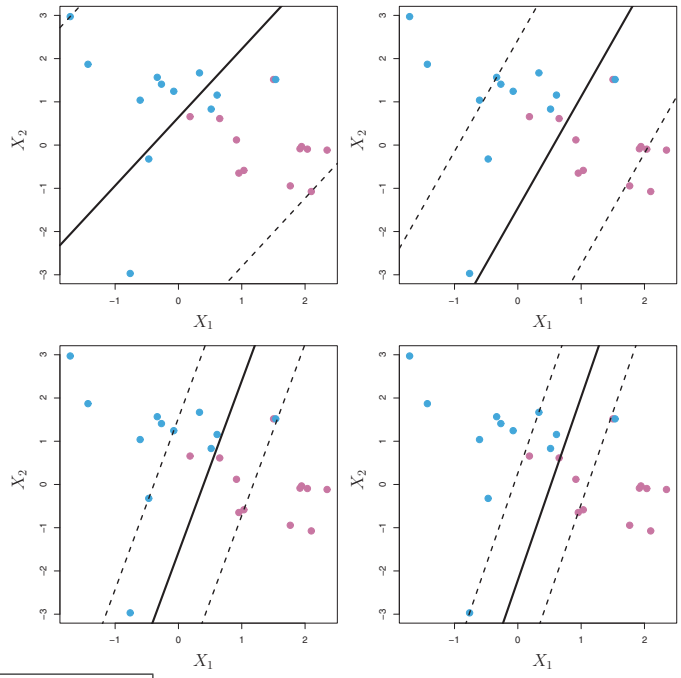
. . . a quadratic program in $d + n + 1$ dimensions and $2n$ constraints.
[It's a quadratic program because its objective function is quadratic and its constraints are linear inequalities.]

$C > 0$ 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"—though it can overfit.]

Use validation to choose C.



svmC.pdf (ISL, Figure 9.7) [Examples of how the slab varies with C. Smallest C at upper left; largest C at 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 wide 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 wider margin. If C is big, it means we're cheap and we won't pay much for violations, even though we'll suffer 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 (aka basis functions) that lift points into a higher-dimensional space.
High-d linear classifier → low-d nonlinear classifier.

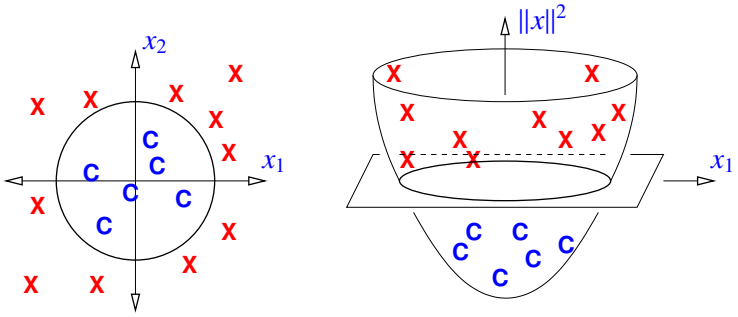
[Added features work with all classifiers—not only linear classifiers like perceptrons and SVMs, but also classifiers that are not linear.]

Example 1: The parabolic lifting map

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

[We've added one new feature, $\|x\|^2$. Even though the new feature is just a function of other input features, it gives our linear classifier more power. Now an SVM can have spheres as decision boundaries.]

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 an ∞-radius hypersphere = hyperplane.)

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

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

Hence points inside sphere ↔ lifted points underneath hyperplane in Φ-space.
[The implication works in both directions.]

[Hyperspheres include hyperplanes as a special, degenerate case. A hyperplane is essentially a hypersphere with infinite radius. So hypersphere decision boundaries can do everything hyperplane decision boundaries can do, plus a lot more. With the parabolic lifting map, if you pick a hyperplane in Φ-space that is vertical, you get a hyperplane in x-space.]

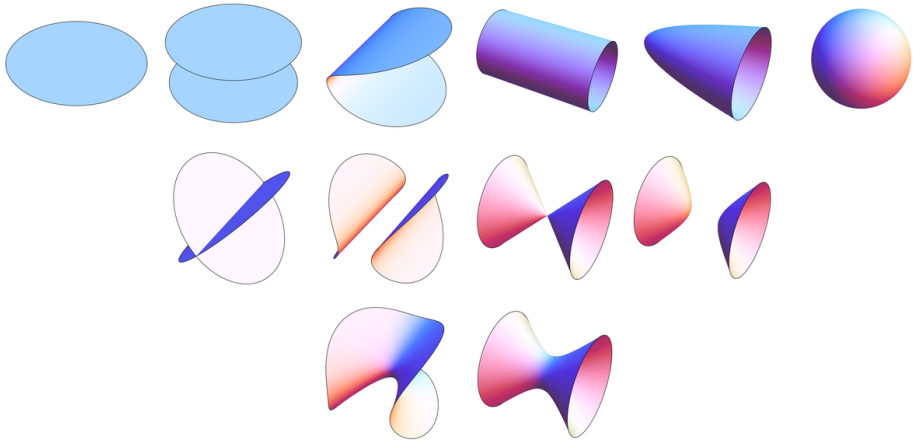
Example 2: Ellipsoid/hyperboloid/paraboloid decision boundaries

[Draw 2D examples of ellipse & hyperbola.]

In 3D, these have the formula

$$Ax_1^2 + Bx_2^2 + Cx_3^2 + Dx_1x_2 + Ex_2x_3 + Fx_3x_1 + Gx_1 + Hx_2 + Ix_3 + \alpha = 0$$

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



quadrics.png (courtesy Rahul Narain) [Quadrics in 3D.]

[If we add all the quadratic monomials as features, our decision boundaries can be arbitrary ellipsoids, hyperboloids, paraboloids, and more.]

$$\Phi(x) = [x_1^2 \quad x_2^2 \quad x_3^2 \quad x_1x_2 \quad x_2x_3 \quad x_3x_1 \quad x_1 \quad x_2 \quad x_3]^\top$$

[For perceptron or regression, add a 1 at end. For SVM, the 1 is built-in.]

Decision function is $\underbrace{[A \quad B \quad C \quad D \quad E \quad F \quad G \quad H \quad I]}_{w^\top} \cdot \Phi(x) + \alpha$

[Now, our decision function can be any degree-2 polynomial. Each component of Φ is also called a basis function, as it is a function of x and the decision function is a linear combination of basis functions.]

Isosurface defined by this equation is called a quadric.
A linear decision boundary in Φ-space imposes a quadric decision boundary in x-space.
[The word *quadric* just means an isosurface of a degree-2 polynomial. In the special case of two dimensions, it's also known as a conic section. Our decision boundary can be an arbitrary ellipsoid, hyperboloid, paraboloid, cylinder, etc.]

[When d is large, there are order-d^2 cross-terms in Φ-space! So we are adding a lot of new features. This will impose a serious computational cost on a classifier like a support vector machine. But it might be worth it to find good classifiers for data that aren't linearly separable.]

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

[For perceptron or regression, add 1 for the fictitious dimension.]

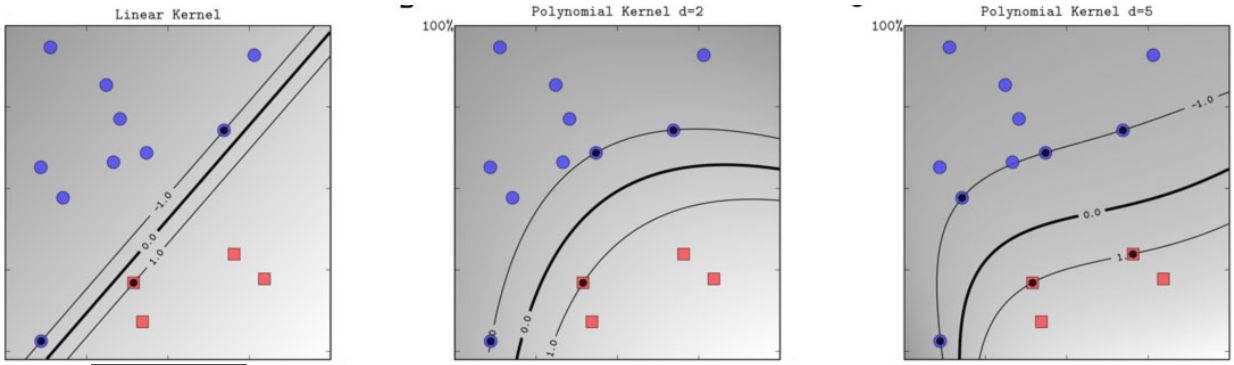
[If all these extra features make the classifier overfit or make it too slow, you can leave out the cross-terms and include only quadratic terms like x_1^2, x_2^2, etc. Then the number of added features is linear in d, not quadratic in d. If you do that, your decision boundaries can be axis-aligned ellipsoids and axis-aligned hyperboloids, but they can't be rotated in arbitrary ways.]

Example 3: Decision fn is degree-p polynomial

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

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

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

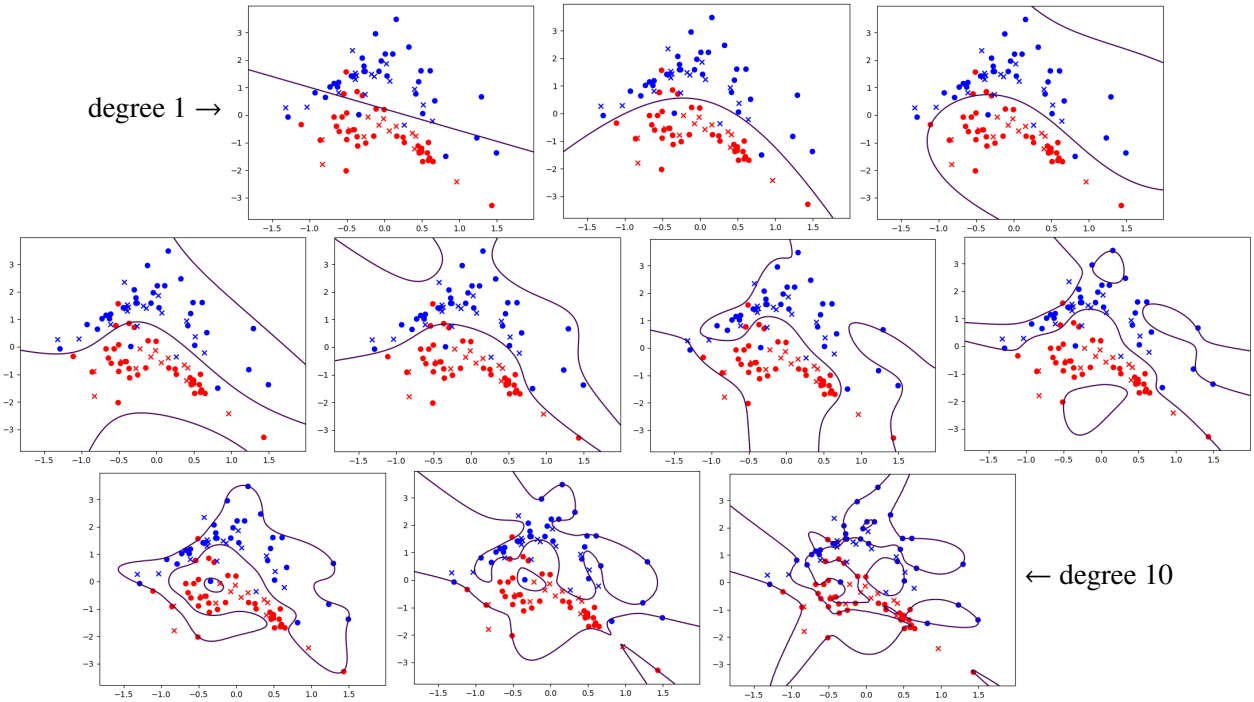


degree5.pdf [Hard-margin SVMs with degree 1/2/5 decision functions. Observe that the margin tends to get wider as the degree increases.]

[Increasing the degree like this accomplishes two things.

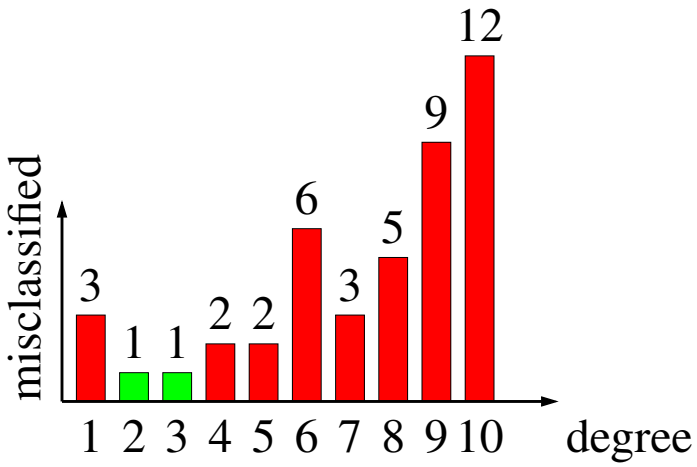
- First, the data might become linearly separable when you lift them to a high enough degree, even if the original data are not linearly separable.
- Second, raising the degree can widen the margin, so you might get a more robust decision boundary that generalizes better to test data.]

degree 1 →



← degree 10

d1.pdf, d2.pdf, . . . , d10.pdf [Decision boundaries found by linear regression with polynomial features of maximum degrees from 1 through 10. Circles are training points; X's are validation points. (Implementation courtesy Josh Levine.)]



misclass.pdf [Misclassified validation points for the linear regressions depicted above, with polynomial features of degrees 1 through 10.]

[Here we see a U-shaped curve, as we do with the regularization parameter C in soft-margin SVMs. This example obtains best results with degree 2 or 3 polynomials. A linear classifier underfits, whereas classifiers of degree 4 or greater overfit; generalization gets worse as the decision boundary becomes too flexible. The degree is an example of a *hyperparameter* that can be optimized by validation.]

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

[With polynomials, 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 decision functions, then each lifted feature vector has a length of roughly 4 million, and your learning algorithm will take approximately forever to run.]

[However, there is 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 quickly.]

[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 a handwriting recognition algorithm.]

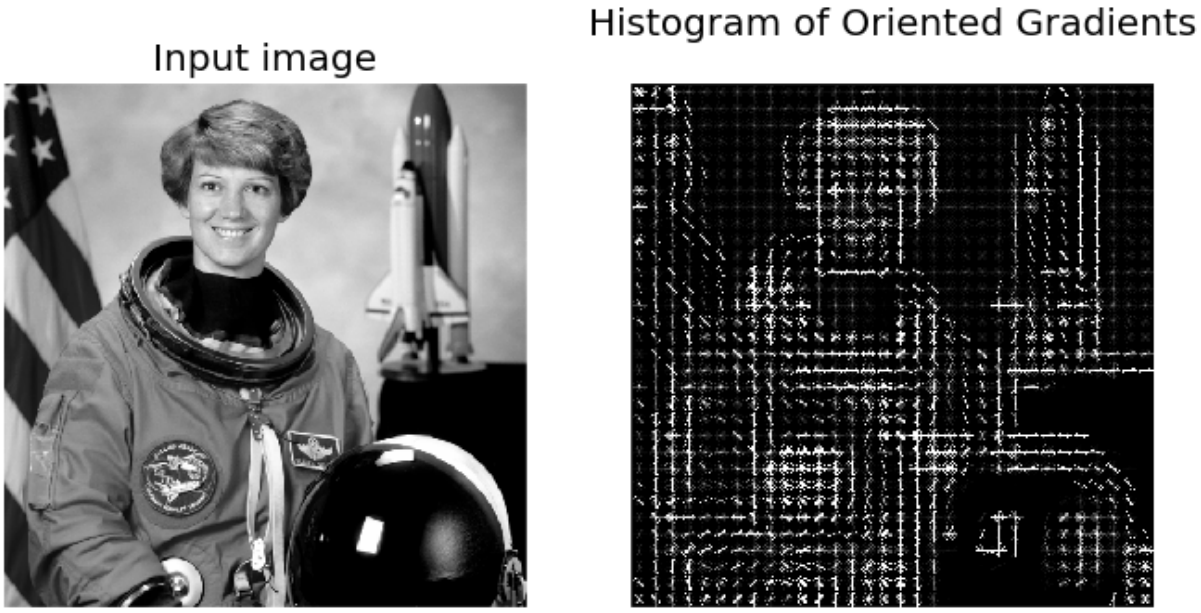
Example 5: Edge detection

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 Derivative" 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 future homeworks 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 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.:
– decision 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 will talk about optimization algorithms, but usually data analysts use optimization codes that are faster and more robust than what they would write themselves.]

[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 have a simple structure. 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 your model from a linear classifier to a neural net, your optimization problem changes, and your optimization algorithm 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

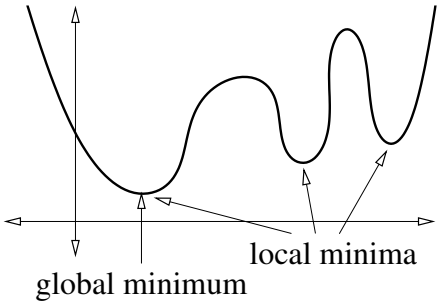
[I want to familiarize you with some types of optimization problems that can be solved reliably and efficiently, and the names of some of the optimization algorithms used to solve them. An important skill for you to develop is to be able to go from an application to a well-defined optimization problem.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

Unconstrained

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

f is smooth if its gradient is continuous too.

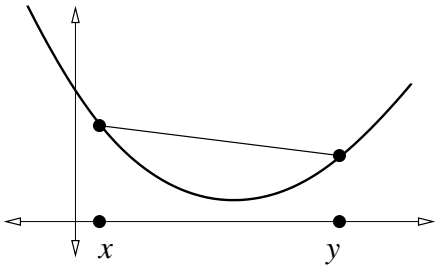
A global minimum of f is a value w such that $f(w) \leq f(v)$ for every v.
A local minimum " " " " " " " " " "
for every v in a tiny ball centered at w.
[In other words, you cannot walk downhill from w.]



[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 segment connecting $(x, f(x))$ to $(y, f(y))$ does not go below $f(\cdot)$.



[Draw this by hand. convex.pdf]

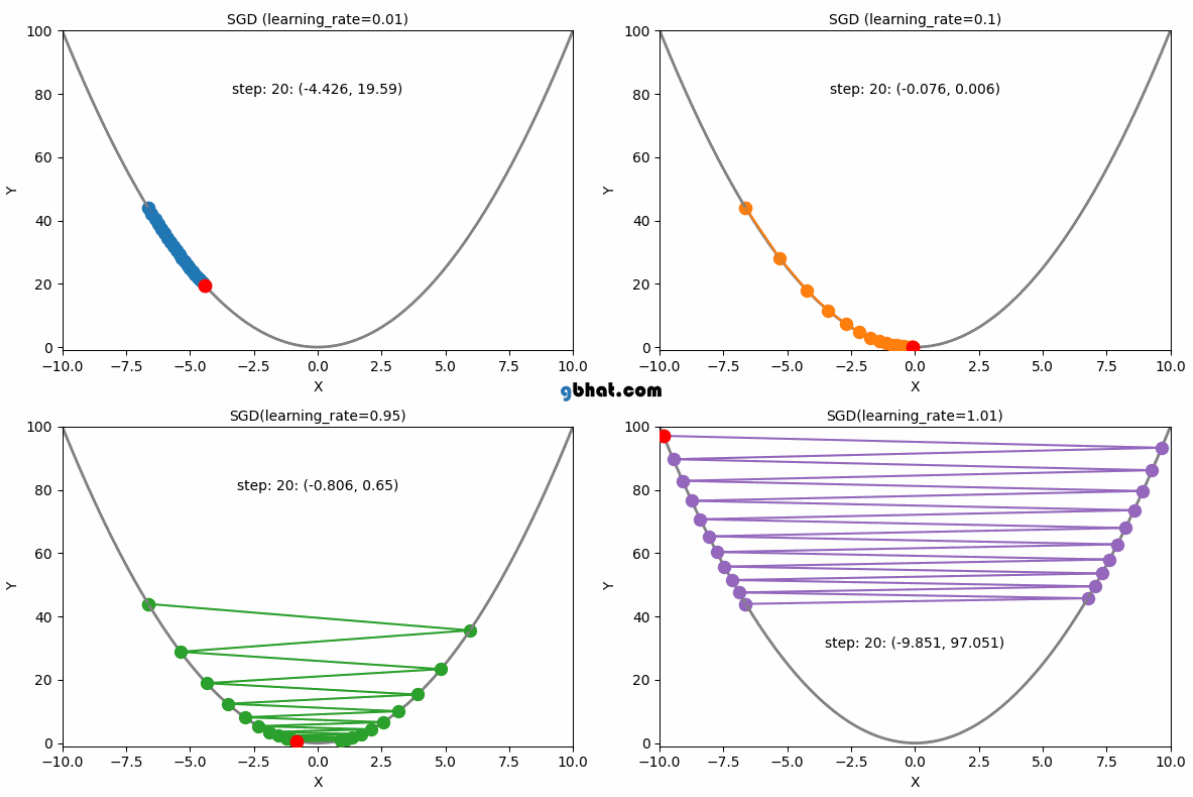
Formally: for every $x, y \in \mathbb{R}^d$ and $\beta \in [0, 1]$, $f(x + \beta(y - x)) \leq f(x) + \beta(f(y) - f(x))$.
E.g., perceptron risk fn is convex and nonsmooth.

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

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 is in the last category.]
[In the last two cases, if you walk downhill, you eventually reach a global minimum.]

Gradient descent: repeat $w \leftarrow w - \epsilon \nabla f(w)$



learningrates20.gif (Gajanan Bhat, gbhat.com) [Gradient descent with different learning rates ϵ. Top left: painfully small. Top right: reasonable, but still smaller than ideal. Bottom left: reasonable, but larger than ideal. Bottom right: too large; diverges. This is an animated GIF; see https://gbhat.com/machine_learning/gradient_descent_learning_rates.html .]

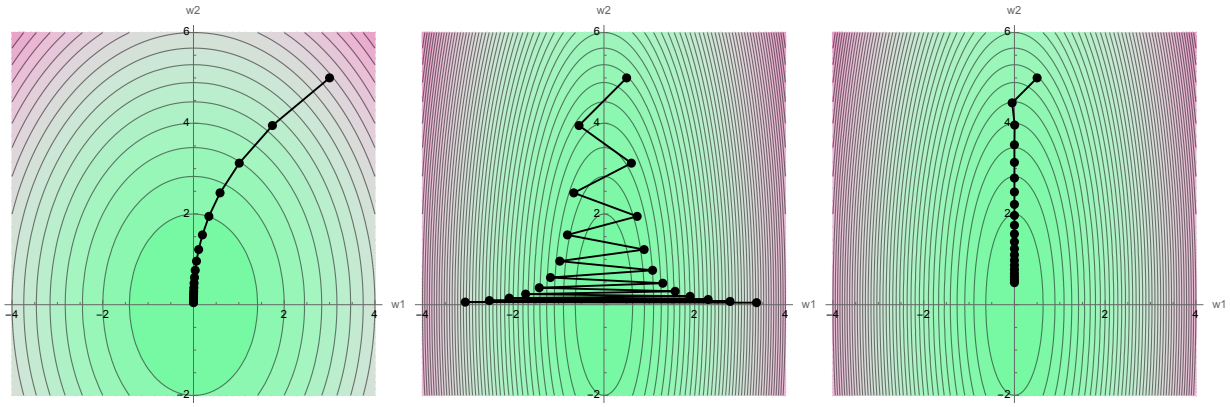
- Fails/diverges if ϵ too large.
- Slow if ϵ too small.
- ϵ often optimized by trial & error [for slow learners like neural networks].

[The best value of ϵ is hard to guess. One common technique for dealing with divergence is to check whether a step of gradient descent increases the function value rather than decreasing it; if so, reduce the step size.]

[That's a simple example of what's called an *adaptive learning rate* or a *learning rate schedule*. These adaptations become even more important when you do stochastic gradient descent or when you optimize non-convex, very twisty objective functions. We'll revisit the idea when we learn neural networks.]

[One interesting aspect of gradient descent that these figures illustrate is that it usually never reaches the exact local minimum. Instead, it gets closer and closer forever, but never exactly reaches the true minimum. We call this behavior "convergence." The last question of Homework 2 will give you some understanding of why convergence happens under the right conditions.]

[When we have a feature space with more than one dimension, another problem arises, which is that the learning rate that's good for one direction might be terrible in another direction. Consider the three examples of gradient descent below.]



goodcondition.pdf, illcondition105.pdf, illcondition055.pdf [Left: 20 iterations of gradient descent on a well-conditioned quadratic function, $f(w) = 2w_1^2 + w_2^2$, with a modest step size $\epsilon = 0.105$. Center: 20 iterations on an ill-conditioned function, $f(w) = 10w_1^2 + w_2^2$; the same step size is now too large. Right: after reducing the step size to $\epsilon = 0.055$, we have convergence again but we aren't approaching the minimum nearly as quickly.]

[The step size that works for the left example is too large for the center example; it diverges in the w_1-direction. At right, we reduce the step size and obtain convergence. But now convergence is slow in the w_2-direction.]

High ellipticity of the contours, a.k.a. ill-conditioning of the Hessian, means no learning rate is good in all directions.

[The Hessian matrix is said to be ill-conditioned if its largest eigenvalue is much larger than its smallest eigenvalue. Ill-conditioning can be a problem even for simple methods like linear regression, making it harder to solve the problem. In response to these observations, there are adaptive learning rate algorithms that explicitly choose different learning rates for different weights. Famous examples are Adam and RMSprop.]

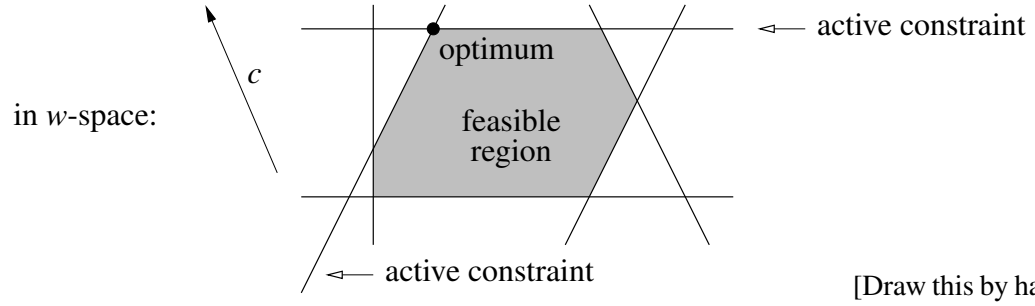
[There are many applications where you don't have a convex objective function. Then gradient descent usually can find a local minimum, but not necessarily a global minimum. And often there is no guarantee that the local minimum you find will be nearly as good as the global minimum. Nevertheless, gradient descent is used for a lot of nonconvex machine learning problems too. For example, neural networks try to optimize an objective function that has *lots* of local minima. But stochastic gradient descent is still the algorithm of choice for training neural nets. We'll talk more later in the semester about why.]

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:
$A_i \cdot w \leq b_i, \qquad i \in [1, n]$



[Draw this by hand. linprog.pdf]

The set of points w 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.

[What is a polytope? 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 training points that touch or violate the slab, and these points are 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 satisfies $y_i(w \cdot X_i + \alpha) \geq 1 \quad$ for all $i \in [1, n]$

[This is the problem of finding a feasible point. This problem can be cast as a slightly different linear program that uses an objective function to make all the inequalities be satisfied *strictly* if that's possible.]

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

[The most famous algorithm for linear programming is the simplex algorithm, invented by George Dantzig in 1947. The simplex algorithm is indisputably one of the most important and useful algorithms of the 20th century. It walks along edges of the feasible region, traveling from vertex to vertex until it finds an optimum.]

[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 mathematics 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

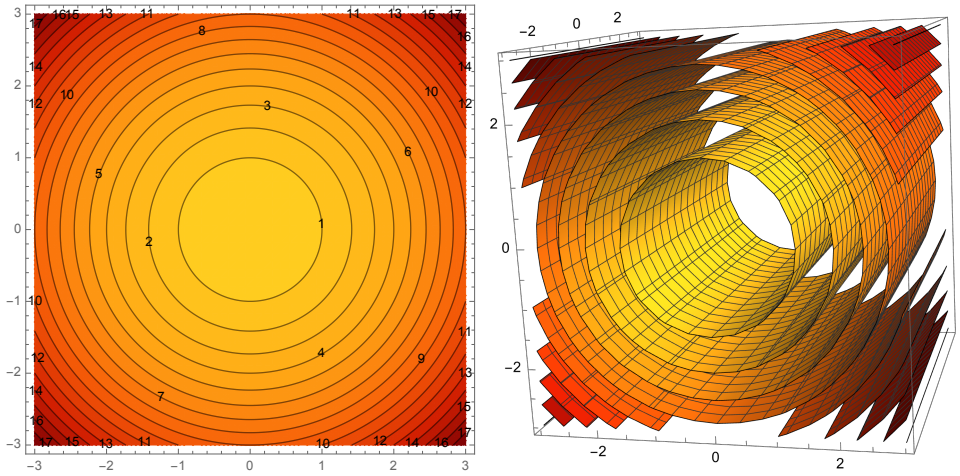
Quadratic, convex objective fn + linear inequality constraints.

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 semidefinite matrix.

[A matrix is positive semidefinite if $w^\top Q w \geq 0$ for all w.]

Example: Find maximum margin classifier.



quadratic.pdf, quadratic3D.pdf [Left: A hard-margin SVM minimizes the objective function $w_1^2 + w_2^2$. Right: There is also an α-axis, so the isosurfaces of the objective function are really cylinders. On the left isocontours, draw two polygons—one with one active constraint, and one with two—and show the constrained minimum for each polygon. "In a hard-margin SVM, we are looking for the point in this polygon that's closest to the α-axis."]

[If Q is positive definite, a quadratic program has just one unique local minimum, which is therefore the global minimum. But in a support vector machine, Q is not definite; it is only positive semidefinite, because the bias term α is a weight but it does not influence the objective function. Sometimes positive semidefinite quadratic programs have multiple solutions, but SVMs are a special case where there is only one unique minimum. By the way, if Q is indefinite, then f is not convex, the minimum is not always unique, and quadratic programming is NP-hard. But we won't need that kind of quadratic program in this class.]

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, "SVC" in scikit)
- Coordinate descent (used in LIBLINEAR, "LinearSVC" in scikit)

Numerical optimization @ Berkeley: EECS 127/227AT/227BT/227C.

6 Decision Theory; Generative and Discriminative Models

DECISION THEORY aka Risk Minimization

[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.]

[One aspect of probabilistic data is that sometimes a point in feature space doesn't have just one class. Suppose your data is adult men and women with just one feature: their height. You want to train a classifier that takes in an adult's height and returns a classification, man or woman. Suppose you are asked to predict the sex of a 5'5" adult. Well, your training set includes some 5'5" women and some 5'5" men. What should you do?]

[In your feature space, you have two training points at the same location with different classes. More generally, the height distributions of men and women overlap. 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 occupation conditioned on cancer, $P(X|Y)$:

job	(X)	miner	farmer	other
cancer	$(Y = 1)$	20%	50%	30%
no cancer	$(Y = -1)$	1%	10%	89%

[caps here mean random variables, not matrices.]

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

Recall: $P(X) = P(X|Y = 1)\,P(Y = 1) + P(X|Y = -1)\,P(Y = -1)$
$P(X = \text{farmer}) = 0.5 \times 0.1 + 0.1 \times 0.9 = 0.14$ [... so 14% of random people are farmers]

You meet a farmer. Guess whether he has cancer?

[If you're in a hurry, you might see that 50% of people with cancer are farmers, but only 10% of people with no cancer are farmers, and conclude that a typical farmer probably has cancer. But that would be wrong, because that reasoning fails to take the prior probabilities into account.]

Bayes' Theorem:

$$\begin{array}{rcll} & & \downarrow \text{posterior probability} \quad \downarrow \text{prior prob.} & \downarrow \text{if } X = \text{ farmer} \\ P(Y = 1|X) & = & \dfrac{P(X|Y = 1)P(Y = 1)}{P(X)} & = \dfrac{0.05}{0.14} \\ P(Y = -1|X) & = & \dfrac{P(X|Y = -1)P(Y = -1)}{P(X)} & = \dfrac{0.09}{0.14} \end{array}$$

[These two probs always sum to 1.]

$P(\text{cancer} \mid \text{farmer}) = 5/14 \approx 36\%$.

[So we probably shouldn't diagnose cancer.]

[BUT ... we're 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. When there's an asymmetry between the awfulness of false positives and false negatives, we can quantify that with a loss function.]

A loss function $L(\hat{y}, y)$ specifies badness if classifier predicts $\hat{y}$, true class is y.

E.g., $L(\hat{y}, y) = \begin{cases} 1 & \text{if } \hat{y} = 1, y = -1, \quad \text{false positive is bad} \\ 5 & \text{if } \hat{y} = -1, y = 1, \quad \text{false negative is BAAAAAD} \\ 0 & \text{if } \hat{y} = y. \quad \text{[loss should } \textit{always} \text{ be zero for a perfectly correct prediction!]} \end{cases}$

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

The loss fn above is asymmetrical.
[A loss is symmetrical if it is the same for false positives and false negatives. For example . . .]

The 0-1 loss function is $L(\hat{y}, y) = \begin{cases} 1 & \text{if } \hat{y} \neq y, \quad \text{[always 1 for a wrong prediction]} \\ 0 & \text{if } \hat{y} = y. \quad \text{[always 0 for a correct prediction]} \end{cases}$

[Another application where you want a very asymmetrical loss function, besides medical diagnosis, is 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. [Memorize this definition!]

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

The Bayes decision rule aka Bayes classifier is the fn r^* that minimizes functional $R(r)$.
Assuming $L(1, 1) = L(-1, -1) = 0$,

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

When L is symmetrical, [the big, key principle you should memorize is]
pick the class with the biggest posterior probability.

[But if the loss function is asymmetrical, then you must weight the posteriors with the losses.]
In cancer example, $r^*(\text{miner}) = 1$, $r^*(\text{farmer}) = 1$, and $r^*(\text{other}) = -1$.

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

$R(r^*) = 0.1(5 \times 0.3) + 0.9(1 \times 0.01 + 1 \times 0.1) = 0.249.$ No decision rule gives a lower risk.

[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.]

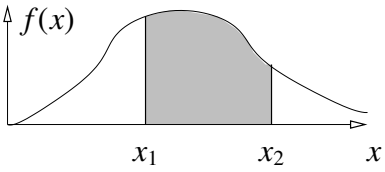
Deriving/using r^* is called risk minimization.

[Did you memorize the two boldfaced lines above yet?]

Continuous Distributions

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.]



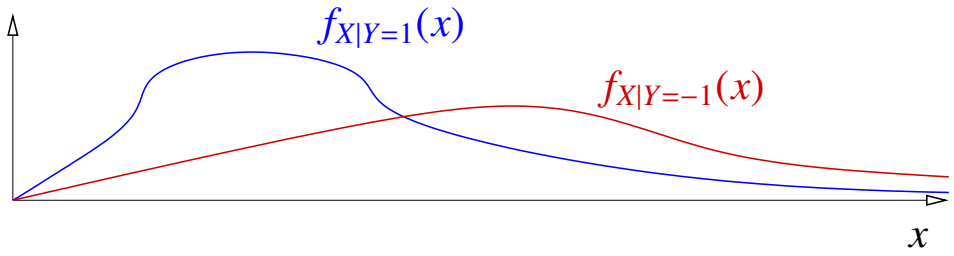
[Draw this by hand. integrate.pdf]

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

area under whole curve $= 1 = \int_{-\infty}^{\infty} f(x)\,dx$ expected value of $g(X)$: $\mathrm{E}[g(X)] = \int_{-\infty}^{\infty} g(x)f(x)\,dx$

mean $\mu = \mathrm{E}[X] = \int_{-\infty}^{\infty} x\,f(x)\,dx$ variance $\sigma^2 = \mathrm{E}[(X-\mu)^2] = \mathrm{E}[X^2] - \mu^2$

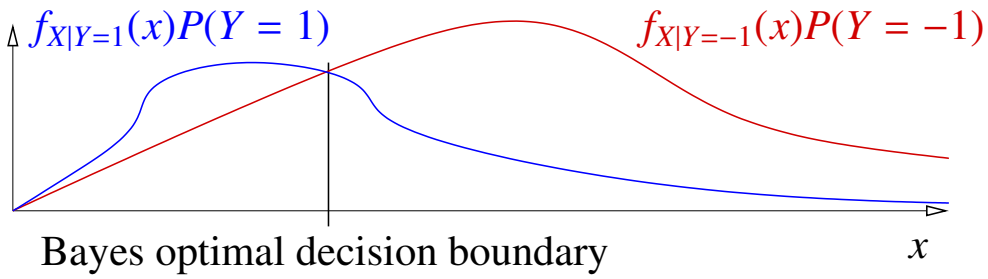
[Perhaps our cancer statistics look like this.]



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

[Let's use the 0-1 loss function. In other words, suppose 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.

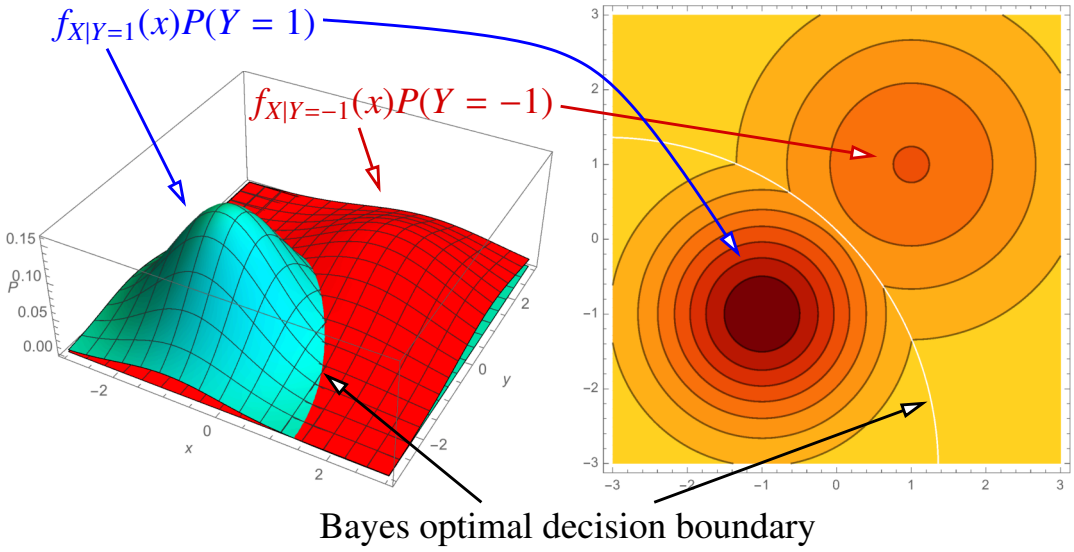
$$\begin{aligned} R(r) &= \mathrm{E}[L(r(X), Y)] \\ &= P(Y = 1) \int L(r(x), 1)\, f_{X|Y=1}(x)\, \mathrm{d}x + \\ &\quad P(Y = -1) \int L(r(x), -1)\, f_{X|Y=-1}(x)\, \mathrm{d}x. \end{aligned}$$

For Bayes decision rule, Bayes risk is the area under minimum of functions above. [Shade it.]
Assuming $L(1, 1) = L(-1, -1) = 0$,

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

[If you want to use an asymmetrical loss function, just scale the curves vertically 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 $\{x : \underbrace{P(Y = 1|X = x)}_{\text{decision fn}} = \underbrace{0.5}_{\text{isovalue}}\}$



qda3d.pdf, qdacontour.pdf [Two different views of the same 2D Gaussians.]

[Notice that 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, choosing among three or more classes. 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 $f_{X|Y=\mathrm{C}}(x)$
- For each C, estimate $P(Y = \mathrm{C})$
- Bayes' Theorem gives $P(Y|X)$
- If 0-1 loss, pick class C that maximizes $P(Y = \mathrm{C}|X = x)$ [posterior probability]
  equivalently, maximizes $f_{X|Y=\mathrm{C}}(x)\, P(Y = \mathrm{C})$

(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.]
Advantages of (1): you can diagnose outliers: $f(x)$ is very small;
stabler for outliers or few training points.
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 that we want to predict.]

[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. In practice, generative models are most popular when you have phenomena that are well approximated by the normal distribution or another "nice" distribution. Generative methods also tend to be more stable than other methods when the number of training points is small or when there are a lot of outliers.]

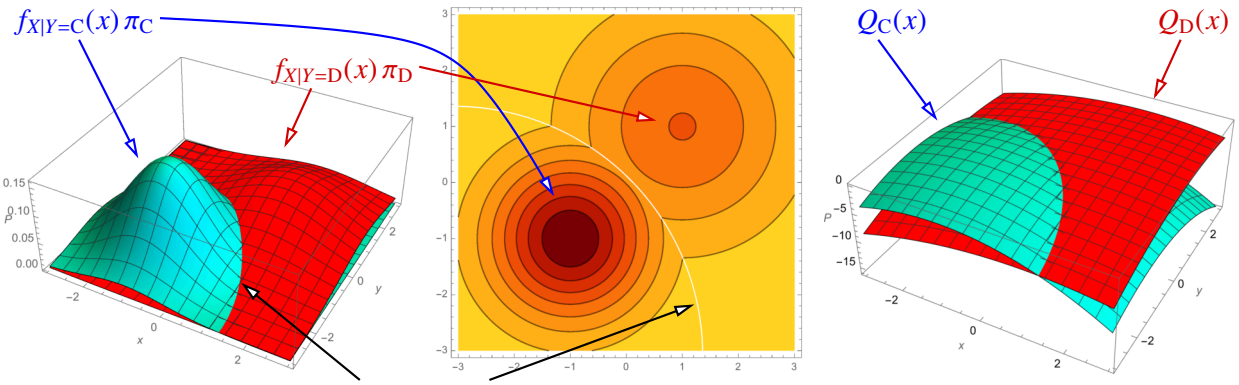
7 Gaussian Discriminant Analysis; Maximum Likelihood Estimation

GAUSSIAN DISCRIMINANT ANALYSIS

Fundamental assumption: each class has a normal distribution [a Gaussian].

$$X \sim \mathcal{N}(\mu, \sigma^2) : f(x) = \frac{1}{(\sqrt{2\pi}\sigma)^d} \exp\left(-\frac{\|x-\mu\|^2}{2\sigma^2}\right). \qquad [\mu \text{ \& } x = \text{vectors; } \sigma = \text{scalar; } d = \text{dimension}]$$

For each class C, suppose we know mean μ_C and variance σ_C^2, yielding PDF $f_{X|Y=C}(x)$,
and prior $\pi_C = P(Y = C)$.



qda3d.pdf, qdacontour.pdf, Q.pdf [Probability density functions for two classes. The Bayes optimal decision boundary is an ellipse.]

[This PDF is a simplified version of the multivariate normal distribution. It is multivariate: x and μ can be vectors, and this plot shows a 2D feature space. But the variance σ^2 is just a scalar; for simplicity, we will avoid the covariance matrix until next lecture. That's why the isocontours are circles, not ellipses. I call this the *isotropic normal distribution*, because the variance is the same in every direction. Next lecture, we'll use the usual multivariate normal distribution, where the isosurfaces are ellipsoids.]

Given x, Bayes decision rule $r^*(x)$ predicts class C that maximizes $f_{X|Y=C}(x)\,\pi_C$.
[Remember our last lecture's main principle: **pick the class with the biggest posterior probability!**]

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

$$Q_C(x) = \ln\left((\sqrt{2\pi})^d f_{X|Y=C}(x)\,\pi_C\right) = -\frac{\|x-\mu_C\|^2}{2\sigma_C^2} - d\ln\sigma_C + \ln\pi_C. \qquad [Q_C \text{ is quadratic in } x]$$

[In a 2-class problem, you can also incorporate an asymmetrical loss function by adding $\ln L(\text{not C}, \text{C})$ to $Q_C(x)$. In a multi-class problem, asymmetric loss may be more difficult to account for, because the penalty for guessing wrong might depend on both the wrong prediction and the true class.]

Quadratic Discriminant Analysis (QDA)

Suppose only 2 classes C, D. Then the Bayes classifier is

$$r^*(x) = \begin{cases} \text{C} & \text{if } Q_C(x) - Q_D(x) > 0, \\ \text{D} & \text{otherwise.} \end{cases} \qquad [\text{Picks the class with the biggest posterior probability}]$$

Decision fn is $Q_C(x) - Q_D(x)$ (quadratic); Bayes decision boundary is $\{x : Q_C(x) - Q_D(x) = 0\}$.

– In 1D, B.d.b. may have 1 or 2 points. [Solutions to a quadratic equation]
– In d-D, B.d.b. is a quadric. [In 2D, that's a conic section; see figure above]

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

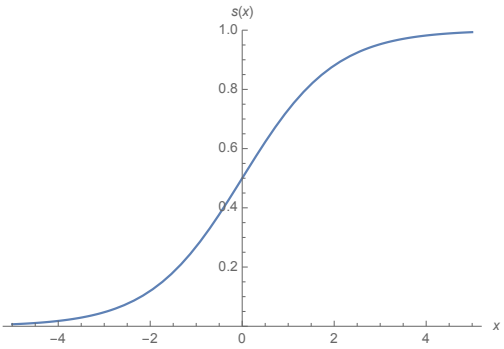
To recover posterior probabilities in 2-class case, use Bayes.

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

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

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

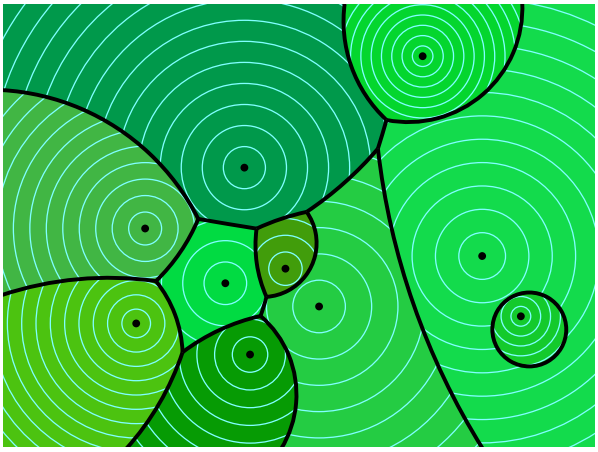
$$s(\gamma) = \frac{1}{1 + e^{-\gamma}} \quad \Leftarrow \underline{\text{logistic fn}} \text{ aka } \underline{\text{sigmoid fn}}$$ [recall $Q_C - Q_D$ is the decision fn]



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.]

Multi-class QDA: [QDA works very naturally with more than 2 classes.]



multiplicative.pdf [Multi-class QDA partitions the feature space 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, additively weighted Voronoi diagram.]

Linear Discriminant Analysis (LDA)

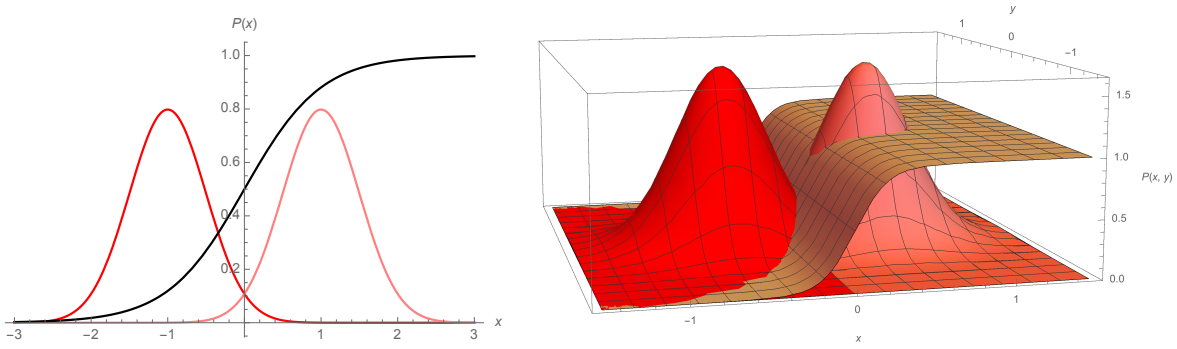
[LDA is a variant of QDA with linear decision boundaries. It's less likely to overfit than QDA.]
Fundamental assumption: all the Gaussians have same variance σ^2.
[The equations simplify nicely in this case.]

$$Q_C(x) - Q_D(x) = \underbrace{\frac{(\mu_C - \mu_D) \cdot x}{\sigma^2}}_{w \cdot x} \underbrace{- \frac{\|\mu_C\|^2 - \|\mu_D\|^2}{2\sigma^2} + \ln \pi_C - \ln \pi_D}_{+\alpha}.$$

[The quadratic terms in Q_C and Q_D canceled each other out!]
Now it's a linear classifier!

- decision boundary is $w \cdot x + \alpha = 0$
- posterior is $P(Y = C|X = x) = s(w \cdot x + \alpha)$

[The effect of "$s(w \cdot x + \alpha)$" is to scale and translate the logistic fn in x-space.]

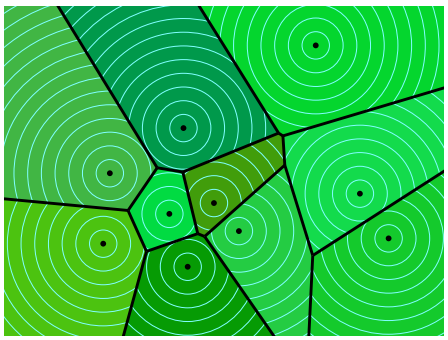


lda1d.pdf, lda2d.pdf [Two Gaussians (red) and the logistic function (black). The logistic function is the right Gaussian divided by the sum of the Gaussians. Observe that even when the Gaussians are 2D, the logistic function still looks 1D.]

Special case: if $\pi_C = \pi_D = \frac{1}{2} \quad \Rightarrow \quad (\mu_C - \mu_D) \cdot x - (\mu_C - \mu_D) \cdot \left(\frac{\mu_C + \mu_D}{2}\right) = 0.$

This is the centroid method!

Multi-class LDA: choose C that maximizes linear discriminant fn $\frac{\mu_C \cdot x}{\sigma^2} - \frac{\|\mu_C\|^2}{2\sigma^2} + \ln \pi_C$.



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

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

[To use Gaussian discriminant analysis, we must first fit Gaussians to the sample points and estimate the class prior probabilities. We'll do priors first—they're easier, because they involve a discrete distribution. Then we'll fit the Gaussians—they're less intuitive, because they're continuous distributions.]

Let's flip biased coins! Heads with probability p; tails w/prob. $1 - p$. [But we don't know p.]

10 flips, 8 heads, 2 tails. [Let me ask you a weird question.] What is the most likely value of p?

# of heads is $X \sim \mathcal{B}(n, p)$, binomial distribution:

$$P(X = x) = \binom{n}{x} p^x (1-p)^{n-x} \qquad \text{[this is the probability of getting exactly } x \text{ heads in } n \text{ coin flips]}$$

Prob. of $x = 8$ heads in $n = 10$ flips is

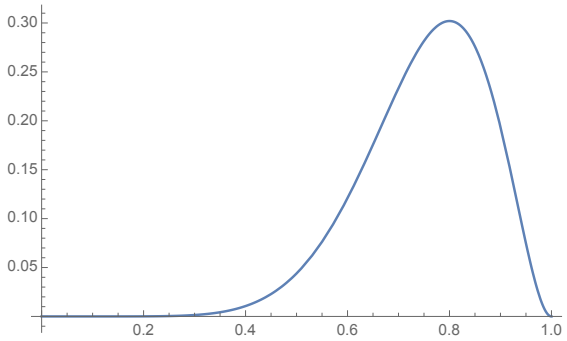
$$P(X = 8) = 45p^8 (1-p)^2 \qquad \stackrel{\text{def}}{=} \mathcal{L}(p)$$

Written as a fn of distribution parameter p, this prob. is the likelihood fn $\mathcal{L}(p)$.

Maximum likelihood estimation (MLE): A method of estimating the parameters of a statistical model by picking the params that maximize [the likelihood function] $\mathcal{L}$.
. . . is one method of density estimation: estimating a PDF [probability density function] from data.

[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 by finding critical point of $\mathcal{L}$:

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

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

[It shouldn't seem surprising that a coin that is biased so it 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.]

[Here's how this applies to prior probabilities.]
Suppose our training set is n points, with x in class C. Then our estimated prior for class C is $\hat{\pi}_\mathrm{C} = x/n$.

Likelihood of a Gaussian

Given sample points $X_1, X_2, \ldots, X_n$, find best-fit Gaussian.

[Now we want to fit a normal distribution to data, instead of a binomial distribution. If you draw a random point from a normal distribution, what is the probability that it will be exactly at X_1?]

[Zero. So it might seem like we have 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 drawing these points [in the specified order] is

$$\mathcal{L}(\mu, \sigma; X_1, \ldots, X_n) = f(X_1)\, f(X_2) \cdots f(X_n). \qquad \text{[How do we maximize this?]}$$

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

$$\begin{aligned} \ell(\mu, \sigma; X_1, ..., X_n) &= \ln f(X_1) + \ln f(X_2) + ... + \ln f(X_n) \\ &= \sum_{i=1}^{n} \underbrace{\left(-\frac{\|X_i - \mu\|^2}{2\sigma^2} - d \ln \sqrt{2\pi} - d \ln \sigma \right)}_{\text{ln of normal PDF}} \end{aligned}$$

Set $\nabla_\mu \ell = 0$, $\dfrac{\partial \ell}{\partial \sigma} = 0$ [Find the critical point of ℓ]

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

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

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

Takeaway: use sample mean & variance of pts in class C to estimate mean & variance of Gaussian for class C.

For QDA: estimate conditional mean $\hat{\mu}_C$ & conditional variance $\hat{\sigma}_C^2$ of **each class C separately** [as above] & estimate the priors:

$$\hat{\pi}_C = \frac{n_C}{\sum_D n_D} \quad \Leftarrow \quad \text{total sample points in all classes} \qquad [\hat{\pi}_C \text{ is the coin flip parameter}]$$

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

$$\hat{\sigma}^2 = \frac{1}{dn} \sum_{C} \sum_{\{i : y_i = C\}} \|X_i - \hat{\mu}_C\|^2 \qquad \Leftarrow \text{ pooled within-class variance}$$

[Notice that although LDA is computing one variance 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 use the global mean! It's usually smaller than the global variance. We say "within-class" because we use each point's distance from its class's mean, but "pooled" because we then pool all the classes together.]

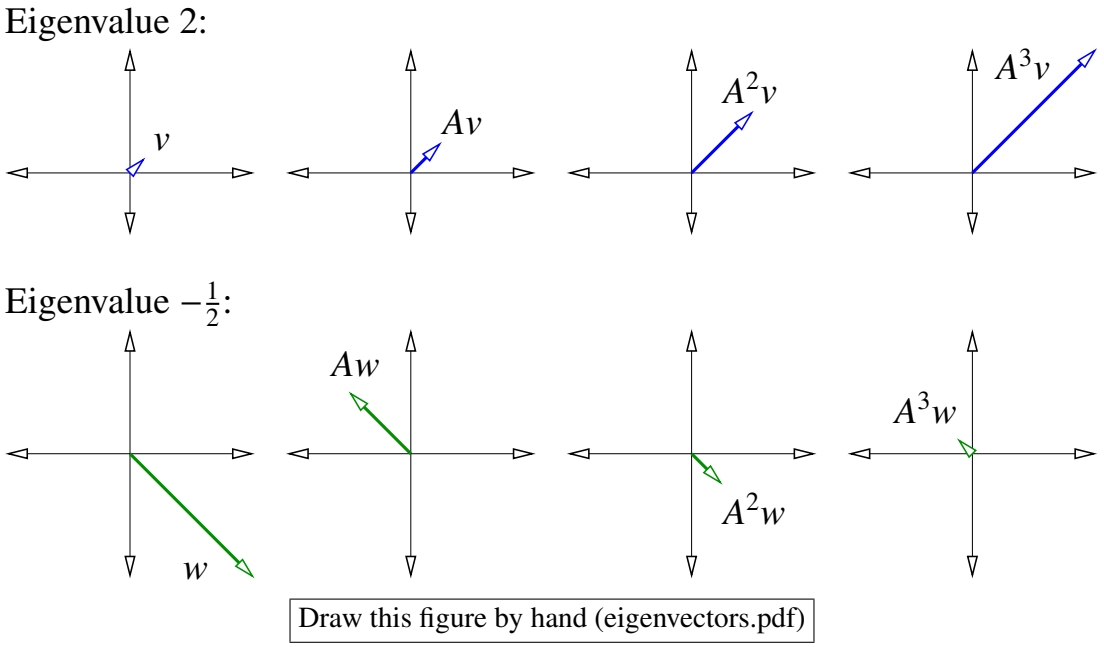
8 Eigenvectors and the (Anisotropic) Multivariate Normal Distribution

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 eigenvalue of A associated w/v.

[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.pdf)

[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 [k is a +ve integer; we will use Theorem later]

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

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

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

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

[Those theorems are pretty obvious. The next theorem is not obvious at all.]

Spectral Theorem: every real, symmetric $n \times n$ matrix has real eigenvalues and
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 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 $\mathbb{R}^n$.

Building a Matrix with Specified Eigenvectors

[There are a lot of applications where you're given a matrix, and you want to extract the eigenvectors and eigenvalues. But when you're learning the math, I think it's more intuitive to go in the opposite direction. Suppose you know what eigenvectors and eigenvalues you want, and you want to create the matrix that has those eigenvectors and eigenvalues.]

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 eigenvalues λ_i:

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

Defn. of eigenvector: $AV = V\Lambda$
[This is the same definition of eigenvector I gave you at the start of the lecture—$Av = \lambda v$—but this version covers all n eigenvectors in one statement. How do we find the A that satisfies this equation?]

$$\Rightarrow AVV^\top = V\Lambda V^\top \qquad \text{[which proves ...]}$$

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

This is a matrix factorization called the eigendecomposition. [every real, symmetric matrix has one]
Example: [Using the eigenvectors and eigenvalues from the start of the lecture]

$$A = \begin{bmatrix} 1/\sqrt{2} & 1/\sqrt{2} \\ 1/\sqrt{2} & -1/\sqrt{2} \end{bmatrix} \begin{bmatrix} 2 & 0 \\ 0 & -1/2 \end{bmatrix} \begin{bmatrix} 1/\sqrt{2} & 1/\sqrt{2} \\ 1/\sqrt{2} & -1/\sqrt{2} \end{bmatrix} = \begin{bmatrix} 3/4 & 5/4 \\ 5/4 & 3/4 \end{bmatrix}.$$

[This completes our task of finding a symmetric matrix with specified orthonormal eigenvectors and eigenvalues. Again, it is more common in practice that you are given a symmetric matrix, such as a sample covariance matrix, and you need to compute its eigenvectors and eigenvalues. That's harder. But I think that going from eigenvectors to the matrix helps to build intuition.]

Observe: $A^2 = V\Lambda V^\top V\Lambda V^\top = V\Lambda^2 V^\top$ $\qquad$ $A^{-1} = (V\Lambda V^\top)^{-1} = (V^\top)^{-1}\Lambda^{-1}V^{-1} = V\Lambda^{-1}V^\top$
[This is another way to see that squaring a matrix squares its eigenvalues without changing its eigenvectors. It also suggests a way to define a matrix square root.]
Given a symmetric PSD matrix Σ, we can find a symmetric square root $A = \Sigma^{1/2}$:

compute eigenvectors/values of Σ
take square roots of Σ's eigenvalues
reassemble matrix A [with the same eigenvectors as Σ but changed eigenvalues]

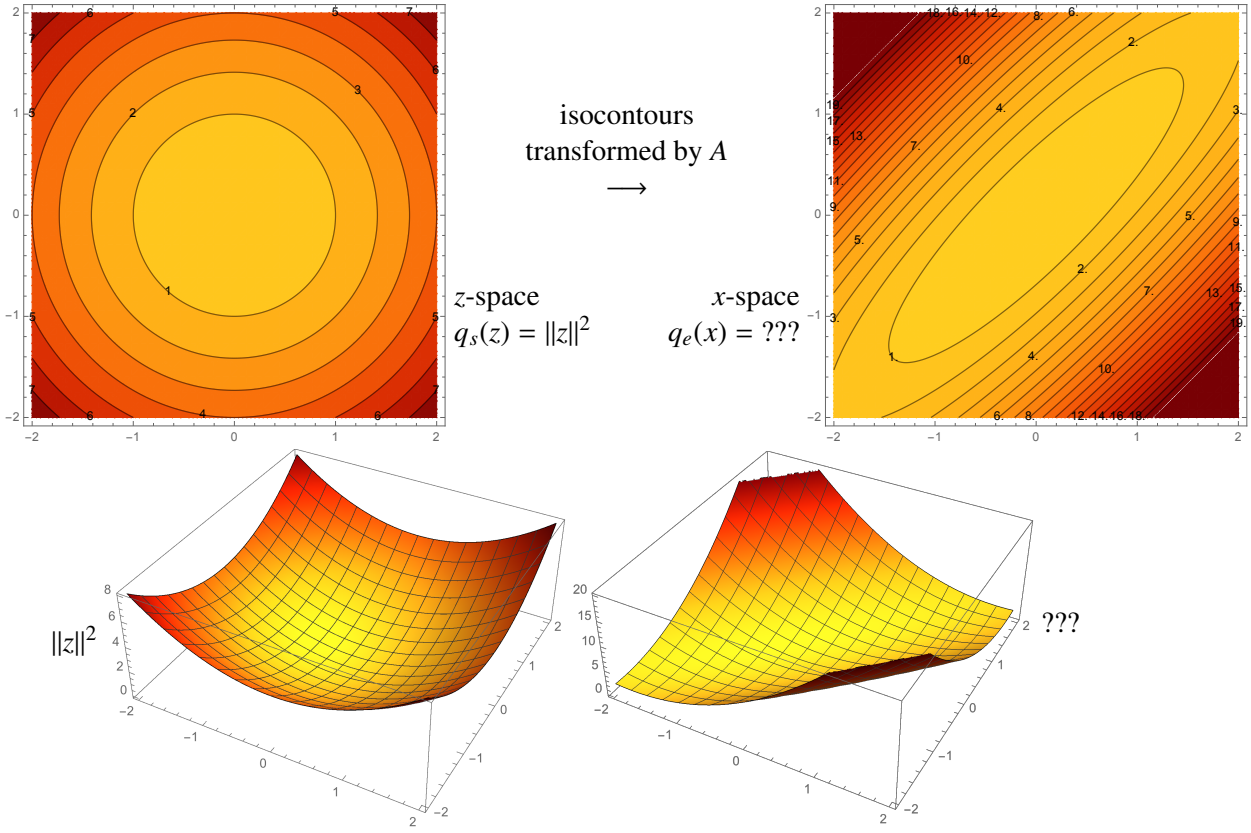
[Again, the first step of this algorithm—computing the eigenvectors and eigenvalues of a matrix—is much harder than the remaining two steps.]

Visualizing Quadratic Forms

[My favorite way to visualize a symmetric matrix is to graph something called *the quadratic form*, which shows how applying the matrix affects the length of a vector.]

The quadratic form of M is $x^\top Mx$.

Suppose you want a matrix whose quadratic form has the isocontours at right below, which are circles transformed by A. [The same matrix A I've been using, which stretches along the direction with eigenvalue 2 and shrinks along the direction with eigenvalue $-1/2$.]



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

[Both figures at left are plots of $||z||^2$, and both figures at right are plots of $x^\top A^{-2}x$.
(Draw the stretch direction $(1, 1)$ with eigenvalue 2 and the shrink direction $(1, -1)$ with eigenvalue $-\frac{1}{2}$ on the ellipses at right.)]

That is, we want $q_e(Az) = q_s(z)$.
Answer: set $x = Az$.
Then $q_e(x) = q_s(z) = q_s(A^{-1}x) = \|A^{-1}x\|^2 = x^\top A^{-2}x$.

The isocontours of the quadratic form $x^\top A^{-2}x$ are ellipsoids determined by the eigenvectors/values of A.
$\{x : x^\top A^{-2}x = 1\}$ is an ellipsoid with axes $v_1, v_2, \ldots, v_n$ and
radii $\lambda_1, \lambda_2, \ldots, \lambda_n$
because if v_i has length 1 (v_i lies on unit circle), $x = Av_i$ has length λ_i (Av_i lies on the ellipsoid).

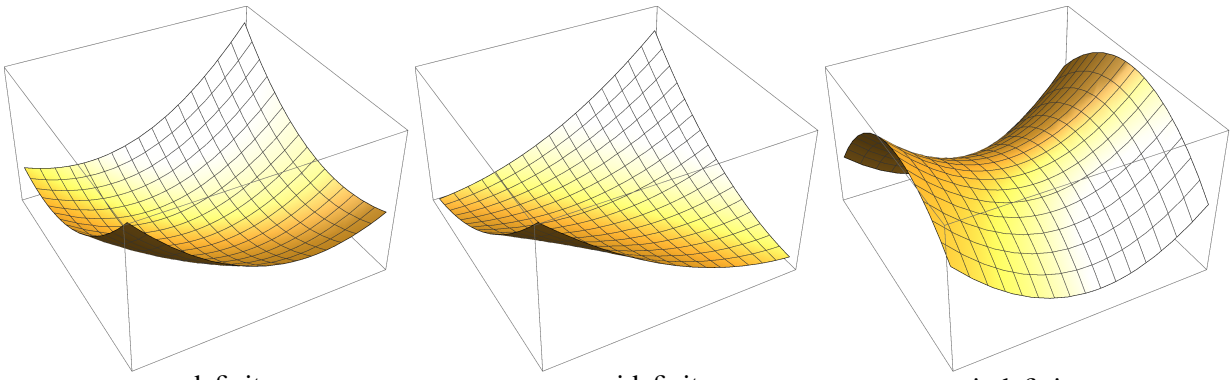
Therefore, isocontours of $x^\top Mx$ are ellipsoids determined by eigenvectors/values of $M^{-1/2}$.
[The eigenvalues of $M^{-1/2}$ are the inverse square roots of the eigenvalues of M.]

Special case: A (or M) is diagonal $\Leftrightarrow$ eigenvectors are coordinate axes
$\Leftrightarrow$ ellipsoids are axis-aligned

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

A symmetric matrix M is

positive definite	if $w^\top Mw > 0$ for all $w \neq 0 \Leftrightarrow$ all eigenvalues positive
positive semidefinite	if $w^\top Mw \geq 0$ for all $w \Leftrightarrow$ all eigenvalues nonnegative
indefinite	if +ve eigenvalue & −ve eigenvalue
invertible	if no zero eigenvalue



pos definite pos semidefinite indefinite

posdef.pdf, possemi.pdf, indef.pdf
[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.)]

Every squared matrix is pos semidef, including A^{-2}. [Eigenvalues of A^{-2} are squared, cannot be negative.]
If A^{-2} exists, it is pos def. [An invertible matrix has no zero eigenvalues.]

What about the isosurfaces of $x^\top Mx$ for a +ve semidef, *singular* M?

[If M is only positive semidefinite, but not positive definite, the isosurfaces are cylinders instead of ellipsoids. These cylinders have ellipsoidal cross sections spanning the directions with nonzero eigenvalues, but they run in straight lines along the directions with zero eigenvalues.]

ANISOTROPIC GAUSSIANS

[Let's revisit the multivariate Gaussian distribution, with different variances along different directions.]

$X \sim \mathcal{N}(\mu, \Sigma)$ [X and μ are d-vectors. X is a random variable with mean μ.]

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

↑ determinant of Σ

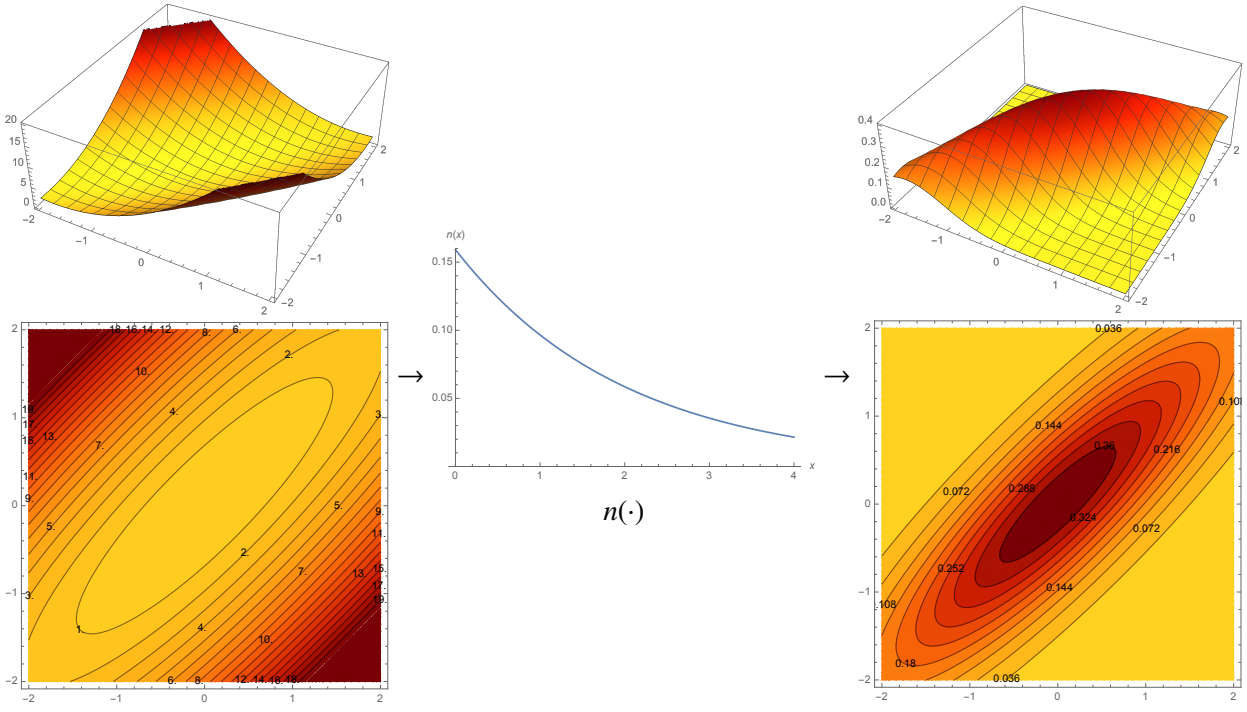
Σ is the $d \times d$ SPD covariance matrix.
Σ^{-1} is the $d \times d$ SPD precision matrix.

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

↑ $\mathbb{R} \to \mathbb{R}$, 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, the quadratic form of the precision matrix Σ^{-1}. The other function $n(\cdot)$ is a simple, monotonic, convex function, an exponential of the negation of half its argument. This mapping $n(\cdot)$ does not change the isosurfaces.]

Principle: given monotonic $n : \mathbb{R} \to \mathbb{R}$, isosurfaces of $n(q(x))$ are same as $q(x)$ (different isovalues).



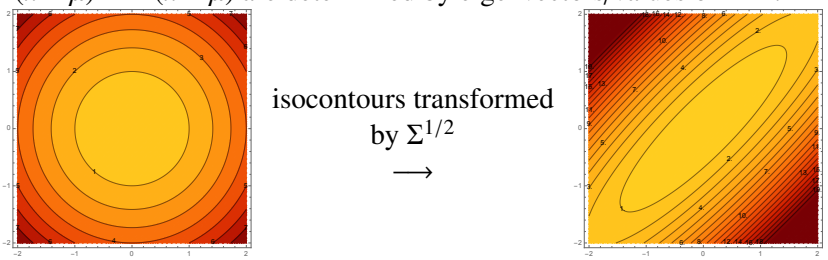
$q(x)$ $n(\cdot)$ $f(x) = n(q(x))$

ellipsebowl.pdf, ellipses.pdf, exp.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 a suitable scalar function (center).]

[One of the main ideas is that if you understand the isosurfaces of a quadratic function, then you understand the isosurfaces of a Gaussian, because they're the same. The differences are in the isovalues—in particular, the Gaussian achieves its maximum at the mean, and decreases to zero as you move infinitely far away from the mean.]

The isocontours of $(x-\mu)^T\Sigma^{-1}(x-\mu)$ are determined by eigenvectors/values of $\Sigma^{1/2}$.



Aside: $q(x)$ is the squared distance from $\Sigma^{-1/2}x$ to $\Sigma^{-1/2}\mu$. Consider the metric

$$d(x,\mu) = \left\|\Sigma^{-1/2}x - \Sigma^{-1/2}\mu\right\| = \sqrt{(x-\mu)^T\Sigma^{-1}(x-\mu)} = \sqrt{q(x)}.$$

[So we think of the precision matrix as a "metric tensor" which defines a metric, a sort of warped distance from x to the mean μ.]

Covariance

Let R, S be random variables—column vectors or scalars
$\text{Cov}(R,S) = \text{E}[(R-\text{E}[R])(S-\text{E}[S])^T] = \text{E}[RS^T] - \mu_R\mu_S^T$
$\text{Var}(R) = \text{Cov}(R,R)$
If R is a vector, covariance matrix for R is

$$\text{Var}(R) = \begin{bmatrix} \text{Var}(R_1) & \text{Cov}(R_1,R_2) & \dots & \text{Cov}(R_1,R_d) \\ \text{Cov}(R_2,R_1) & \text{Var}(R_2) & & \text{Cov}(R_2,R_d) \\ \vdots & & \ddots & \vdots \\ \text{Cov}(R_d,R_1) & \text{Cov}(R_d,R_2) & \dots & \text{Var}(R_d) \end{bmatrix} \quad \text{[symmetric; each } R_i \text{ is scalar]}$$

For a Gaussian $R \sim \mathcal{N}(\mu,\Sigma)$, one can show $\text{Var}(R) = \Sigma$. [. . . as you did in Homework 2.]
[An important point is that statisticians didn't just arbitrarily decide to call Σ a covariance matrix. Rather, statisticians discovered that if you find the covariance of the normal distribution by integration, it turns out that the covariance is Σ. This is a happy fact; it's rather elegant.]

R_i, R_j independent $\Rightarrow$ $\text{Cov}(R_i,R_j) = 0$ [the reverse implication is not generally true, but . . .]
$\text{Cov}(R_i,R_j) = 0$ AND multivariate normal dist. $\Rightarrow$ R_i, R_j independent

all features pairwise independent $\Rightarrow$ $\text{Var}(R)$ is diagonal [the reverse is not generally true, but . . .]
$\text{Var}(R)$ is diagonal AND multi normal
$\Leftrightarrow$ $\underbrace{f(x)}_{\text{multivariate}} = \underbrace{f(x_1)\,f(x_2)\cdots f(x_d)}_{\text{univariate Gaussians}}$
$\Rightarrow$ ellipsoids are axis-aligned, with squared radii on diagonal of $\Sigma = \text{Var}(R)$

[So when the features are independent, you can write the multivariate Gaussian PDF as a product of univariate Gaussian PDFs. When they aren't, you can do a change of coordinates to the eigenvector coordinate system, and write it as a product of univariate Gaussian PDFs in eigenvector coordinates. You did something very similar in Q7.2 of Homework 2.]

9 Anisotropic Gaussians: MLE, QDA, and LDA Revisited

GDA WITH ANISOTROPIC GAUSSIANS

[Recall from our last lecture the probability density function of the multivariate normal distribution in its full generality. x and μ are d-vectors.]

Normal PDF: $f(x) = n(q(x)), \qquad n(q) = \dfrac{1}{\sqrt{(2\pi)^d |\Sigma|}} e^{-q/2}, \qquad q(x) = (x-\mu)^\top \Sigma^{-1} (x-\mu).$

$\mathbb{R} \to \mathbb{R}$, exponential; determinant of Σ; $\mathbb{R}^d \to \mathbb{R}$, quadratic

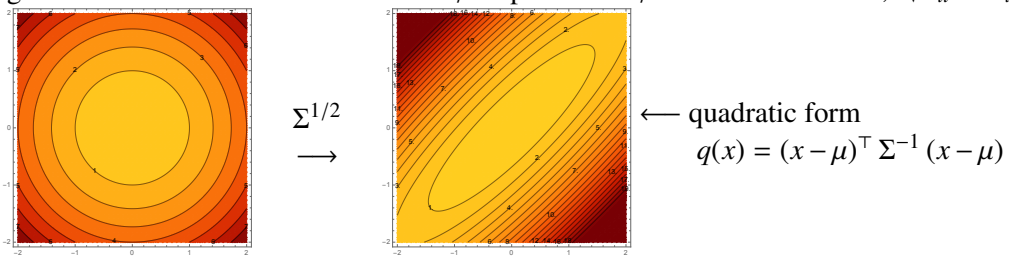
[The covariance matrix Σ and its symmetric square root and its inverse all play roles in our intuition about the multivariate normal distribution. Consider their eigendecompositions.]

$\Sigma = V \Gamma V^\top$ covariance matrix

↑ eigenvalues of Σ are variances along the eigenvectors, $\Gamma_{ii} = \sigma_i^2$

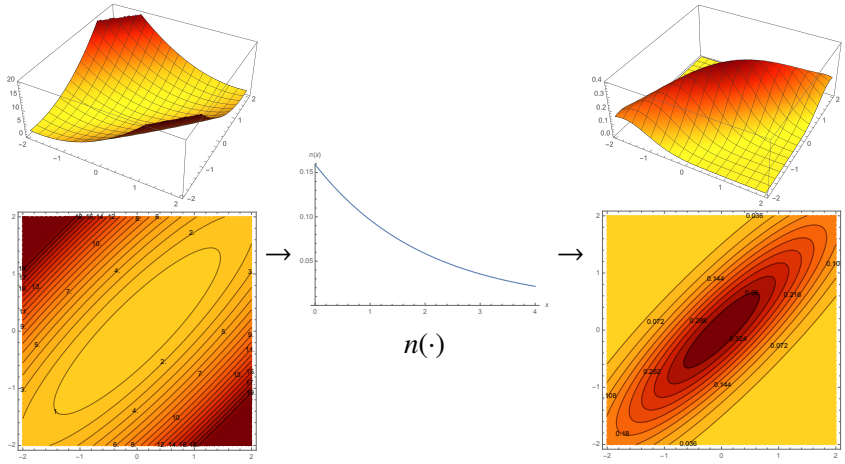
$\Sigma^{1/2} = V \Gamma^{1/2} V^\top$ maps spheres to ellipsoids [recall end of last lecture]

↑ eigenvalues of $\Sigma^{1/2}$ are Gaussian widths / ellipsoid radii / standard deviations, $\sqrt{\Gamma_{ii}} = \sigma_i$

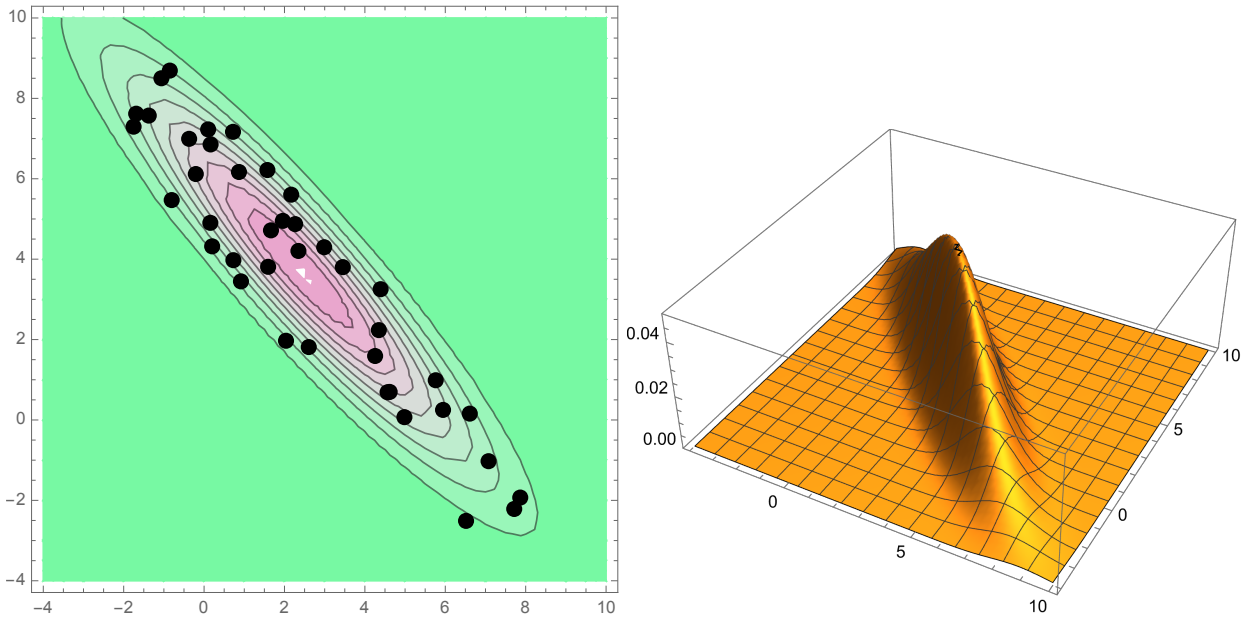


$\Sigma^{-1} = V \Gamma^{-1} V^\top$ precision matrix (metric tensor) [↑ quadratic form of Σ^{-1} defines contours]

[Recall from last lecture that the isocontours of the multivariate normal distribution are the same as the isocontours of the quadratic form of the precision matrix Σ^{-1}.]



Maximum Likelihood Estimation for Anisotropic Gaussians



MLE40pts.pdf, MLE40pts3D.pdf [Maximum likelihood estimation takes these 40 points as input and outputs this Gaussian. Note that the points do not actually come from a normal distribution; they come from a uniform distribution over a tilted rectangle. Nevertheless, the Gaussian is a decent approximation of that.]

Given training points $X_1, \ldots, X_n$ and classes $y_1, \ldots, y_n$, find best-fit Gaussians.
Let n_C = # of training pts in class C.

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

For QDA:

$$\hat{\Sigma}_C = \frac{1}{n_C} \sum_{i:y_i=C} \underbrace{(X_i - \hat{\mu}_C)(X_i - \hat{\mu}_C)^T}_{\text{outer product matrix, } d \times d} \qquad \Leftarrow \text{ conditional covariance for pts in class C}$$

Prior $\hat{\pi}_C$, mean $\hat{\mu}_C$: same as Lecture 7.
[$\hat{\pi}_C$ is number of training points in class C ÷ total training points; $\hat{\mu}_C$ is mean of training points in class C.]

$\hat{\Sigma}_C$ is positive semidefinite, but not always definite!
[If there are some zero eigenvalues, the standard version of QDA just doesn't work. We can try to fix it by eliminating the zero-variance dimensions (eigenvectors). Homework 3 suggests a way to do that.]

For LDA:

$$\hat{\Sigma} = \frac{1}{n} \sum_{C} \sum_{i:y_i=C} (X_i - \hat{\mu}_C)(X_i - \hat{\mu}_C)^T \qquad \Leftarrow \text{ pooled within-class covariance matrix}$$

[Let's revisit QDA and LDA and see what has changed now that we use the multivariate normal distribution in its full, anisotropic generality. The short answer is "not much has changed, but the graphs look cooler." Conflicting notation warning: capital X represents a random variable, but later it will represent a matrix.]

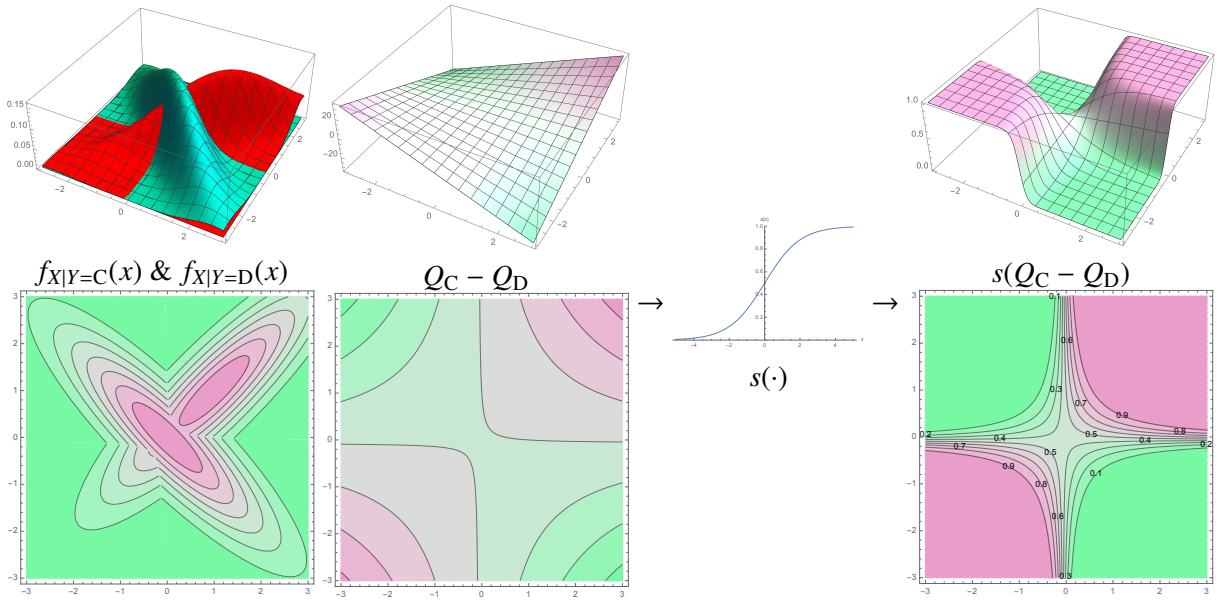
QDA

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

$$Q_{\mathrm{C}}(x) = \ln\left((\sqrt{2\pi})^d f_{X|Y=\mathrm{C}}(x)\,\pi_{\mathrm{C}}\right) = -\frac{1}{2}(x-\mu_{\mathrm{C}})^\top \Sigma_{\mathrm{C}}^{-1}(x-\mu_{\mathrm{C}}) - \frac{1}{2}\ln|\Sigma_{\mathrm{C}}| + \ln\pi_{\mathrm{C}}.$$

[This works for any number of classes. In a multi-class problem, you just pick the class with the greatest quadratic discriminant for x.]

2 classes: Decision fn $Q_{\mathrm{C}}(x) - Q_{\mathrm{D}}(x)$ is quadratic, but may be indefinite.
$\Rightarrow$ Decision boundary is a quadric.
Posterior is $P(Y = \mathrm{C}|X = x) = s(Q_{\mathrm{C}}(x) - Q_{\mathrm{D}}(x))$ where $s(\cdot)$ is logistic fn.

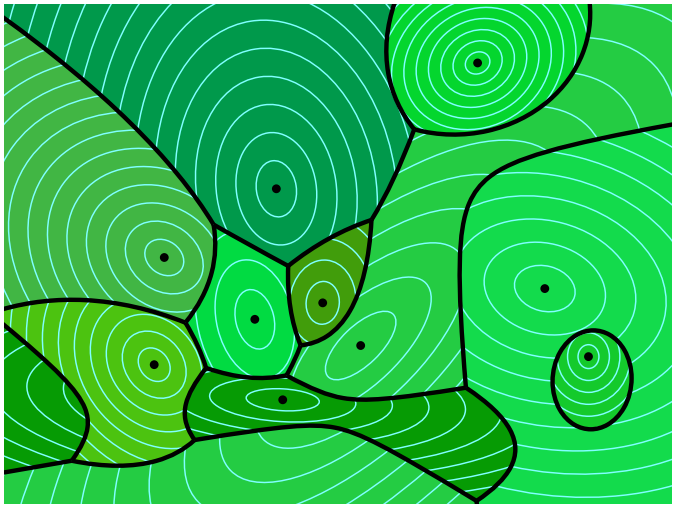


qdaaniso3d.pdf, qdaanisocontour.pdf, qdaanisodiff3d.pdf, qdaanisodiffcontour.pdf, logistic.pdf, qdaanisoposterior3d.pdf, qdaanisoposteriorcontour.pdf

[(Show this figure on a separate "whiteboard.") An example where the decision boundary is a hyperbola—which is not possible with isotropic Gaussians. At left, two anisotropic Gaussians. Center left, the difference $Q_{\mathrm{C}} - Q_{\mathrm{D}}$. After applying the logistic function to this difference we obtain the posterior probabilities at right, which tells us the probability that x is in class C. Observe that we can see the decision boundary in all three contour plots: it is $Q_{\mathrm{C}} - Q_{\mathrm{D}} = 0$ and $s(Q_{\mathrm{C}} - Q_{\mathrm{D}}) = 0.5$. We don't need to apply the logistic function to find the decision boundary, but we do need to compute it if we want the posterior probabilities.]

[This procedure has two interpretations. If we actually know the exact, true parameters π_C, μ_C, and Σ_C, this procedure gives us the Bayes classifier and the Bayes optimal decision boundary. By contrast, when we estimate $\hat{\pi}_C$, $\hat{\mu}_C$, and $\hat{\Sigma}_C$ from data, this procedure is the QDA algorithm. We hope the QDA classifier will approximate the Bayes classifier. Sometimes in our textbooks, you will see examples where they plot both the Bayes optimal decision boundary and the decision boundary computed by a learning algorithm. (See the figure two pages forward.) When you see that, the authors know the exact, true probability distributions because they have chosen them and written a program that produces synthetic data from those distributions. With data from the real world, you cannot know the Bayes optimal decision boundary.]

Multi-class QDA:



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. Decision fn is
[Once again, the quadratic terms cancel each other out so the decision function is linear and the decision boundary is a hyperplane.]

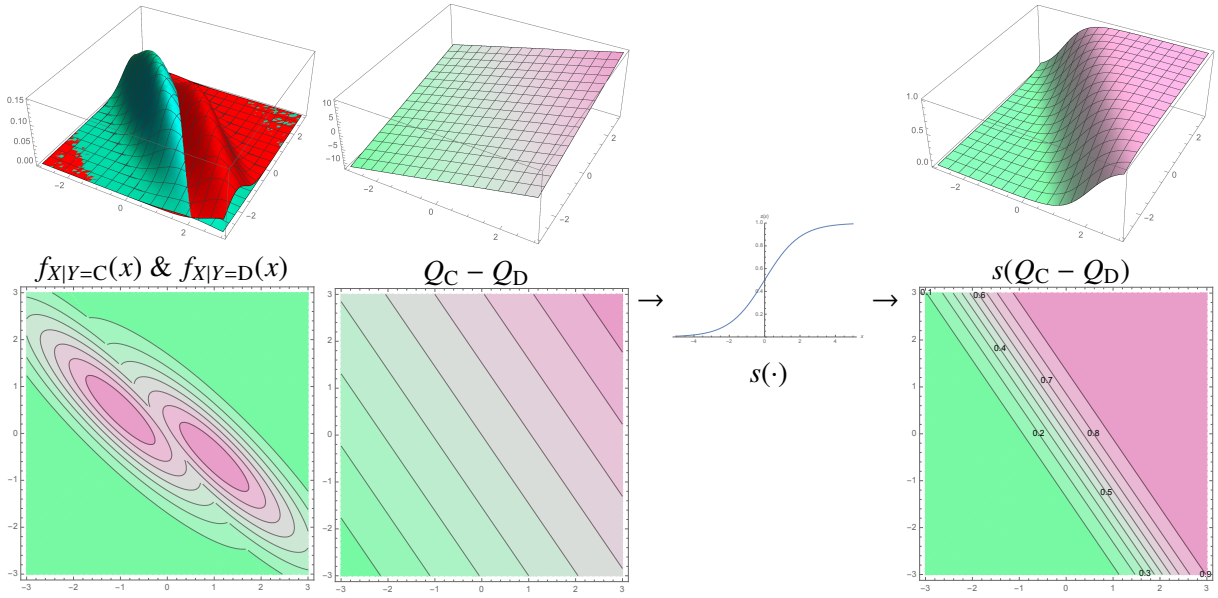
$$Q_C(x) - Q_D(x) = \underbrace{(\mu_C - \mu_D)^\top \Sigma^{-1}}_{w^\top} x \underbrace{- \frac{\mu_C^\top \Sigma^{-1} \mu_C - \mu_D^\top \Sigma^{-1} \mu_D}{2} + \ln \pi_C - \ln \pi_D}_{+\alpha}.$$

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

Multi-class LDA: choose class C that maximizes the linear discriminant fn

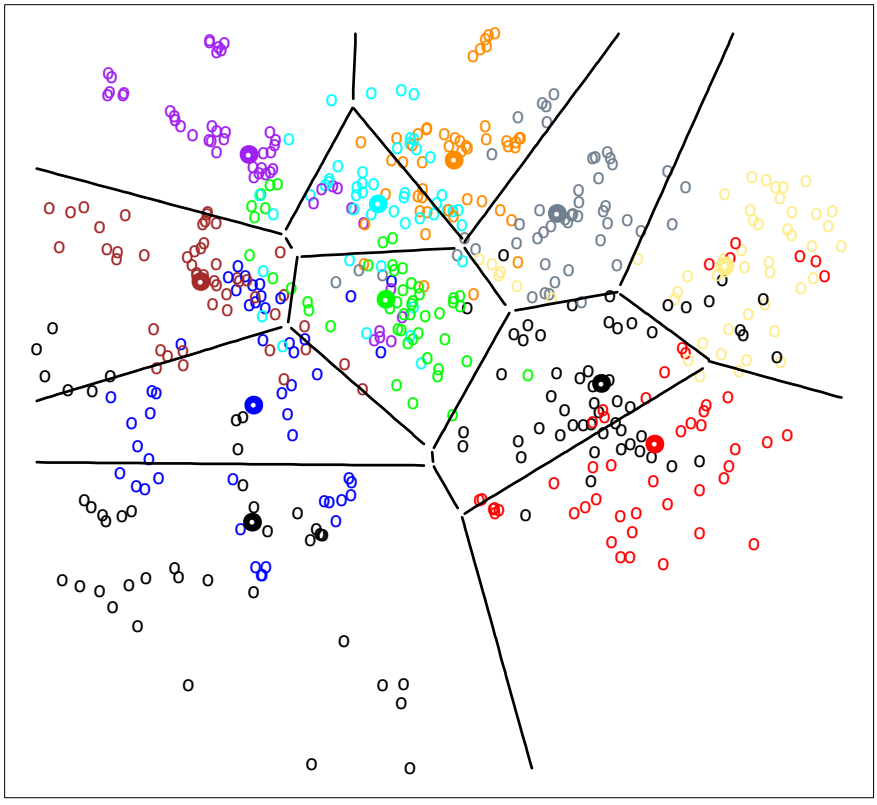
$$\mu_C^\top \Sigma^{-1} x - \frac{\mu_C^\top \Sigma^{-1} \mu_C}{2} + \ln \pi_C. \qquad \text{[works for any \# of classes]}$$

[Note that we use a linear solver to efficiently compute $\mu_C^\top \Sigma^{-1}$ just once, so the classifier can evaluate test points quickly.]



ldaaniso3d.pdf, ldaanisocontour.pdf, ldaanisodiff3d.pdf, ldaanisodiffcontour.pdf, logistic.pdf, ldaanisoposterior3d.pdf, ldaanisoposteriorcontour.pdf

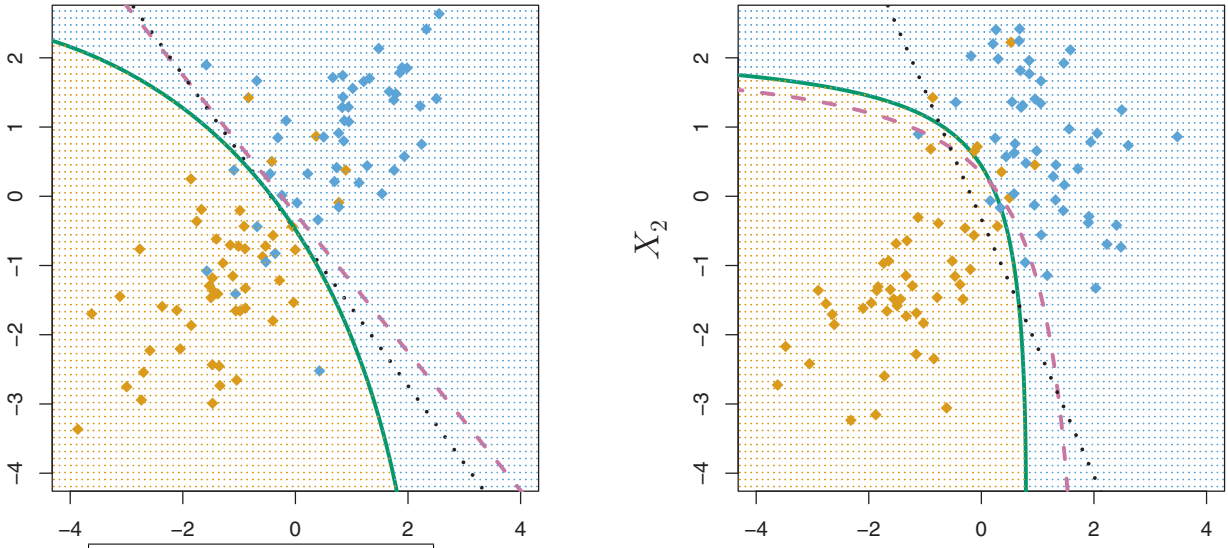
[(Show this figure on a separate "whiteboard.") In LDA, the decision boundary is always a hyperplane. Note that Mathematica messed up the top left plot a bit; there should be no red in the left corner, nor blue in the right corner.]



LDAdata.pdf (ESL, Figure 4.11) [An example of LDA with messy data. The real-world distributions almost surely aren't Gaussians, but LDA still works reasonably well.]

Notes on QDA/LDA

- For 2 classes,
  - LDA has $d + 1$ parameters (w, α);
    [. . . in the decision function. We estimated more statistical parameters than that, but only the degrees of freedom of the decision function matter for diagnosing underfitting or overfitting.]
  - QDA has $\frac{d(d+3)}{2} + 1$ params;
  - LDA more likely to underfit;
  - QDA more likely to overfit. [The danger is much bigger when the dimension d is large.]



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

- QDA on data doesn't find **true** optimum Bayes classifier.
  - estimate distributions from finite data.
  - real-world data not perfectly Gaussian.
- Changing priors or loss = adding constants to discriminant fns.
  [So it's very easy. In the 2-class case, it's equivalent to changing the isovalue . . .]
- Posterior gives decision boundaries for 10% probability, 50%, 90%, etc.
  choosing isovalue = probability p is equivalent to
  - choosing $\pi_C = 1 - p$, $\pi_D = p$; OR
  - choosing asymmetrical loss p for false positive, $1 - p$ for false negative.
- With added features, LDA can give nonlinear boundaries; QDA nonquadratic.

[LDA & QDA are the best method in practice for many applications. In the STATLOG project, either LDA or QDA were among 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.]

Some Terms

Let X be $n \times d$ design matrix of sample pts
Each row i of X is a sample pt $X_i^\top$.

[Now I'm using capital X as a matrix instead of a random variable vector. I'm treating X_i as a column vector to match the standard convention for multivariate PDFs like the Gaussian, but $X_i^\top$ is a row of X.]

centering X: subtracting $\hat{\mu}^\top$ from each row of X. $\quad X \to \dot{X}$
[$\hat{\mu}^\top$ is the mean of all the rows of X. Now the mean of all the rows of $\dot{X}$ is zero.]

Let R be drawn from uniform distribution on sample pts. Sample covariance matrix is

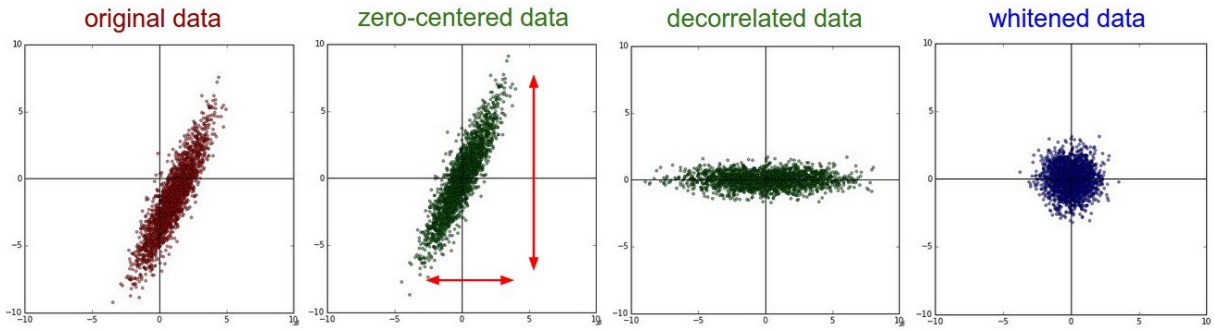
$$\text{Var}(R) = \frac{1}{n}\dot{X}^\top \dot{X}.$$

[This is the simplest way to remember how to compute a covariance matrix for QDA. Imagine you have a design matrix X_C that contains only the sample points of class C; then you have $\hat{\Sigma}_C = \frac{1}{n_C}\dot{X}_C^\top \dot{X}_C$.]

[When we have points from an anisotropic Gaussian distribution, sometimes it's useful to perform a linear transformation that maps them to an axis-aligned distribution, or maybe even to an isotropic distribution.]

decorrelating $\dot{X}$: applying rotation $Z = \dot{X}V$, where $\text{Var}(R) = V\Lambda V^\top$
[rotates the sample points to the eigenvector coordinate system]

Then $\text{Var}(Z) = \Lambda$. $\quad$ [Z has diagonal covariance. If $X_i \sim \mathcal{N}(\mu, \Sigma)$, then approximately, $Z_i \sim \mathcal{N}(0, \Lambda)$.]
[Proof: $\text{Var}(Z) = \frac{1}{n}Z^\top Z = \frac{1}{n}V^\top \dot{X}^\top \dot{X} V = V^\top \text{Var}(R) V = V^\top V \Lambda V^\top V = \Lambda$.]



original.jpg, centered.jpg, decorrelated.jpg, whitened.jpg

sphering $\dot{X}$: applying transform $W = \dot{X}\,\text{Var}(R)^{-1/2}$ $\quad$ [Recall that $\Sigma^{-1/2}$ maps ellipsoids to spheres.]

whitening X: centering + sphering, $X \to W$

Then W has covariance matrix I. $\quad$ [If $X_i \sim \mathcal{N}(\mu, \Sigma)$, then approximately, $W_i \sim \mathcal{N}(0, I)$.]

[Whitening input data is often used with other machine learning algorithms, like SVMs and neural networks. The idea is that some features may be much bigger than others—for instance, because they're measured in different units. SVMs penalize violations by large features more heavily than they penalize small features. Whitening the data before you run an SVM puts the features on an equal basis.]

[One nice thing about discriminant analysis is that whitening is built in.]

[Incidentally, what we've done here—computing a sample covariance matrix and its eigenvectors/values—is about 75% of an important unsupervised learning method called principal components analysis, or PCA, which we'll learn later in the semester.]

10 Regression, including Least-Squares Linear and Logistic Regression

REGRESSION aka Fitting Curves to Data

Classification: given point x, predict class (often binary)
Regression: given point 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 estimate a classifier; they also give us the probability that a particular prediction is correct. So QDA and LDA do regression on probability values.]

- Choose form of regression fn $h(x; w)$ with parameters w (h = hypothesis)
  - like decision 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., empirical risk = expected loss on data

Some regression fns:
(1) linear: $h(x; w, \alpha) = w \cdot x + \alpha$
(2) polynomial [equivalent to linear regression with added polynomial features]
(3) logistic: $h(x; w, \alpha) = s(w \cdot x + \alpha)$ 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 expression. So the logistic function seems to be a natural form for modeling certain probabilities. If we want to model posterior probabilities, sometimes we use LDA; but alternatively, we could skip fitting Gaussians to points, and instead just try to directly fit a logistic function to a set of probabilities.]

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

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 points are more important than others]
(d) $J(h) =$ (a), (b), or (c) $+\lambda \|w\|^2$ ℓ_2 penalized/regularized
(e) $J(h) =$ (a), (b), or (c) $+\lambda \|w\|_{\ell_1}$ ℓ_1 penalized/regularized

Some famous regression methods:

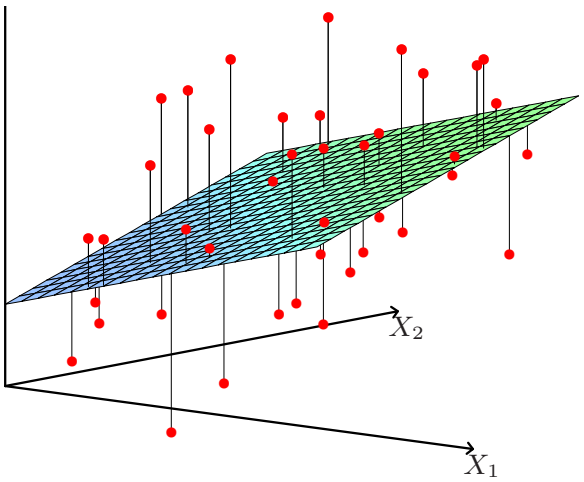
Least-squares linear regr.:	(1) + (A) + (a)	} quadratic cost; minimize w/calculus
Weighted least-squ. linear:	(1) + (A) + (c)	
Ridge regression:	(1) + (A) + (a) + (d)	
Lasso:	(1) + (A) + (a) + (e)	quadratic program
Logistic regr.:	(3) + (C) + (a)	convex cost; minimize w/gradient descent
Least absolute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

[I have given you several choices of regression function, several choices of loss function, and several choices of objective function. 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 consider 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 sample pts
y is n-vector of scalar labels

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

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

Usually $n > d$. [But not always.]

Recall fictitious dimension trick [from Lecture 3]: rewrite $h(x) = x \cdot w + \alpha$ as

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

Now X is an $n \times (d+1)$ matrix; w is a $(d+1)$-vector. [We've added a column of all-1's to the end of X.]
[We rewrite the optimization problem above:]

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

Optimize by calculus:

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

If $X^\top X$ is singular, problem is underconstrained.
[. . . because the sample points all lie on a common subspace (through the origin).]
[Notice that $X^\top X$ is always positive semidefinite, but not always positive definite.]

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!]

[We never compute X^+ directly, but we are interested in the fact that w is a linear transformation of y.]
[X is usually not square, so X can't have an inverse. However, every X has a pseudoinverse X^+, and if $X^\top X$ is invertible, 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_i are $\hat{y}_i = w \cdot X_i \quad \Rightarrow \quad \hat{y} = Xw = XX^+ y = Hy$
where $\underbrace{H}_{n\times n} = XX^+$ is called the hat matrix because it puts the hat on y.
[Ideally, H would be the identity matrix and we'd have a perfect fit, but if $n > d+1$, then H is singular.]

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

Disadvantages:
- Very sensitive to outliers, because errors are squared!
- Fails if $X^\top X$ is singular. [Which means the problem is underconstrained, has multiple solutions.]

[In discussion section 6, we'll address how to handle the underconstrained case where $X^\top X$ is singular.]

[Apparently, least-squares linear regression was first posed and solved in 1801 by the great mathematician Carl Friedrich Gauss, who used least-squares regression 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].

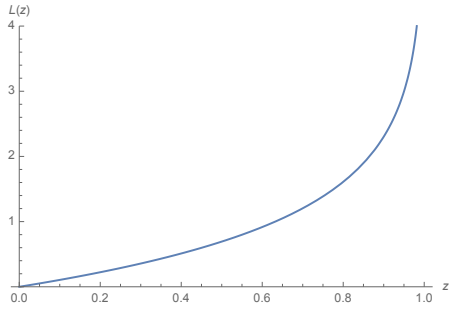
Usually used for classification. The input y_i's *can* be probabilities,
but in most applications they're all 0 or 1.

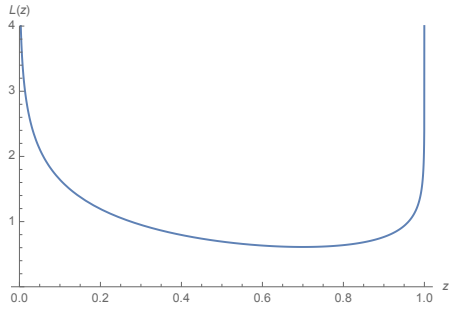
QDA, LDA: generative models
logistic regression: discriminative model
[We've learned from LDA that in classification, the posterior probabilities are often modeled well by a logistic function. So why not just try to fit a logistic function directly to the data, skipping the Gaussians?]

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

Find w that minimizes

$$J = \sum_{i=1}^{n} L(s(X_i \cdot w), y_i) = -\sum_{i=1}^{n} \Big(y_i \ln s(X_i \cdot w) + (1 - y_i) \ln (1 - s(X_i \cdot w)) \Big).$$

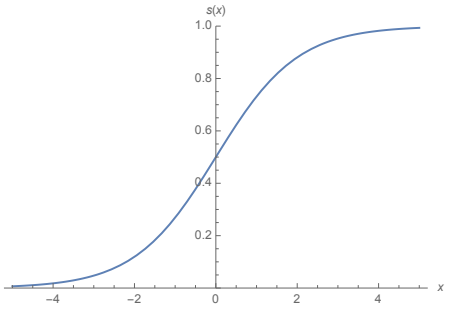
$L(\hat{y}, 0)$ 

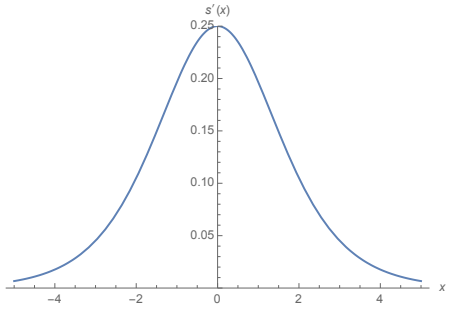
$L(\hat{y}, 0.7)$ 

logloss0.pdf, loglosspt7.pdf [Plots of the loss $L(\hat{y}, y)$ for $y = 0$ (left) and $y = 0.7$ (right). As you might guess, the left function is minimized at $\hat{y} = 0$, and the right function is minimized at $\hat{y} = 0.7$. These loss functions are always convex.]

$J(w)$ is convex! Solve by gradient descent.
[To do gradient descent, 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 $s_i = s(X_i \cdot w)$

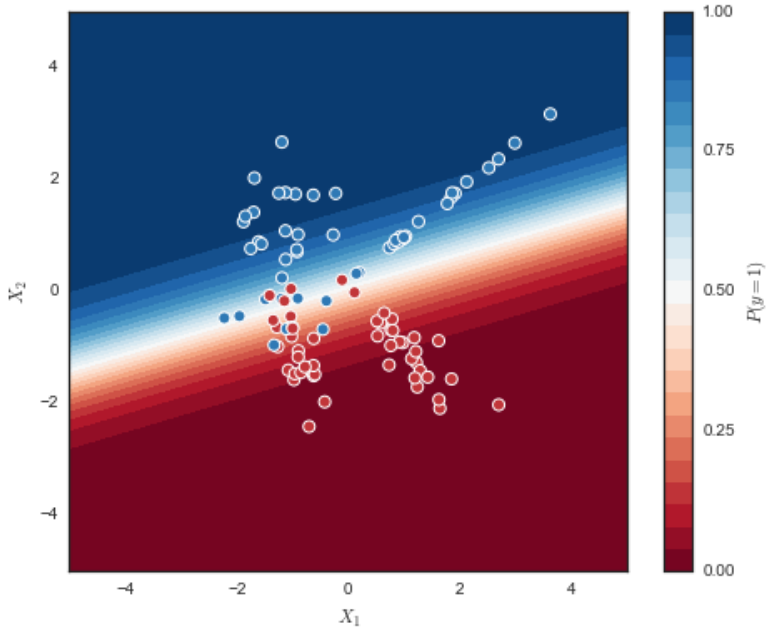
$$\begin{aligned} \nabla_w J &= -\sum \left(\frac{y_i}{s_i} \nabla s_i - \frac{1 - y_i}{1 - s_i} \nabla s_i \right) \\ &= -\sum \left(\frac{y_i}{s_i} - \frac{1 - y_i}{1 - s_i} \right) s_i (1 - s_i)\, X_i \\ &= -\sum (y_i - s_i)\, X_i \\ &= -X^\top (y - s(Xw)) \qquad \text{where } s(Xw) = \begin{bmatrix}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

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

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

[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.]

If sample pts are linearly separable and $w \cdot x = 0$ separates them (with decision boundary touching no pt), scaling w to have infinite length causes $s(X_i \cdot w) \to 1$ for a pt i in class C, $s(X_i \cdot w) \to 0$ for a pt not in class C, and $J(w) \to 0$ [in the limit as $\|w\| \to \infty$].

[Moreover, making w grow extremely large is the only way to get the cost function J to approach zero.]

Therefore, logistic regression always separates linearly separable pts!

[In this case, the cost function $J(w)$ has no finite local minimum, but gradient descent will "converge" to a solution, in the sense that the cost J will get arbitrarily close to zero, though of course the weight vector w will never become infinitely long. Mathematically speaking, w doesn't converge at all—it diverges—though $J(w)$ does converge to zero.]

[A 2018 paper by Soudry, Hoffer, Nacson, Gunasekar, and Srebro shows that gradient descent applied to logistic regression eventually converges to the maximum margin classifier, but the convergence is very, very slow. A practical logistic regression solver should use a different optimization algorithm.]

11 Polynomial and Weighted Regression; Newton's Method; ROC Curves

LEAST-SQUARES POLYNOMIAL REGRESSION

Replace each X_i with feature vector $\Phi(X_i)$ with all terms of degree $0 \ldots p$

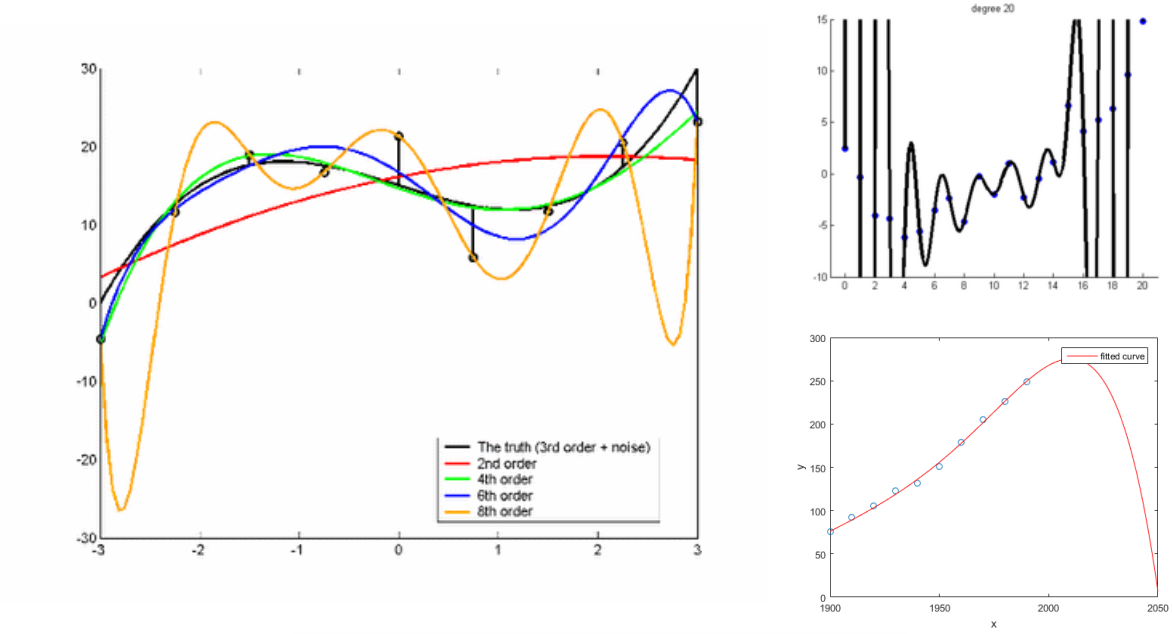
e.g., $\Phi(X_i) = [X_{i1}^2 \quad X_{i1}X_{i2} \quad X_{i2}^2 \quad X_{i1} \quad X_{i2} \quad 1]^\top$

[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} and X_{i2}.]

Otherwise just like linear or logistic regression.

Log. reg. + quadratic features = same form of posteriors as QDA.

Very easy to overfit!

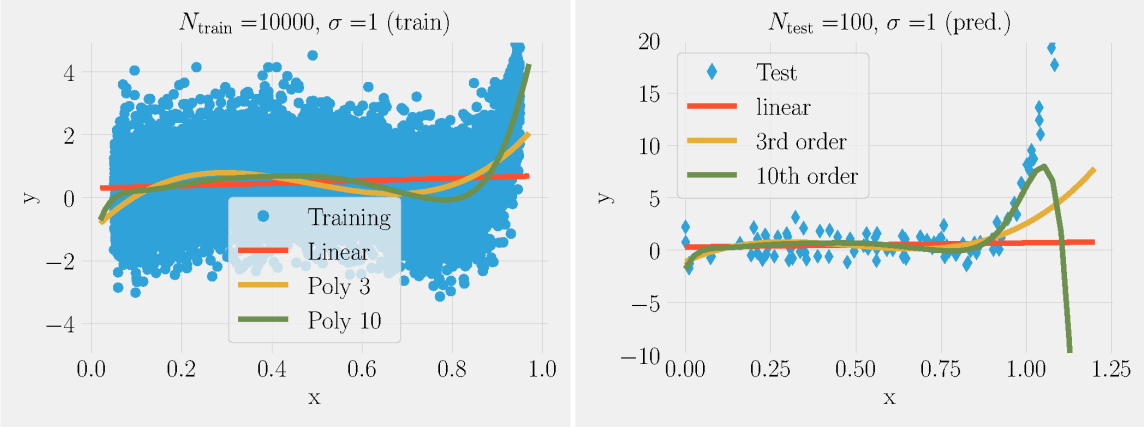


overunder.png, degree20.png, UScensusquartic.png

[Here are 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-2 curve (red) underfits and the degree-6 and 8 curves (blue, yellow) overfit the noise and oscillate. 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. It takes a great deal of densely spaced data to tame the oscillations in a high degree curve, and there isn't nearly enough data here.]

[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. As a general rule, extrapolation is much harder than interpolation. The k-nearest neighbor classifier is one of the few that does extrapolation decently without occasionally returning crazy values.]



order10extrap.pdf [From Mehta, Wang, Day, Richardson, Bukov, Fisher, and Schwab, "A High-Bias, Low-Variance Introduction to Machine Learning for Physicists."]

[This example shows that a fitted degree-10 polynomial (green) can be tamed by using a very large amount of training data (left), even if the training data is noisy. The training data was generated from a different degree-10 polynomial, with noise added. On the right, the same curves are plotted, but the blue diamonds are test points, some of which go outside the range of the training data. We see that the degree-10 regression does decent extrapolation for a short distance, albeit only because the original data was also from a degree-10 polynomial.]

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 pt a weight ω_i; collect ω_i's in $n \times n$ diagonal matrix Ω.

Greater $\omega_i \rightarrow$ work harder to minimize $(\hat{y}_i - y_i)^2$ recall: $\hat{y} = Xw$ [$\hat{y}_i$ is predicted label for X_i]

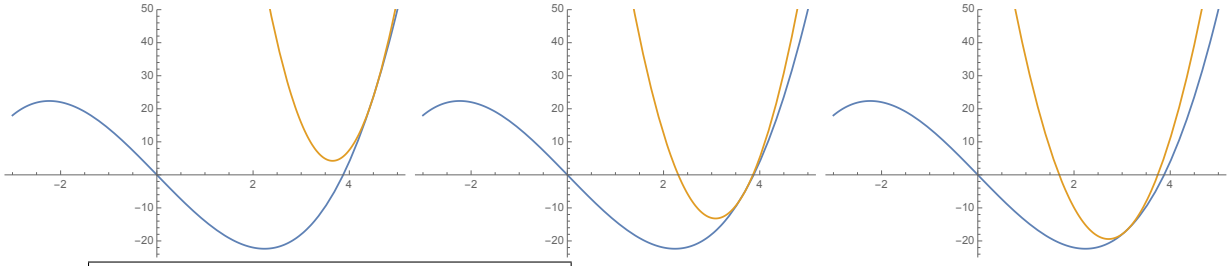
Find w that minimizes $(Xw - y)^\top \Omega (Xw - y) = \sum_{i=1}^{n} \omega_i (X_i \cdot w - y_i)^2$.

[As with ordinary least-squares regression, we find the minimum by setting the gradient to zero, which leads us to the normal equations.]
Solve for w in normal equations: $X^\top \Omega X w = X^\top \Omega y$

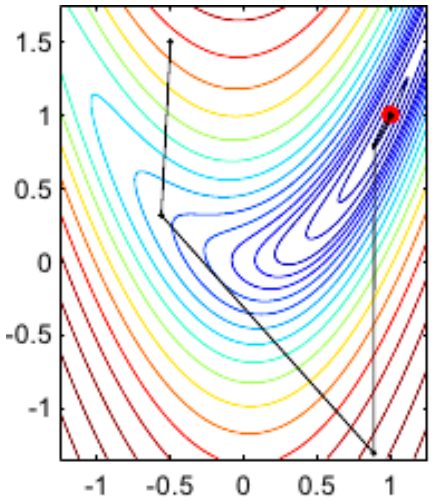
NEWTON'S METHOD

Iterative optimization method for smooth fn $J(w)$.
Often much faster than gradient descent. [We'll use Newton's method 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 at v.

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

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

[This is an iterative update rule you can repeat until it converges to a solution. As usual, we probably don't want to compute a matrix inverse directly. It is faster to solve a linear system of equations, typically by Cholesky factorization or the conjugate gradient method.]

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 pts.
Starting pt must be "close enough" to desired critical pt.

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

[Newton's method is superior to gradient descent with a fixed step size for some optimization problems for at least two 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 choose a better descent direction.]

[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 weight spaces, but you would never use it for a neural network, 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 the Hessian is not even defined.]

LOGISTIC REGRESSION (continued)

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

$$\text{Recall: } s'(\gamma) = s(\gamma)\,(1 - s(\gamma)), \qquad s_i = s(X_i \cdot w), \qquad s = \begin{bmatrix} s_1 \\ s_2 \\ \vdots \\ s_n \end{bmatrix},$$

$$\nabla_w J(w) = -\sum_{i=1}^{n} (y_i - s_i)\, X_i = -X^\top (y - s)$$

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

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

Ω is +ve definite $\forall w \quad \Rightarrow \quad X^\top \Omega X$ is +ve semidefinite $\forall w \quad \Rightarrow \quad J$ is convex.
[The logistic regression cost function is convex, so Newton's method finds a globally optimal point if it converges at all.]

Newton's method:

$w \leftarrow 0$
repeat until convergence
  $e \leftarrow$ solution to normal equations $(X^\top \Omega X)\, e = X^\top (y - s)$ Recall: Ω, s are fns of w
  $w \leftarrow w + e$

[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. So we call it . . .]
An example of iteratively reweighted least squares.

[We need to be very careful with the analogy, though. The weights don't have the same meaning they had when we learned weighted least-squares regression, because there is no Ω on the right-hand side of $(X^\top \Omega X)\, e = X^\top (y - s)$. Contrary to what you'd expect, a *small* weight in Ω causes the Newton iteration to put *more* emphasis on a point when it computes e.]

[Misclassified points far from the decision boundary have the most influence on the step e, and correctly classified points far from the decision boundary have the least (because $y_i - s_i$ is small for such a point). Points near the decision boundary have medium influence. But if there are no misclassified points far from the decision boundary, then points near the decision boundary have most of the influence.]

[Here's one more idea for speeding up logistic regression.]
Idea: If n very large, save time by using a random subsample of the pts 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.]

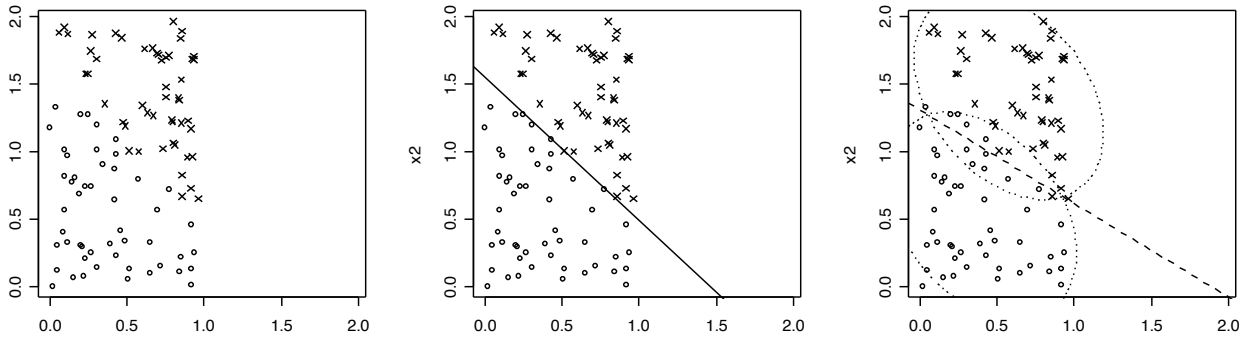
LDA vs. Logistic Regression

Advantages of LDA:

- For well-separated classes, LDA stable; log. reg. surprisingly unstable
- > 2 classes easy & elegant; log. reg. needs modifying (softmax regression) [see Discussion 6]
- LDA slightly more accurate when classes nearly normal, especially if n is small

Advantages of log. reg.:

- More emphasis on decision boundary; always separates linearly separable pts
  [Correctly classified points far from the decision boundary have a small effect on logistic regression—albeit a bigger effect than they have on SVMs—whereas misclassified points far from the decision boundary have the biggest effect. By contrast, LDA gives all the sample points equal weight when fitting Gaussians to them. Weighting points according to how badly they're misclassified is good for reducing training error, but it can also be bad if you want stability or insensitivity to bad data.]

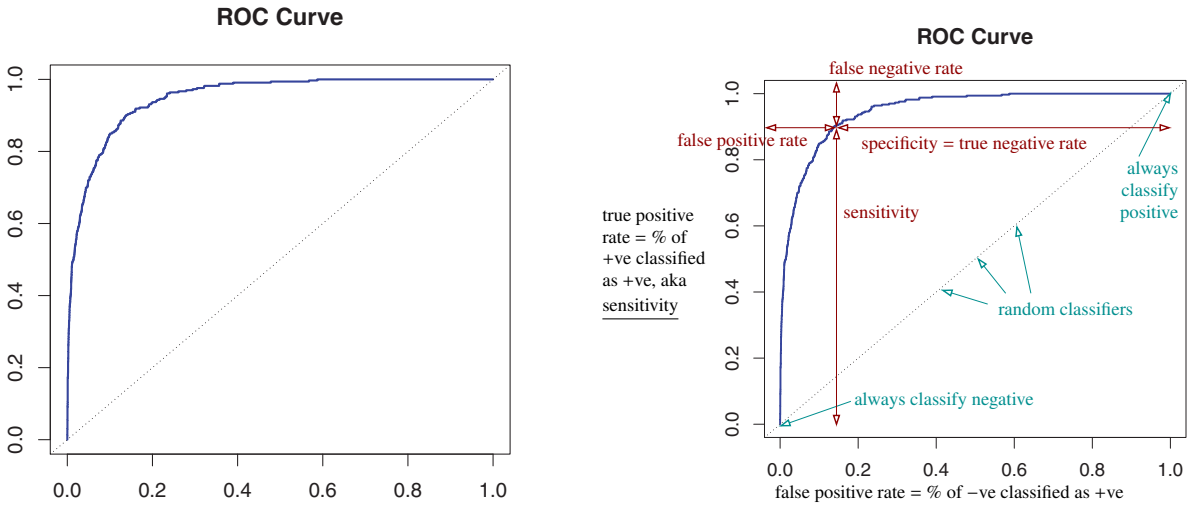


logregvsLDAuni.pdf [Logistic regression vs. LDA for a linearly separable data set with a very narrow margin. Logistic regression (center) always succeeds in separating linearly separable classes, because the cost function approaches zero for a maximum margin classifier. In this example, LDA (right) misclassifies some of the training points.]

- More robust on some non-Gaussian distributions (e.g., dists. w/large skew)
- Naturally fits labels between 0 and 1 [usually probabilities]

[When you use logistic regression with added quadratic features, you get a quadric decision boundary, just as you do with QDA. Based on what I've said here, do you think logistic regression with quadratic features gives you exactly the same classifier as QDA?]

ROC CURVES (for test sets)

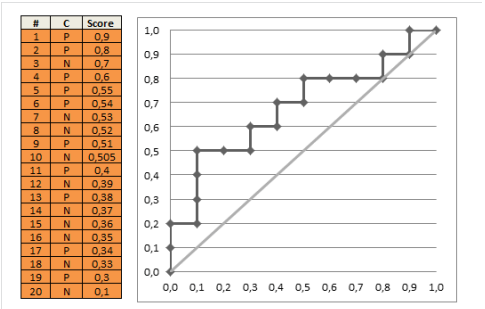


ROC.pdf

[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.
A ROC curve is a way to evaluate your classifier *after* it is trained.
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 false positives against false negatives. For our purposes, that knob is the posterior probability threshold for Gaussian discriminant analysis or logistic regression. However, neither axis of this plot is that knob.]

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 [1− sensitivity]
"specificity": horizontal distance from curve to right [1− false positive rate; "true negative rate"]
[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. For the random classifier, the area under the curve is $1/2$, so you'd better do better than that.]



#	C	Score
1	P	0,9
2	P	0,8
3	N	0,7
4	P	0,6
5	P	0,55
6	P	0,54
7	N	0,53
8	N	0,52
9	P	0,51
10	N	0,505
11	P	0,4
12	N	0,39
13	P	0,38
14	N	0,37
15	N	0,36
16	N	0,35
17	P	0,34
18	N	0,33
19	P	0,3
20	N	0,1

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.

[Close up, ROC curves are made of horizontal and vertical line segments (see the figure at right), as the test data is finite and there are only finitely many thresholds where some test point's classification changes.]

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 that 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. But why did we pick these cost functions? Today, let's take a step up 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 fn + random noise:

$$\forall X_i, \quad y_i = g(X_i) + \epsilon_i, \quad \epsilon_i \sim D', \quad D' \text{ has mean zero.} \quad [g = \text{“ground truth.”}]$$

[We are positing that reality is described by a "ground truth" function g. We don't know g, but g is not random; it represents a consistent relationship between X and y that we can estimate. We add to g 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 questionable 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 g.
Ideal approach: choose $h(x) = \underbrace{\mathrm{E}_Y[Y|X = x]} = g(x) + \mathrm{E}[\epsilon] = g(x)$.

[If this expectation exists at all, it partly justifies our model of reality. We can retroactively define g to be this expectation.]

Least-Squares Cost Function from Maximum Likelihood

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

Recall that log of normal PDF is

$$\ln f(y_i) = -\frac{(y_i - \mu_i)^2}{2\sigma^2} - \text{constant} \qquad \Leftarrow \mu_i = g(X_i)$$

& log likelihood is

$$\ell(g; X, y) = \ln\left(f(y_1)\, f(y_2) \cdots f(y_n)\right) = \ln f(y_1) + \ldots + \ln f(y_n) = -\frac{1}{2\sigma^2} \sum (y_i - g(X_i))^2 - \text{constant}.$$

Takeaway: Max likelihood on "parameter" $g \Rightarrow$ choose a g that minimizes $\sum (y_i - g(X_i))^2$.

[We treat g as a "distribution parameter." If the noise is normally distributed, maximum likelihood tells us to estimate g by least-squares regression.]
[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 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 of noise, for example, least-squares isn't a great choice.]

Empirical Risk

The risk for hypothesis h is expected loss $R(h) = \mathrm{E}[L]$ over all (X, Y) in some joint distribution.
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 joint probability distribution for X and 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 approximate the distribution in a very crude way: we pretend that the sample points *are* the distribution.]

Empirical distribution: the **discrete** uniform distribution over the sample pts

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 that $\hat{R}$ is only a cheap approximation of the true, unknown statistical risk R we really want to minimize. Often, this is the best we can do. For many but not all distributions, the empirical risk converges to the true risk in the limit as $n \to \infty$. Choosing h that minimizes $\hat{R}$ is called empirical risk minimization.]

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

Logistic Loss from Maximum Likelihood

What cost fn should we use for probabilities?

Actual probability pt X_i is in class C is y_i; predicted prob. is $h(X_i)$.

Imagine β duplicate copies of X_i, with $y_i\beta$ in class C, and $(1 - y_i)\beta$ not.
[The size of β isn't very important, but imagine that $y_i\beta$ and $(1 - y_i)\beta$ are both integers for all i. If y_i is irrational, approximate it with a very close rational number.]

[Let's use maximum likelihood estimation to choose the hypothesis most likely to generate these labels for these sample points. The following likelihood is the probability of generating these labels in a particular fixed order.]

Likelihood is $\mathcal{L}(h; X, y) = \prod_{i=1}^{n} h(X_i)^{y_i\beta}(1 - h(X_i))^{(1-y_i)\beta}$

$$\begin{aligned}
\text{Log likelihood is } \ell(h) &= \ln \mathcal{L}(h) \\
&= \beta \sum_i \Big(y_i \ln h(X_i) + (1 - y_i) \ln(1 - h(X_i)) \Big) \\
&= -\beta \sum \text{logistic loss fn } L(h(X_i), y_i).
\end{aligned}$$

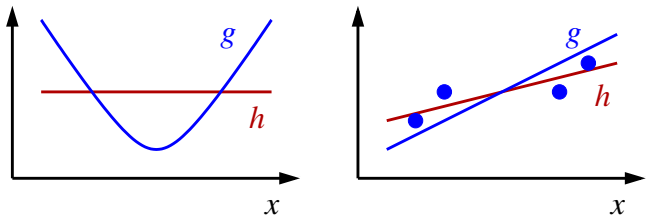
Takeaway: Max likelihood ⇒ minimize $\sum$ logistic losses.
[So the principle of maximum likelihood explains where the weird logistic loss function comes from.]

THE BIAS-VARIANCE DECOMPOSITION

There are 2 sources of error in a hypothesis h:

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

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



biasvar.pdf [Draw this figure by hand. At left, the error due to bias: a linear hypothesis h just can't fit a degree-2 ground truth g well. At right, the error due to variance: although h could fit g perfectly, the noise in the data misleads it.]

Model: $X_i \sim D$, $\epsilon_i \sim D'$, $y_i = g(X_i) + \epsilon_i$ [remember that D' has mean zero]
fit hypothesis h to X, y
Now h is a random variable; i.e., its weights are random

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

Risk fn when loss = squared error:

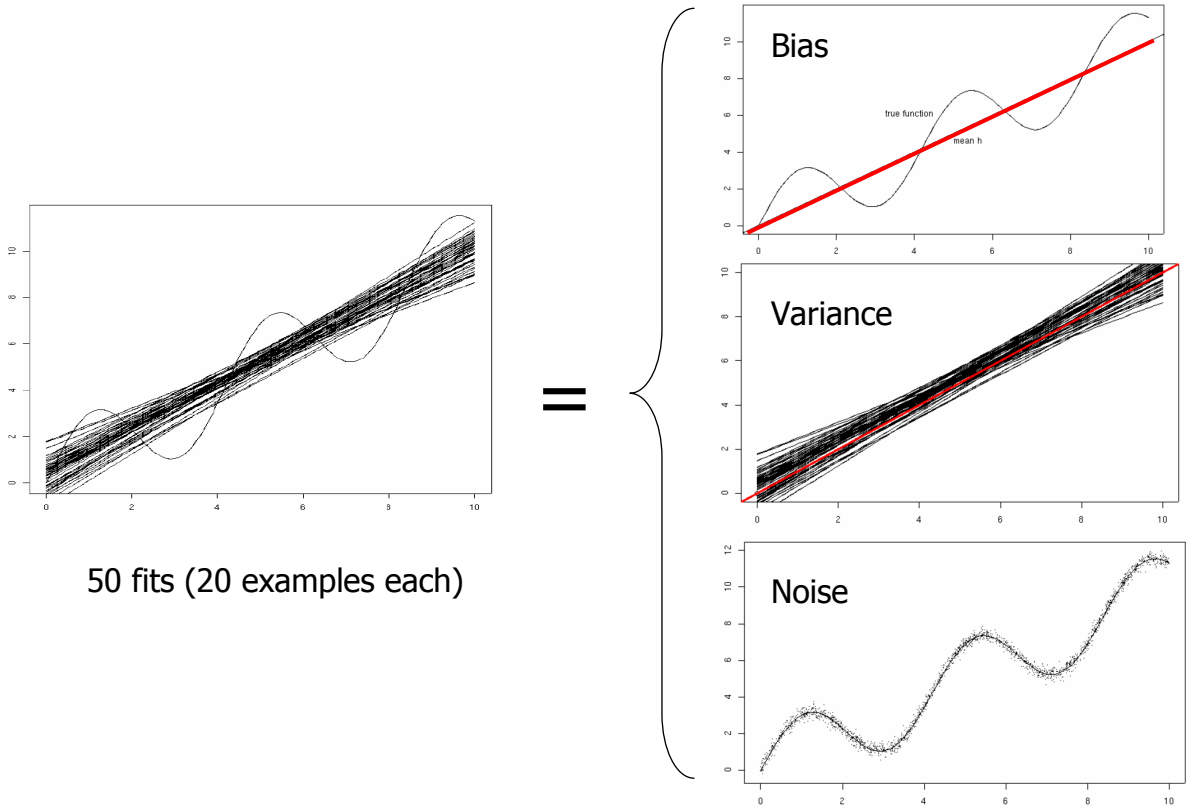
$$R(h) = \mathrm{E}[L(h(z), \gamma)]$$

↑ take expectation over possible training sets X, y & 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 a joint probability distribution. 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.]

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

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



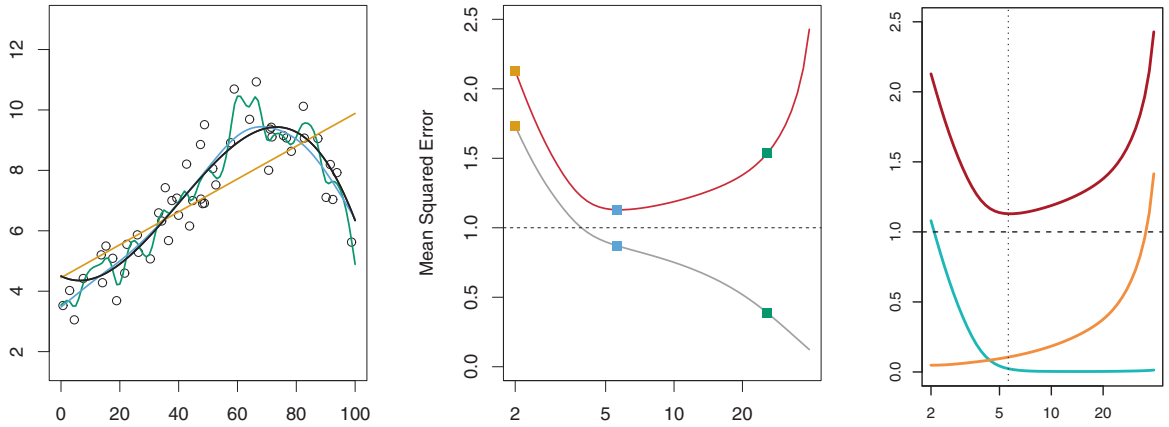
bvn.pdf [In this example, we're trying to fit a sine wave with lines, which obviously aren't going to be accurate. At left, we have generated 50 different hypotheses (lines). Each line was generated from 20 random training points by least-squares linear regression. At upper right, the black curve is the ground truth function g, and the red line is the *expected hypothesis*—an average over infinitely many hypotheses. We see that at most points in feature space, the bias (difference between the black and red curves) is large, because lines don't fit sine waves well. However, the bias is small at some points—where the sine wave crosses the red line. At center right, the variance is the expected squared difference between a random hypothesis (black line) and the expected hypothesis (red line) at an arbitrary point in feature space. At lower right, the irreducible error is the expected squared difference between a random test point's noisy label and the sine wave. The irreducible error is optional; it only makes sense to talk about it if we can make real-world measurements at test points.]

This is pointwise version [of the bias-variance decomposition.]
Mean version: let $z \sim D$ be random variable; take mean over D of bias^2, variance.

[So you can decompose one test point's error into these three numbers, or you can decompose the error of the hypothesis over its entire range into three numbers, which tells you roughly 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
- Most overfitting caused by too much variance
- Training error reflects bias but not much 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 g exactly, for many distributions bias $\to 0$ as $n \to \infty$
- If h cannot fit g 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 [don't add a feature unless it reduces bias more]
- 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)
- We can't precisely measure bias or variance of real-world data
  [because we cannot know g exactly and our noise model might be wrong]
- But we can test learning algs by choosing g & 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 6 degrees because the bias decreases, but for more degrees of freedom the test error increases because the variance increases.]

[The bias-variance decomposition is sometimes called the "bias-variance trade-off," because sometimes you see a U-shaped curve like this, for some hyperparameter like polynomial degree or C in SVMs. But that's misleading; it's not really a trade-off. Sometimes both bias and variance are very high; sometimes both are very low. If you try to fit 30 periods of a sine wave with a degree-10 polynomial fit to 15 points sampled from the sine wave, both your bias and variance will be very high. Then you're underfitting and overfitting at the same time! They're not opposites. There are neural networks for image classification with 99% accuracy on test sets, so bias and variance are both admirably low. Always we seek models that can fit the ground truth well, but aren't easily perturbed by noise in the data.]

Example: Least-Squares Linear Reg.

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

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

Lin. reg. computes weights

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

BIAS is $|\mathrm{E}[h(z)] - g(z)| = |\mathrm{E}[w^\top z] - v^\top z| = |z^\top \mathrm{E}[w - v]| = |z^\top \mathrm{E}[X^+ e]| = 0$.
[$\mathrm{E}[X^+ e]$ is zero because X^+ and e are independent, and e's Gaussian PDF is symmetric around zero.]

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

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

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

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

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

If we choose coordinate system so D has mean zero, then $X^\top X \to n \,\mathrm{Var}(D)$ as $n \to \infty$, so for $z \sim D$,

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

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

[Why? With the eigendecomposition $\mathrm{Var}(D) = V \Lambda V^\top$, we have $\mathrm{E}[z^\top \mathrm{Var}(D)^{-1} z] = \mathrm{E}[\|\Lambda^{-1/2} V^\top z\|^2] = \sum_{i=1}^{d} \mathrm{E}[(v_i \cdot z)^2]/\lambda_i$. But as $z \sim D$, $\mathrm{E}[(v_i \cdot z)^2] = \mathrm{Var}[v_i \cdot z] = \lambda_i$, so $\mathrm{E}[z^\top \mathrm{Var}(D)^{-1} z] = d$. Approximating the covariance $\mathrm{Var}(D)$ with the sample covariance matrix gives $\mathrm{E}[z^\top (X^\top X)^{-1} z] \approx d/n$.]

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

13 Shrinkage: Ridge Regression, Subset Selection, and Lasso

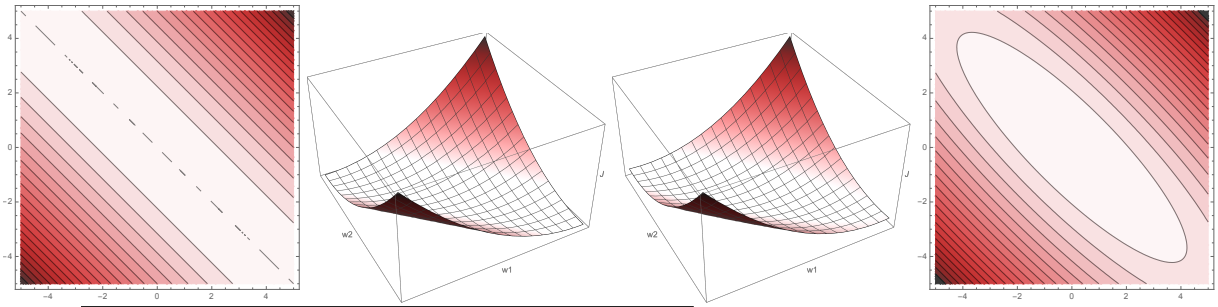
RIDGE REGRESSION aka Tikhonov Regularization

Least-squares linear regression + ℓ_2 penalized mean loss. (1) + (A) + (a) + (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 regularization term, aka a penalty term, for shrinkage: to encourage small $\|w'\|$. Why?

(1) Guarantees positive definite normal eq'ns; always unique solution.
[Standard least-squares linear regression yields singular normal equations when the sample points lie on a common hyperplane in feature space—for example, when $d > n$.]



lslrcontour.pdf, lslr.pdf, ridge.pdf, ridgecontour.pdf [The cost function $J(w)$ without and with regularization. This plot ignores the dimension of the bias term α.]

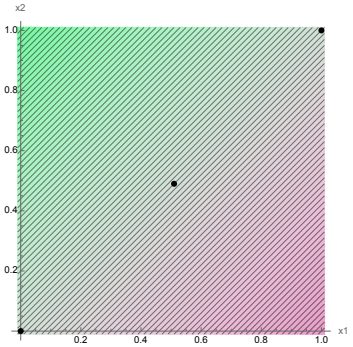
[At left, we see a cost function for least-squares regression, a positive semidefinite quadratic form. This cost function has many minima, and the regression problem is said to be ill-posed. By adding a small penalty term, we obtain a positive definite quadratic form (right), with one unique minimum. "Regularization" implies that we are turning an ill-posed problem into a well-posed problem.]

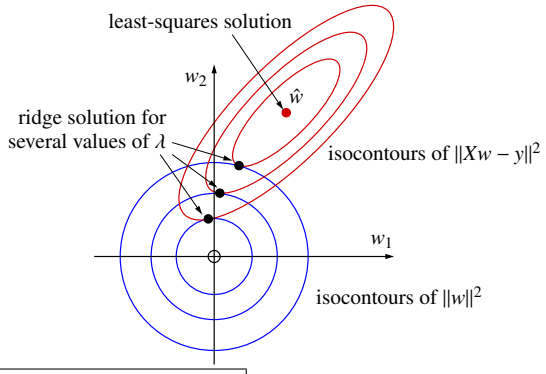
[That was the original motivation, but the next has become more important in machine learning . . .]

(2) Reduces overfitting by reducing variance. Why?

Example: Input $X_1 = (0, 0)$ with label 0; $X_2 = (1, 1)$ with label 0; $X_3 = (0.51, 0.49)$ with label 1.
Linear regr. gives $50x_1 - 50x_2$. [This linear function fits all three points exactly.]
Big weights!

[Weights this big would be justified if there were big differences between labels, or if there were small distances between points, but neither is true. Large weights imply that tiny changes in x can cause huge changes in y. Consider that the labels don't differ by more than 1 and the points are separated by distances greater than 0.7. So these disproportionately large weights are a sure sign of overfitting.]
So we penalize large weights.
[This use of regularization is closely related to the first one. When you have large variance and a lot of overfitting, it implies that your problem is *close to* being ill-posed, even though technically it might be well-posed.]





ridgeterms2.pdf (redrawing of ISL, Figure 6.7) [In this plot of weight space, for simplicity, we're not using a bias term α (we set it to zero). $\hat{w}$ is the least-squares solution. The red ellipses are isocontours of $\|Xw - y\|^2$. The blue circles are isocontours of $\|w\|^2$, centered at the origin. The ridge regression 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 shrinks w and helps to reduce overfitting.]

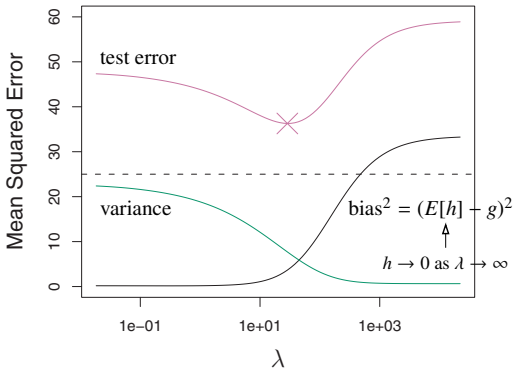
Setting $\nabla J = 0$ gives normal eq'ns

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

where I' is identity matrix w/bottom right set to zero. [Don't penalize the bias term α.]
[Don't worry; $X^\top X + \lambda I'$ is always positive definite for $\lambda > 0$, assuming X ends with a column of 1's.]

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

Increasing λ $\Rightarrow$ more regularization; smaller $\|w'\|$
Recall [from the previous lecture] our data model $y = Xv + e$, where e is noise.
Variance of ridge regr. at test pt z 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.]

[The test error as a function of λ is a U-shaped curve. We find the bottom by validation. Regularization is intended to reduce the variance, but this method of regularization also increases the bias.]

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. [For example, if you use polynomial features, you could use different penalties for monomials of different degrees.]

Bayesian Justification for Ridge Reg.

Assign a prior probability on w': $w' \sim \mathcal{N}(0, \varsigma^2)$, with PDF $f(w') \propto e^{-\|w'\|^2/(2\varsigma^2)}$
[This prior probability says that we think weights close to zero are more likely to be correct.]
Apply MLE to maximize the posterior prob.

$$\begin{aligned}
\text{Bayes' Theorem: posterior } f_{W|X,Y}(w) &= \frac{f_{Y|X,W}(y)\, f(w')}{f_{Y|X}(y)} \\
\text{Maximize log posterior} &= \ln f_{Y|X,W}(y) + \ln f(w') - \text{const} \\
&= -\text{const}\, \|Xw - y\|^2 - \text{const}\, \|w'\|^2 - \text{const} \\
&\Rightarrow \text{Minimize } \|Xw - y\|^2 + \lambda \|w'\|^2
\end{aligned}$$

[We are treating w and y as random variables, but X as a fixed constant—it's not random.]
This method (using MLE, but maximizing posterior) is called maximum *a posteriori* (MAP).
[A prior probability on the weights is another way to understand regularizing ill-posed problems.]

FEATURE SUBSET SELECTION

[Some of you may have noticed as early as Homework 1 that you can sometimes get better performance on a spam classifier simply by dropping some useless features.]

All features increase variance, but not all features reduce bias.
Idea: Identify poorly predictive features, ignore them (weight zero).
Less overfitting, smaller 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. [Train one classifier per subset.]
Choose best classifier by (cross-)validation. Slow.

[Obviously, best subset selection isn't feasible 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 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 validation errors start increasing (due to overfitting) instead of decreasing. At each outer iteration, inner loop tries every feature & chooses the best by 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 validation error.
Also trains $O(d^2)$ models.

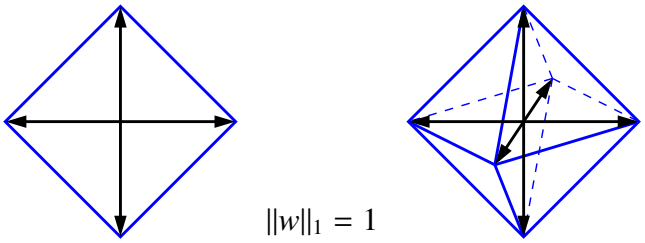
[Forward stepwise is a better choice when you suspect only a few features will be good predictors; e.g., spam. Backward stepwise is better when most features are important. If you're lucky, you'll stop early.]

LASSO (Robert Tibshirani, 1996)

Least-squares linear regression + ℓ_1 penalized mean loss. (1) + (A) + (a) + (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.]

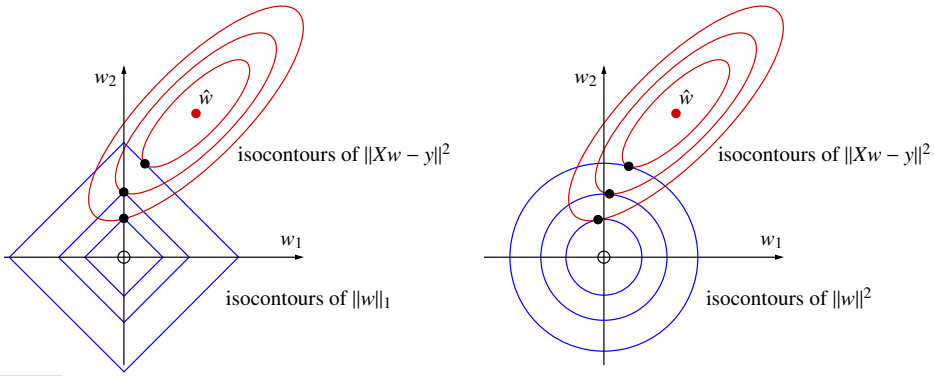
Find w that minimizes $\|Xw - y\|^2 + \lambda \|w'\|_1$ where $\|w'\|_1 = \sum_{i=1}^{d} |w_i|$. (Don't penalize α.)

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 & negative unit coordinate vectors.



[Draw this figure by hand crosspolys.pdf]

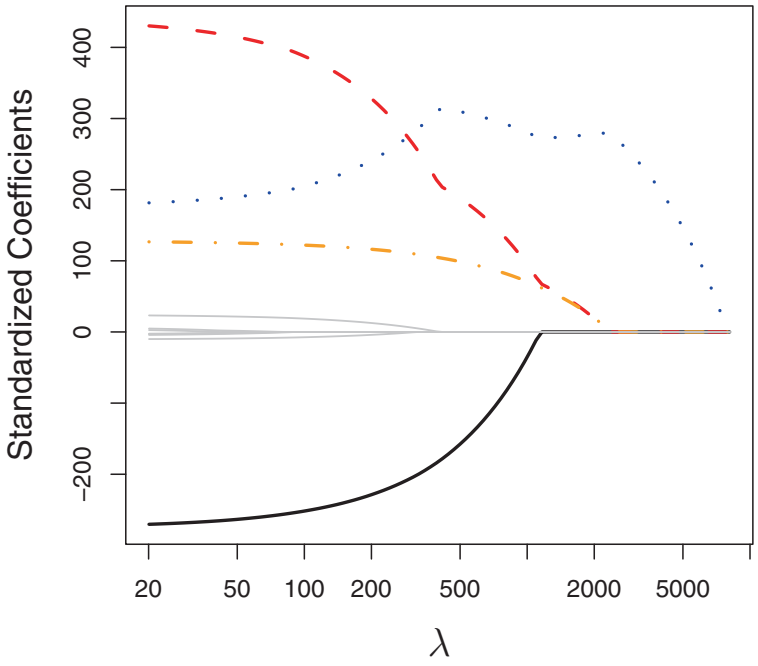
[You get larger and smaller cross-polytope isosurfaces by scaling these.]



lassoridge2.pdf [Isocontours of the terms of the objective function for the Lasso appear at left. Compare with the ridge regression isocontours at right.]

[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 is that for large values of λ, the red isocontour touches just the tip of a diamond. Then the weight w_1 gets set to zero. That's what we want to happen to features that don't have enough predictive power. For small values of λ, the red isosurface just barely touches a side of a diamond instead of a tip of the diamond, and no weight gets set to zero.]

[When you go to higher dimensions, you might have several weights set to zero. For example, in 3D, if the red isosurface just touches a sharp vertex of a cross-polytope, two of the three weights get set to zero. If it just touches a sharp edge of a cross-polytope, one weight gets set to zero. If it just touches a flat side of a 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 validation.]

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

[Lasso can be reformulated as a quadratic program, but it's a quadratic program with 2^d constraints, because a d-dimensional cross-polytope has 2^d facets. In practice, special-purpose optimization methods have been developed for Lasso. I'm not going to teach you one, but 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 Lasso.]

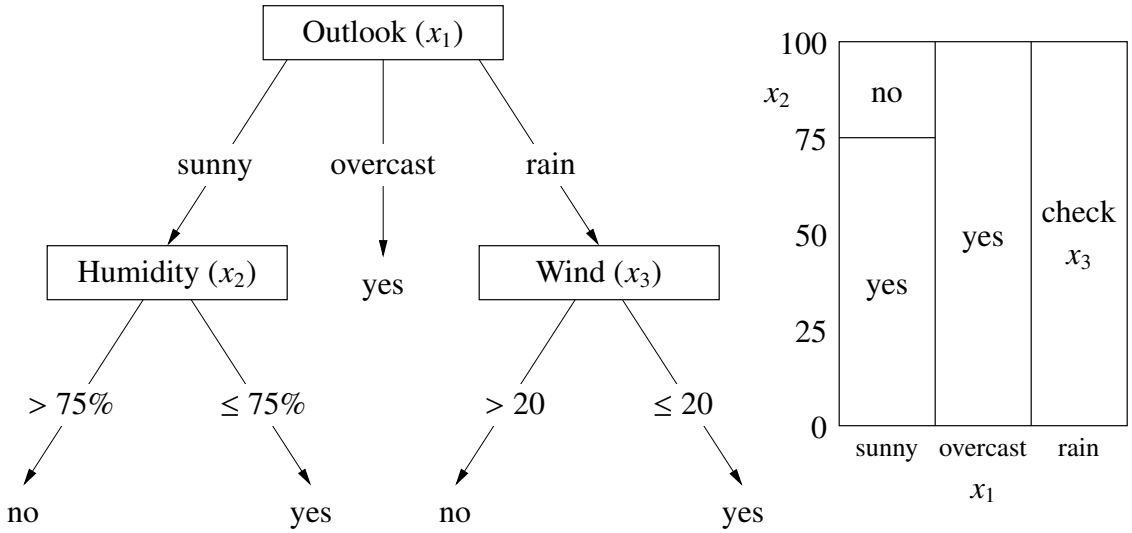
14 Decision Trees

DECISION TREES

Nonlinear method for classification and regression.

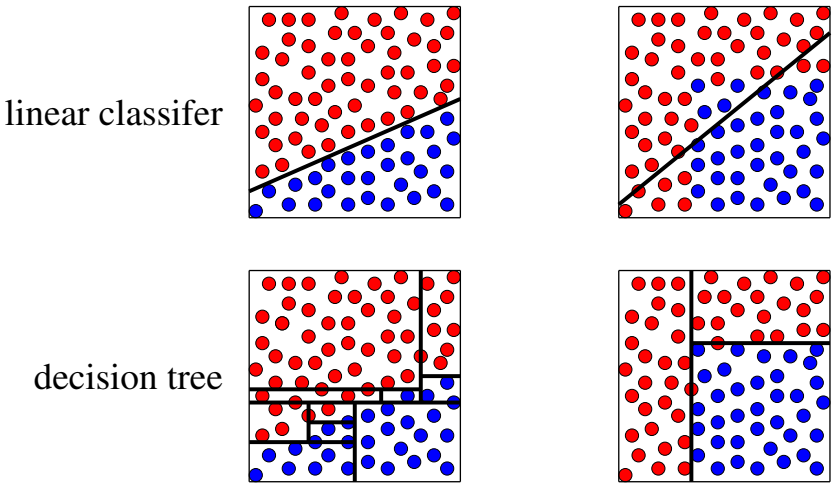
Uses tree with 2 node types:

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



[Draw this by hand. dectree.pdf Deciding whether to go out for a picnic.]

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



treelinearcompare2.pdf (redrawing of ISL, Figure 8.7) [Comparison of linear classifiers vs. decision trees 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 X be $n \times d$ design matrix; $y \in \mathbb{R}^n$ be labels.
Let $S \subseteq \{1, 2, \ldots, n\}$ be set of sample point indices.
Top-level call: $S = \{1, 2, \ldots, n\}$.

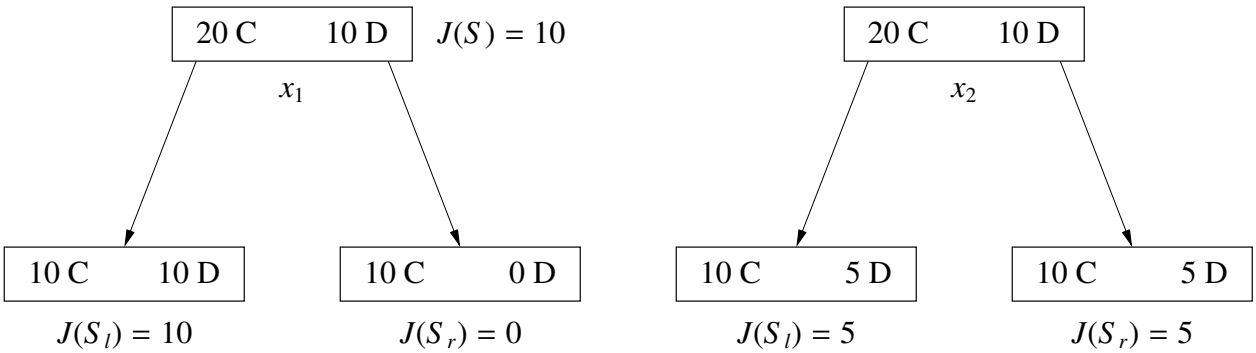
```
GrowTree(S)
    if (y_i = C for all i ∈ S and some class C) then {
        return new leaf(C)                    [We say the leaves are pure]
    } else {
        choose best splitting feature j and splitting value β     (*)
        S_l = {i ∈ S : X_ij < β}              [Or you could use ≤ and >]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

[Here, I'm using the vertical bars $|\cdot|$ 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: $J(S_l) + J(S_r) = 10$ for both splits, but left split is much better. Weighted avg prefers right split!

[There are many different splits that all have the same total cost. We want a cost function that better distinguishes between them.]

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 $-\log_2 p_\mathrm{C}$. [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.]

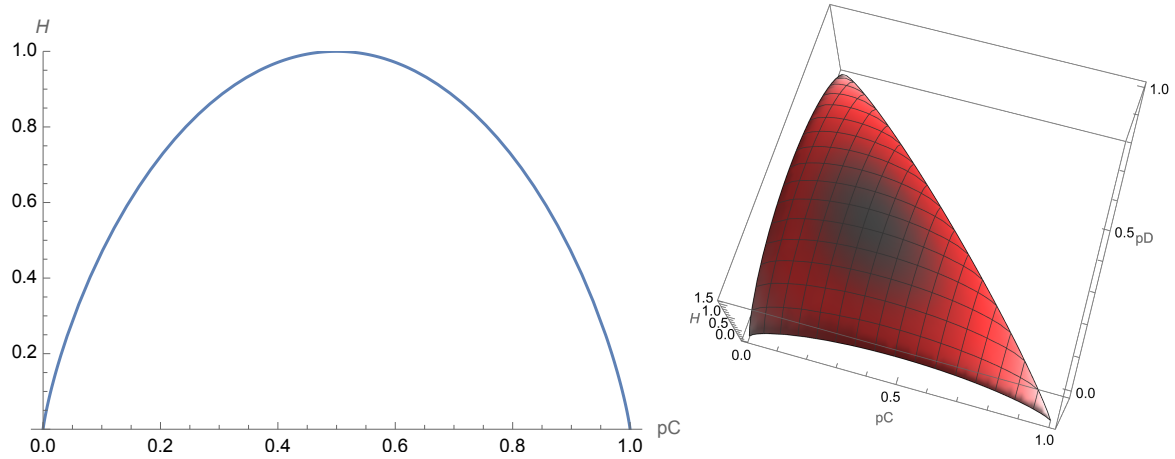
The entropy of an index set S is the average surprise [when you draw a point at random from S],

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

[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? $H(S) = -\log_2 \frac{1}{n} = \log_2 n$.

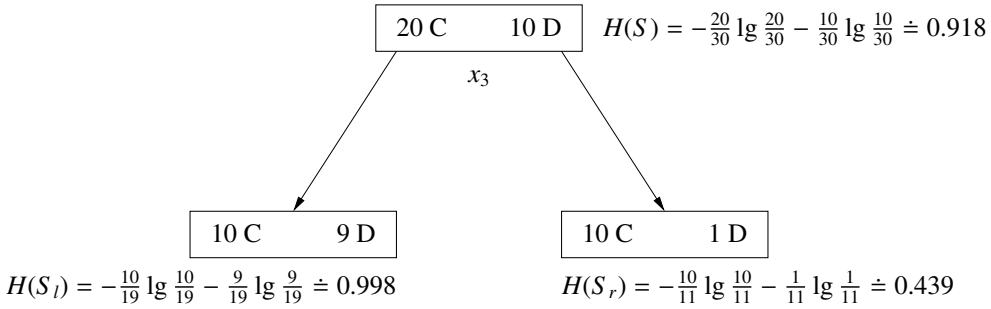
[The entropy is the expected number of bits of information we need to transmit to identify the class of a sample point in S chosen uniformly at random. It makes sense that it takes 1 bit to specify C or D when each class is equally likely. And it makes sense that it takes $\log_2 n$ bits to specify one of n classes when each class is equally likely.]



entropy.pdf [Left: plot of the entropy $H(p_\mathrm{C})$ when there are only two classes. The probability of the second class is $p_\mathrm{D} = 1 - p_\mathrm{C}$, so we can plot the entropy with just one dependent variable. Right: plot of the entropy $H(p_\mathrm{C}, p_\mathrm{D})$ when there are three classes. The probability of the third class is $p_\mathrm{E} = 1 - p_\mathrm{C} - p_\mathrm{D}$. Observe that the entropy is strictly concave.]

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

Choose split that maximizes information gain $H(S) - H_{\text{after}}$. [Which is just the same as minimizing H_{after}.]



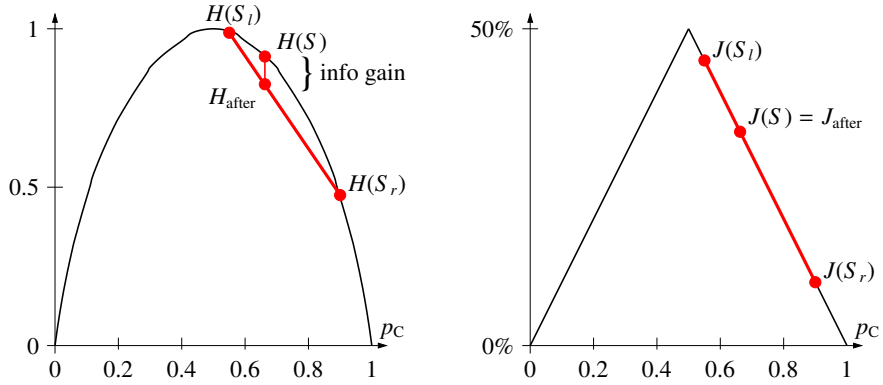
$H_{\text{after}} = 0.793$ info gain = 0.125

[Draw this by hand. infogain.pdf]

Info gain always positive *except* it is zero when one child is empty or
for all C, $P(y_i = \text{C} | i \in S_l) = P(y_i = \text{C} | i \in S_r)$. [Which is the case for the second split we considered.]

[Recall the graph of the entropy.]

$H(p_{\text{C}})$ entropy: strictly concave $J(p_{\text{C}})$ = % misclassified: concave, not strict



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

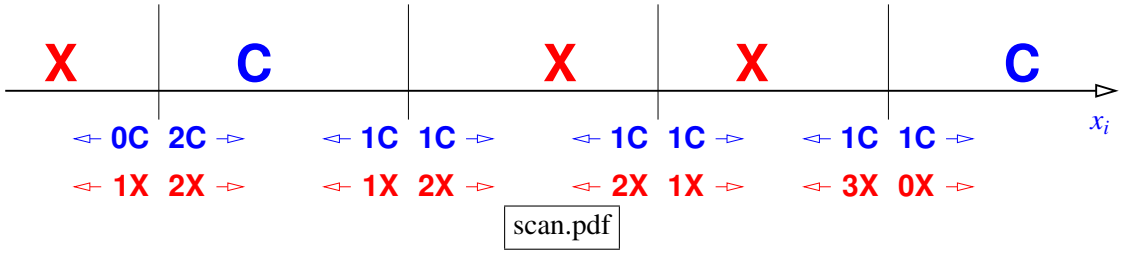
[Suppose we pick two points on the entropy curve, then draw a line segment connecting them. Because the entropy curve is strictly concave, the interior of the line segment is strictly below the curve. Any point on that segment represents a weighted average of the two entropies for suitable weights. If you unite the two sets into one parent set, the parent set's value p_{C} is the weighted average of the children's p_{C}'s. Therefore, the point directly above that point on the curve represents the parent's entropy. The information gain is the vertical distance between them. So the information gain is positive unless the two child sets both have exactly the same p_{C} and lie at the same point on the curve.]

[On the other hand, for the graph on the right, plotting 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 child sets into one, or splitting the parent set into two, changes neither the total misclassified sample points nor the weighted average of the % misclassified. The bigger problem, though, is that many different splits will get the same weighted average cost; this test doesn't distinguish the quality of different splits well.]

[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 x_i values in S; try splitting between each pair of *unequal* consecutive values.
  [We can radix sort the points 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![6]
  [This is important for obtaining a fast tree-building time.]
  [Draw a row of C's and X's; show how we update the # of C's and # of X's in each of S_l and S_r as we scan from left to right.]



scan.pdf

Algs & running times:

- Classify 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
  [Training on quantitative features is asymptotically just as fast as training on 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.
  [This is an amortized analysis: we are charging $O(d$ depth) time to each sample point.]
  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.]

[6] Let C be the number of class C sample points to the left of a potential split and c be the number to the right of the split. Let D be the number of class not-C points to the left of the split and d be the number to the right of the split. Update C, c, D, and d at each split (in $O(1)$ time per split) as you move from left to right. At each potential split, calculate the entropy of the left set as $-\frac{C}{C+D}\log_2\frac{C}{C+D} - \frac{D}{C+D}\log_2\frac{D}{C+D}$ and the entropy of the right set as $-\frac{c}{c+d}\log_2\frac{c}{c+d} - \frac{d}{c+d}\log_2\frac{d}{c+d}$. Note: $\log 0$ is undefined, but this formula works if we use the convention $0 \log 0 = 0$.

It follows that the weighted average of the two entropies is $-\frac{1}{n'}\left(C\log_2\frac{C}{C+D} + D\log_2\frac{D}{C+D} + c\log_2\frac{c}{c+d} + d\log_2\frac{d}{c+d}\right)$, where $n' = C + D + c + d$ is the total number of sample points stored in this treenode. Choose the split that minimizes this weighted average.

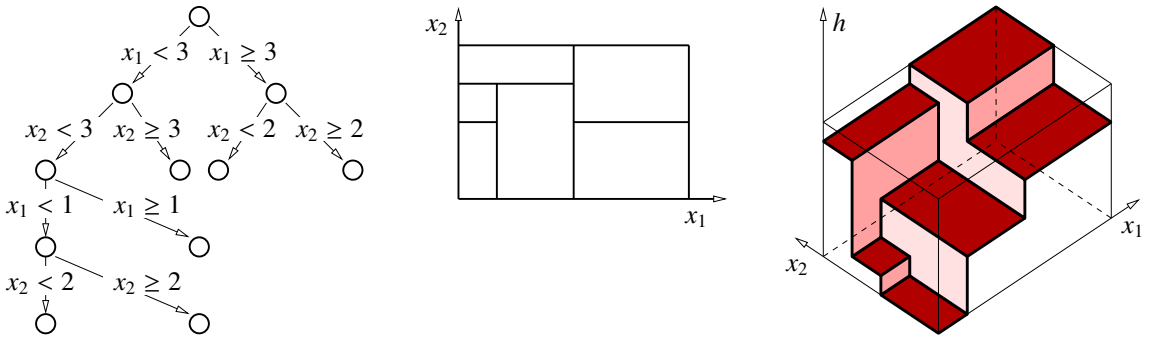
15 More Decision Trees, Ensemble Learning, and Random Forests

DECISION TREE VARIATIONS

[Last lecture, I taught you the vanilla algorithm for building decision trees and using them to classify test points. There are many variations on this basic algorithm; I'll discuss a few now.]

Decision Tree Regression

Creates a piecewise constant regression fn. [This seems too rudimentary to be true, but it's true.]



treeregresstree.pdf [Decision tree regression.]

Leaf stores label $\mu_S = \frac{1}{|S|}\sum_{i \in S} y_i$, the mean label for training pts $i \in S$.

Cost $J(S) = \text{Var}(\{y_i : i \in S\}) = \frac{1}{|S|}\sum_{i \in S}(y_i - \mu_S)^2$.

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

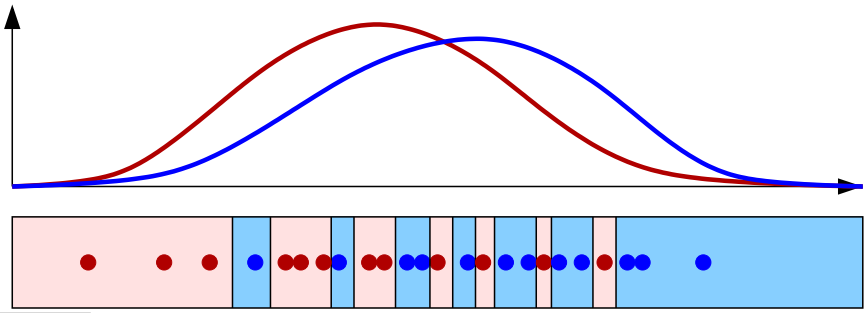
Stopping Early

[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?

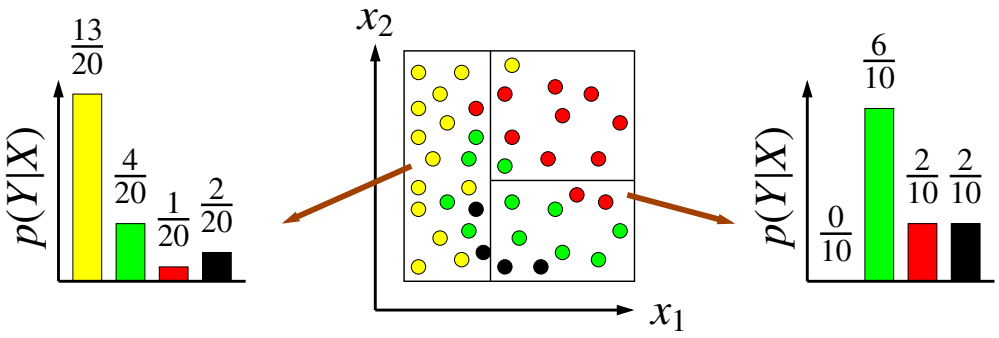
- Limit tree depth (for speed)
- Limit tree size (big data sets)
- Pure tree may overfit
- Given noise or overlapping distributions, pure leaves tend to overfit; better to stop early and estimate posterior probs

[When you have strongly overlapping class distributions, refining the tree down to one training point per leaf is absolutely guaranteed to overfit, giving you a classifier akin to the 1-nearest neighbor classifier. It's better to stop early, then classify each leaf node by taking a vote of its training points; this gives you a classifier akin to a k-nearest neighbor classifier.]



treeoverfit.pdf [Overlapping distributions cause pure decision trees to overfit. Compare this decision tree with the Bayes decision rule; the Bayes optimal decision boundary would be just one point.]

[Alternatively, you can use the points to estimate a posterior probability for each leaf, and return that. If there are many points in each leaf, the posterior probabilities might be reasonably accurate.]



leaf2.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.]

Leaves with multiple points return
- a majority vote or class posterior probs (classification) or
- an average (regression).

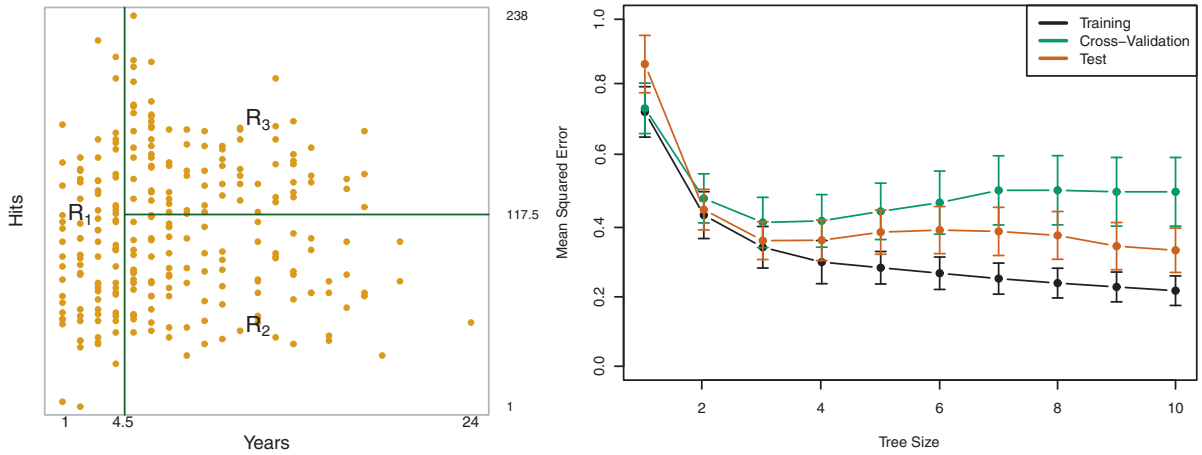
How to stop? Select stopping condition(s):
- Next split doesn't reduce entropy/error enough (dangerous; pruning is better)
- Most of node's points (e.g., > 90%) have same class [to deal with outliers & overlapping distribs]
- Node contains few training points (e.g., < 10) [especially for big data]
- Box's edges are all tiny [super-fine resolution may be pointless]
- Depth too great [risky if there are still many training points in the box]
- Use 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 lowers your validation error. 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 . . .]

Pruning

Grow tree too large; greedily remove each split whose removal improves validation performance.
[We have to do validation once for each split that we're considering reversing.]
More reliable than stopping early.

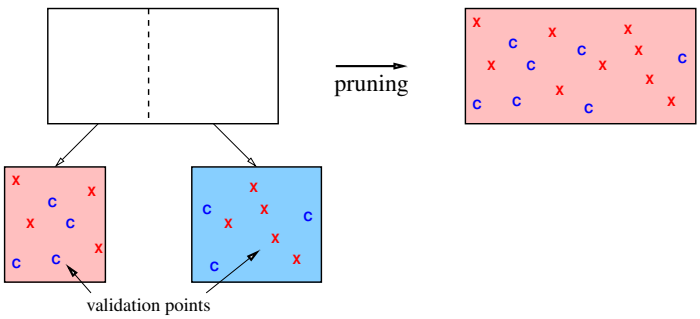
[The reason why pruning often works better than stopping early is because often a split that doesn't seem to make much progress is followed by a split that makes a lot of progress. If you stop early, you'll never find out. Pruning is simple and highly recommended when you have enough time to build and prune the tree.]



prunedhitters.pdf, prunehitters.pdf (ISL, Figures 8.5 & 8.2) [At left, a decision tree predicting the salaries of baseball players from years in Major League Baseball and hitting average: R1 = \$165,174, R2 = \$402,834, R3 = \$845,346. At right, a plot of decision tree leaf nodes vs. mean squared error. The graph shows that the decision tree with the best validation accuracy has three leaves, so that tree appears at left.]

[In this example, a 10-leaf decision tree was constructed to predict the salaries of baseball players, based on their years in the league and average hits per season. Then the tree was pruned by validation. The best decision tree on the validation data turned out to have just three leaves.]
[Observe that this tree is very *interpretable*. You could easily explain it to grandpa.]

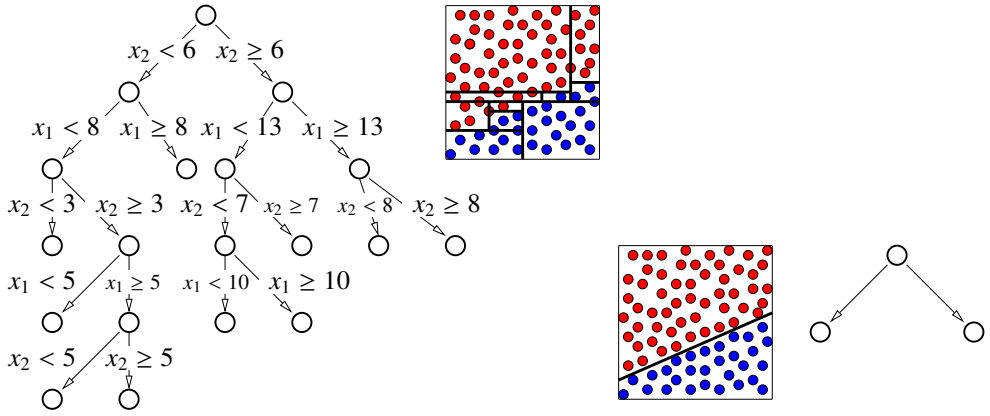
[It might seem expensive to do validation once for each split we consider reversing. But you can do it pretty cheaply. What you *don't* do is reclassify every validation point from scratch. Instead, you first compute which leaf each validation point winds up in, then for each leaf you make a list of its validation points. When you are deciding whether to remove a split, you just look at the validation 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 compute this very quickly.]



prunecheck.pdf [After we determine which leaf boxes each validation point ends up in, we find that pruning these two leaves improves the validation accuracy. Box colors indicate the majority classes of the training points (not shown). Local validation accuracy improves from 7/16 to 9/16.]

Multivariate Splits

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



multivar.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.]

[Here you can use other classification algorithms such as SVMs, logistic regression, and Gaussian discriminant analysis. Decision trees permit these algorithms to find nonlinear decision boundaries by making them hierarchical.]

May gain better classifier at cost of worse interpretability or speed.
[Standard decision trees are very fast because they check only one feature at each treenode. But if there are hundreds of features, and you have to check all of them at every level of the tree to classify a point, it slows down classification a lot.]
[A good compromise is to set a limit on the number of features you check at each treenode—say, three. You can use forward stepwise selection at each treenode to choose the three features.]
[On exams, assume we check only one feature per treenode unless we say otherwise!]

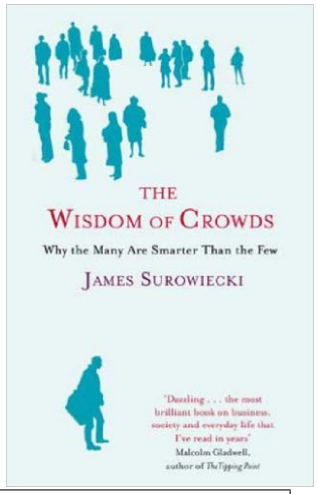
ENSEMBLE LEARNING

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. [Though we can achieve very low bias.]

[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 root 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 have just one random number, its variance might be high. But if you have n random 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.]

[National Public Radio 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.]

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.]
Metalearner takes test point, feeds it into all T learners, returns majority class or average output.

[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 [of all the weak learners]
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 with a $1,000,000 prize. The winning team, BellKor's Pragmatic Chaos, used an extreme ensemble method that took an average of many different learning algorithms. 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 are okay. Averaging reduces the variance!
[Each learner may overfit, but each overfits in its own unique way.]
Averaging sometimes reduces bias & increases flexibility a bit, but not reliably.
e.g., creating nonlinear decision boundary from linear classifiers.
[Averaging rarely reduces bias as much as it reduces variance, so get the bias small before averaging.]

Hyperparameter settings usually different than 1 learner. [Averaging reduces variance more than bias.]

[Sometimes the number of learners is said to be a hyperparameter, but extra learners improve the variance without worsening the bias. The main limit on the number of learners is computation time. So you trade off time for improved variance.]

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.]

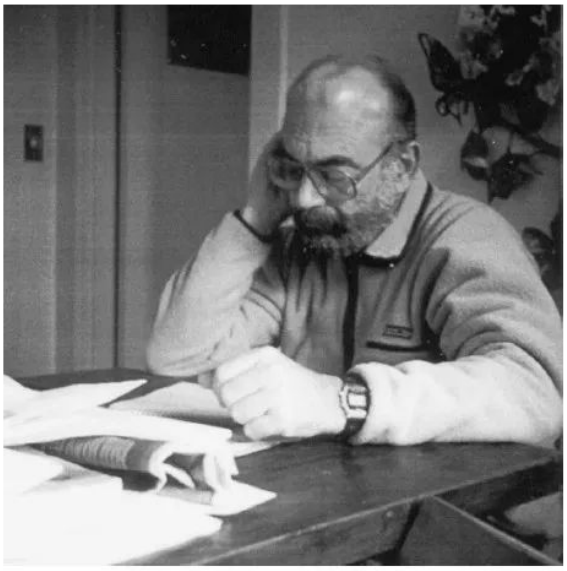
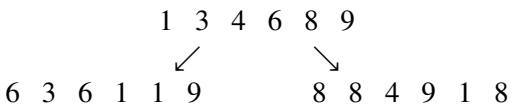


FIG. 6. *Leo working in a prior Berkeley residence*, 1985.

leobreiman3.png [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.

Build T learners from T subsamples.

Random Forests

Bagging + trees 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 might not change that. If the trees are too similar, then taking their average doesn't reduce the variance much.]
Let's reduce the correlation between different trees. [That makes averaging work better.]

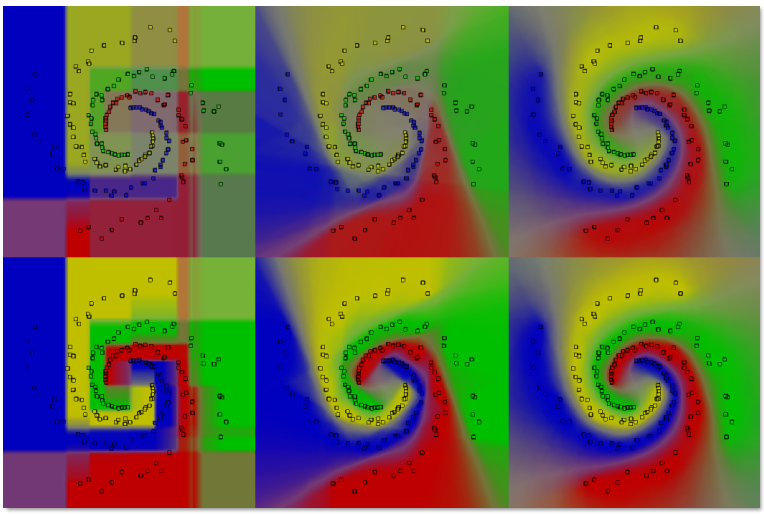
Idea: At each treenode, take random sample of m features (out of d).
Choose best split from m features.
[We're not allowed to split on the other $d - m$ features!]
Different random sample for each treenode.
$m \approx \sqrt{d}$ works well for classification; $m \approx d/3$ for regression.
[So if you have a 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. These values of m are good starting guesses.]
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, your average will have less variance than a single tree.]
[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.]

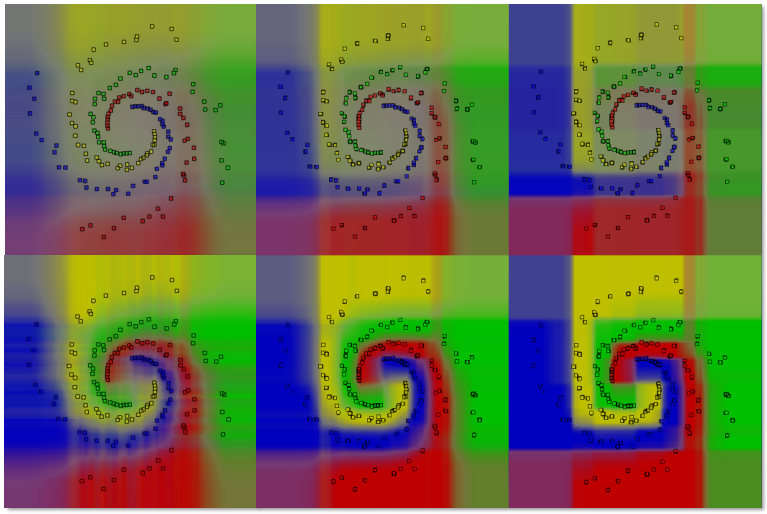
Sometimes test error drops even at 100s or 1,000s of decision trees!
Disadvantages: slow; loses interpretability/inference.
[But the compensation is it's a more accurate predictor than a single decision tree.]

[I will end by showing you examples of a very non-standard method for random forests that works magic in certain difficult circumstances.]
Idea: generate s random multivariate splits (oblique lines, quadrics); choose best split.
[You have to be clever about how you generate random decision boundaries; I'm not going to discuss that. 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 [Random forest classifiers for 4-class spiral data. Each forest takes the average of 400 trees. The top row uses trees of depth 4. The bottom row uses 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 [Random forest classifiers for the same data. Each forest takes the average of 400 trees. In these examples, *all* the splits are axis-aligned. The top row uses trees of depth 4. The bottom row uses trees of depth 12. From left to right, we choose each split from 1, 5, or 50 random choices. The more choices, the less bias and the better the classifier.]

16 Neural Networks

NEURAL NETWORKS

Can do both classification & regression.

[They tie together many ideas from the course: perceptrons, linear regression, 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 until something unfortunate happened in 1969. Marvin Minsky, one of the founding fathers of AI, and Seymour Papert published a book called "Perceptrons." Sounds promising? Well, part of the book was devoted to things perceptrons can't do. One of those things is XOR.]

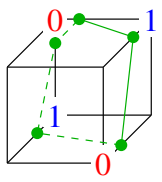
		x_1	
	XOR	0	1
x_2	0	0	1
	1	1	0

[Think of the four outputs here as training 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.]

[The XOR problem is also called parity, especially when you have more features: the input is a bunch of bits and you answer whether the number of 1's is even or odd. It was known even then that you could solve parity problems by adding extra layers of perceptrons, but Minsky and Papert gave technical proofs about some circumstances where this can't be done, and those limitations were misinterpreted. 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 first "AI Winter." Shortly after the book was published, Frank Rosenblatt died in a boating accident.]

[There are several almost obvious ways to get around the XOR problem. Here's the easiest.]

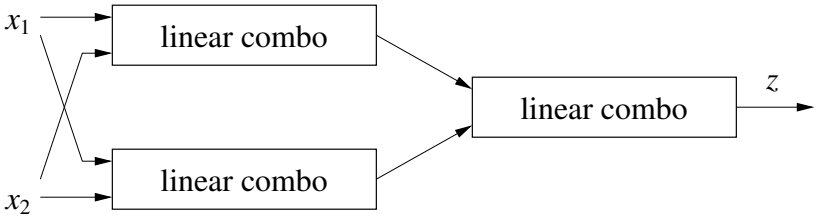
If you add one new quadratic feature, $x_1 x_2$, XOR is linearly separable in 3D.



[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 arbitrary logic circuits. Suppose I put together some linear decision functions like this.]



[Draw this by hand. lincombo.pdf]

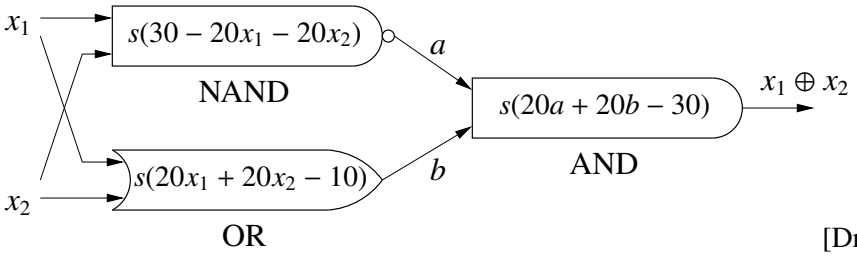
[Interpret the output as true if z is greater than one-half or false if z is less than one-half. Can I do XOR with this?]

A linear combo of linear combos 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 sends 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. That's what people usually use in practice these days.]

[However, the traditional choice was 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 for optimization. And we know that the logistic is often a good model for posterior probabilities.]

[With logistic functions between the linear combinations, here's a two-layer perceptron that computes the XOR function.]



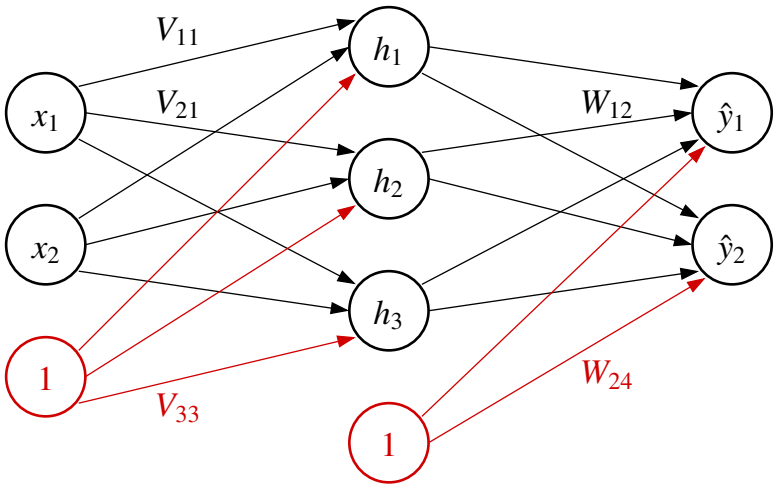
[Draw this by hand. xorgates.pdf]

[The big question is: can an algorithm *learn* a function like this?]

Network with 1 Hidden Layer

Input layer: $x_1, \ldots, x_d$; $x_{d+1} = 1$ [Index $d+1$ is the fictitious dimension.]
Hidden units: $h_1, \ldots, h_m$; $h_{m+1} = 1$
Output layer: $\hat{y}_1, \ldots, \hat{y}_k$

Layer 1 weights: $m \times (d+1)$ matrix V $V_i^\top$ is row i: weights into h_i
Layer 2 weights: $k \times (m+1)$ matrix W $W_i^\top$ is row i: weights into $\hat{y}_i$



[Draw this by hand. neuralnetwork.pdf]

Recall [logistic function] $s(\gamma) = \dfrac{1}{1+e^{-\gamma}}$. Other nonlinear fns can be used, called the activation fns.

For vector u, $s(u) = \begin{bmatrix} s(u_1) \\ s(u_2) \\ \vdots \end{bmatrix}$, $s_1(u) = \begin{bmatrix} s(u_1) \\ s(u_2) \\ \vdots \\ 1 \end{bmatrix}$ [We apply s to a vector component-wise.]

$$h = s_1(Vx) \quad \ldots \text{that is, } h_i = s(V_i \cdot x)$$
$$\hat{y} = s(Wh) = s(Ws_1(Vx))$$

[Neural networks often have more than one output. This allows us to 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.]

[We can add more hidden layers, and for image recognition tasks it's common to have 6 to 200 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 $L(\hat{y}, y)$ e.g., $L(\hat{y}, y) = \|\hat{y} - y\|^2$.

↑ ↑

predictions true labels (could be vectors)

Find V and W that minimize the cost fn $J(V, W) = \frac{1}{n} \sum_{i=1}^{n} L(\hat{y}(X_i), Y_i)$.

[I'm using a capital Y here because Y is a matrix with one row for each training point and one column for each output unit of the neural net. Each training point has a whole vector of labels Y_i, stored as a row of Y.]

Usually there are many local minima!
[The cost function for a neural net is, generally, not even close to convex. Sometimes, it's possible to wind up in a bad minimum. Usually, you can avoid bad minima by having lots of units in each layer.]

[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 out of an input unit have the same value, and all the weights into an output unit 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 local minimum, we start with random weights.]

Let w be a vector containing all the weights in V & W. Batch gradient descent:

$w \leftarrow$ vector of random weights
repeat
    $w \leftarrow w - \epsilon \nabla J(w)$

[We've just rewritten all the weights as a vector for notational convenience. When you actually write the code, for the sake of speed, you should probably operate directly on the weight matrices V and 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 a modest change in the input values causes barely any change in the output value. Stuck units tend to stay stuck. I'll say more about that next lecture.]

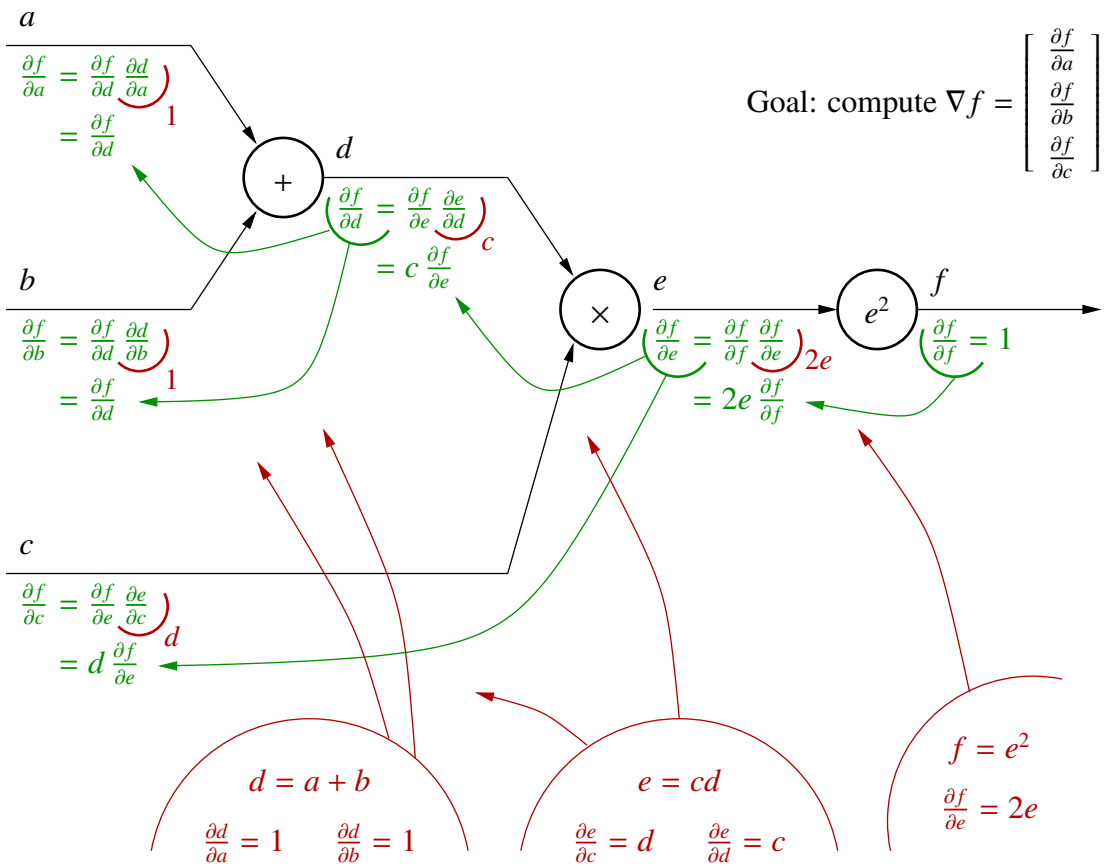
[Instead of batch gradient descent, we can use stochastic gradient descent, which means we use the gradient of one training point's loss function at each step. Typically, we shuffle the points in a random order, or just pick one randomly at each step. I'll say more about that next week.]

[The hard part of this algorithm is computing the gradient. If you simply derive one derivative for each weight, you'll find that for a network with many layers of hidden units, it takes time linear in the number of edges in the neural network to compute a derivative for one weight. Multiply that by the number of weights. We'll spend the rest of this lecture learning to improve the running time to linear in the number of edges.]

Naive gradient computation: $O(\text{edges}^2)$ time
Backpropagation: $O(\text{edges})$ 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

where z is an input to n.

$$\frac{\partial f}{\partial z} = \left(\frac{\partial f}{\partial n}\right)\left(\frac{\partial n}{\partial z}\right)$$

computed during forward pass

computed during backward pass after forward pass

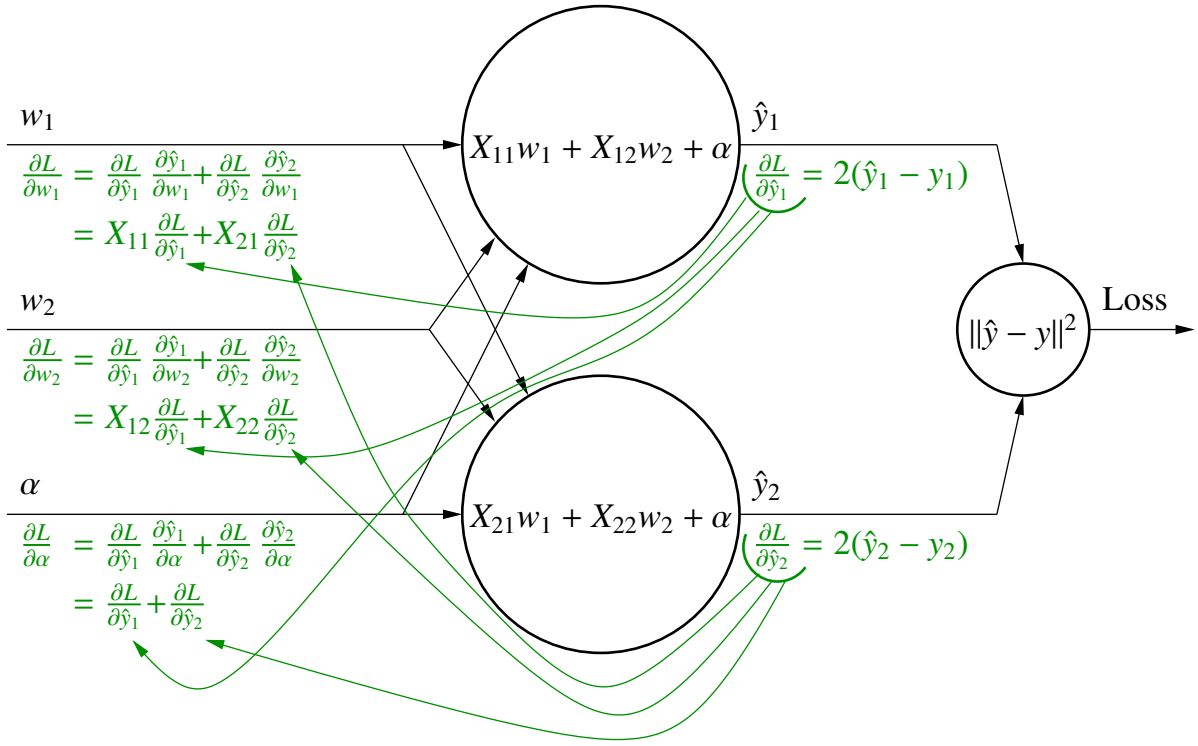
"backpropagation"

[Draw this by hand. gradientsarith.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. This is a standard rule of multivariate calculus:]

$$\frac{\partial}{\partial \alpha} L(y_1(\alpha), y_2(\alpha)) = \frac{\partial L}{\partial y_1}\frac{\partial y_1}{\partial \alpha} + \frac{\partial L}{\partial y_2}\frac{\partial y_2}{\partial \alpha} = \nabla_y L \cdot \frac{\partial}{\partial \alpha} y$$

[With this rule, let's compute gradients for an expression from least-squares linear regression.]



[Draw this by hand. gradientsmulti.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.]

[In one sense, all we've done here is to rederive the fact that the gradient of the least-squares regression cost function is $\nabla_w L = 2X^\top(\hat{y} - y)$, where $\hat{y} = Xw$. But the way we've divided it into a forward pass and a backward pass gives us a way to generalize it by adding more layers of computations.]

The Backpropagation Alg.

[Backpropagation is a dynamic programming algorithm for computing the gradients we need to do neural net stochastic gradient descent in time linear in the number of weights.]

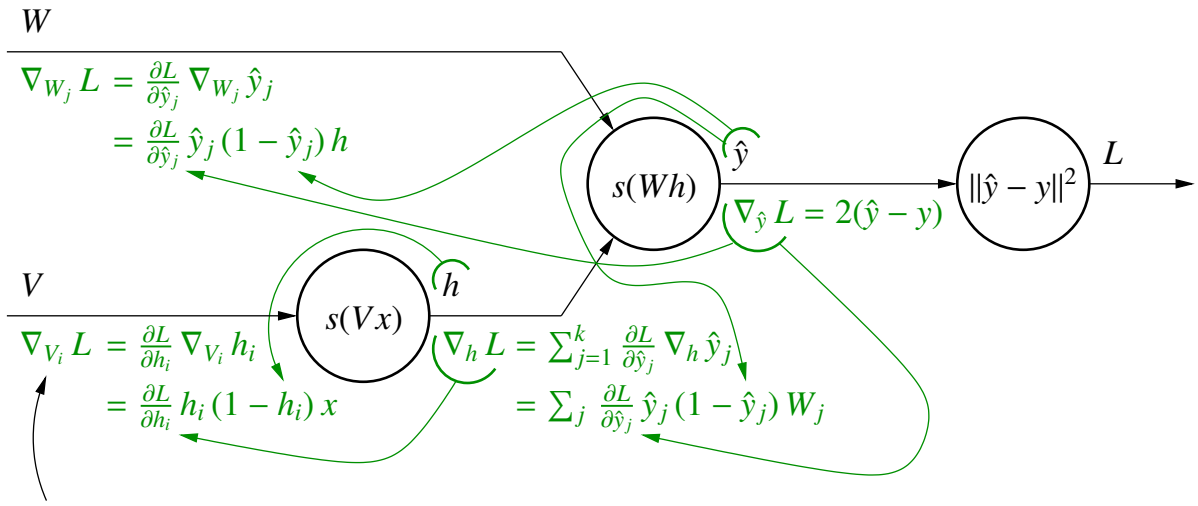
Recall $s'(\gamma) = s(\gamma)\,(1 - s(\gamma))$; $\quad V_i^\top$ is row i of weight matrix V [and likewise for rows of W]

$h_i = s(V_i \cdot x)$, so $\quad \nabla_{V_i} h_i = s'(V_i \cdot x)\, x = h_i\,(1 - h_i)\, x$

$\hat{y}_j = s(W_j \cdot h)$, so $\quad \nabla_{W_j} \hat{y}_j = s'(W_j \cdot h)\, h = \hat{y}_j\,(1 - \hat{y}_j)\, h$

$\nabla_h \hat{y}_j = \hat{y}_j\,(1 - \hat{y}_j)\, W_j$

[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 I started with. But now we treat the weights V and W as the inputs, rather than the point x.]



Compute $\nabla_V L$, $\nabla_W L$ one row at a time.

[Draw this by hand. backalg.pdf]

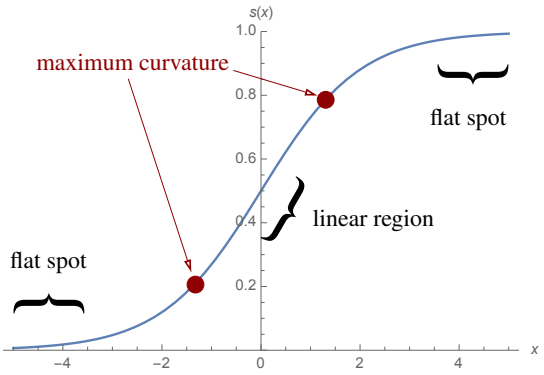
[Note that h and $\hat{y}$ are computed during the forward pass, and $\nabla_{\hat{y}} L$, $\nabla_h L$, $\nabla_W L$, and $\nabla_V L$ are computed during the backward pass. In particular, we can't compute $\nabla_V L$ until after we compute $\nabla_h L$, and we can't compute that until after we compute $\nabla_{\hat{y}} L$. The loss L doesn't need to be explicitly computed at all! We can compute all the gradients without it.]

17 Vanishing Gradients; ReLUs; Output Units and Losses; Neurobiology

THE VANISHING GRADIENT PROBLEM; ReLUs

[Last lecture, we put a logistic function at the output of every unit except the input units. These units are called sigmoid units. But in practice, sigmoid units are usually a poor choice for hidden layers.]

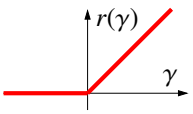
Problem: When unit output s is close to 0 or 1 for most training points, $s' = s(1-s) \approx 0$, so gradient descent changes s very slowly. Unit is "stuck." Slow training.



logistic.pdf [Draw flat spots, "linear" region, & maximum curvature points (at $s(\lambda) \doteq 0.21$ and $s(\lambda) \doteq 0.79$) of the sigmoid function. Ideally, we would stay away from the flat spots.]

[This is called 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.]

Solution: Replace sigmoids with ReLUs: rectified linear units.
ramp fn: $r(\gamma) = \max\{0, \gamma\}$.



$$r'(\gamma) = \begin{cases} 1, & \gamma \geq 0, \\ 0, & \gamma < 0. \end{cases}$$

[The derivative is not defined at zero, but we just pretend it is for the sake of gradient descent.]

Most neural networks today use ReLUs for the hidden units.
[However, it is still common to use sigmoids for the output units in classification problems.]

[ReLUs are preferred over sigmoids as hidden units because in practice, they're much less likely to get stuck. But the derivative r' 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 training points. But yes, ReLUs sometimes get stuck too; just not as often as sigmoids.]

[The output of a ReLU can be arbitrarily large; the fact that ReLUs don't saturate like sigmoids do leaves them vulnerable to a related problem called the "exploding gradient problem." It is not a big problem in shallow networks, but it becomes a big problem in deep or recurrent networks.]

[Even though ReLUs are linear in each half of their range, they're still nonlinear enough to easily compute functions like XOR. Of course, if you replace sigmoids with ReLUs, you have to change the derivation of backprop to reflect the changes in the gradients. We'll do that later in this lecture.]

OUTPUT UNITS

[Many neural networks use ReLUs for all or most of the hidden units, but ReLUs are rarely used as output units. The output units are chosen to fit the application, and there are three common choices.]

Most output units are linear units (regression) or sigmoid/logistic or softmax units (classification).

[When you train a neural network with these output units by gradient descent, the last layer of edges of the network is solving a problem in linear regression, logistic regression, or softmax regression by gradient descent. Or maybe all three!]

(1) Linear units for regression.
Given vector h of unit values in last hidden layer, output layer computes $\hat{y} = Wh$.
Activation fn is the identity fn. [You could say there is no activation function.]
Then the final layer of edges is doing linear regression (on values of h & y)!
Usually trained with squared-error loss. If so, it's least-squares linear regression.

[When we train a neural network by gradient descent, each linear output unit finds the solution of a linear regression problem by gradient descent. In principle we could find the weights entering that unit by solving the normal equations. But we don't, because the hidden unit values keep changing during training.]

(2) Sigmoid units [aka logistic units] for [two-class] classification.
Let $y \in \mathbb{R}^k$ be vector of labels; $y_i \in [0, 1]$.
Given hidden layer h, output layer computes pre-activation $a = Wh \in \mathbb{R}^k$ and applies sigmoid activations to obtain prediction $\hat{y} = s(a)$.

[Here, s is the logistic function applied component-wise to the vector a. The labels y_i are usually 0's and 1's, but $\hat{y}_i$ can never be exactly 0 or 1. So it might be better to choose target labels like $y_i = 0.05$ or $y_i = 0.95$, because then a neural network with enough weights and sufficiently wide layers can achieve $\hat{y} = y$ exactly for every training point! Unless there are co-located training points with different labels.]

Loss fn: Use logistic loss instead of squared error. Fixes vanishing gradients at output!
[The logistic loss function prevents output units from suffering the vanishing gradient problem, but it can't solve the vanishing gradient problem for hidden units. So we don't use sigmoid hidden units.]

[When we train a neural network by gradient descent, each sigmoid output unit finds the solution of a logistic regression problem by gradient descent.]

(3) Softmax units for k-class classification.
[E.g., in the MNIST digit recognition problem, we would have $k = 10$ softmax output units, one for each digit class.]

Let $y \in \mathbb{R}^k$ be vector of labels for training pt x [indicating x's membership in the k classes].
[It is easy to design a neural network to solve more than one multi-class classification problem simultaneously, but for ease of notation let's suppose we're solving just one, so there are only k output units.]

Strongly recommended: choose training labels so $\sum_{i=1}^{k} y_i = 1$.

We commonly use a one-hot encoding: one label is 1, the others are 0.
[But one-hot encoding has a disadvantage we've already discussed for sigmoids: each softmax prediction $\hat{y}_i$ can never be exactly 1 or 0. It might be better to choose target labels such as 0.9, 0.05, and 0.05. Think of the labels as posterior probabilities, so they should sum to 1.]

Given hidden layer h, output layer computes pre-activation $a = Wh \in \mathbb{R}^k$ and applies softmax activation to obtain prediction $\hat{y} \in \mathbb{R}^k$.

Softmax output is $\hat{y}_i(a) = \dfrac{e^{a_i}}{\sum_{j=1}^{k} e^{a_j}}$. Each $\hat{y}_i \in (0, 1)$; $\sum_{i=1}^{k} \hat{y}_i = 1$.

[Interpret $\hat{y}_i$ as an estimate of the posterior probability that the input belongs to class i.]

Loss fn: Use cross-entropy. Fixes vanishing gradients at output.

For k-class softmax output, cross-entropy is $L(\hat{y}, y) = -\sum_{i=1}^{k} y_i \ln \hat{y}_i$.

$\uparrow$ true labels
$\uparrow$ prediction
$\Big\}$ k-vectors

[When we train a neural network by gradient descent, if there are softmax output units, those units find the solution of a softmax regression problem by gradient descent.]

[Cross-entropy losses are only for softmax and sigmoid outputs. For linear or ReLU outputs, cross-entropy makes no sense, but squared error loss makes sense.]

Backpropagation for Outputs

For backprop, we need $\nabla_W L$ and $\nabla_h L$, where h is last hidden unit layer, W is weights of last edge layer.

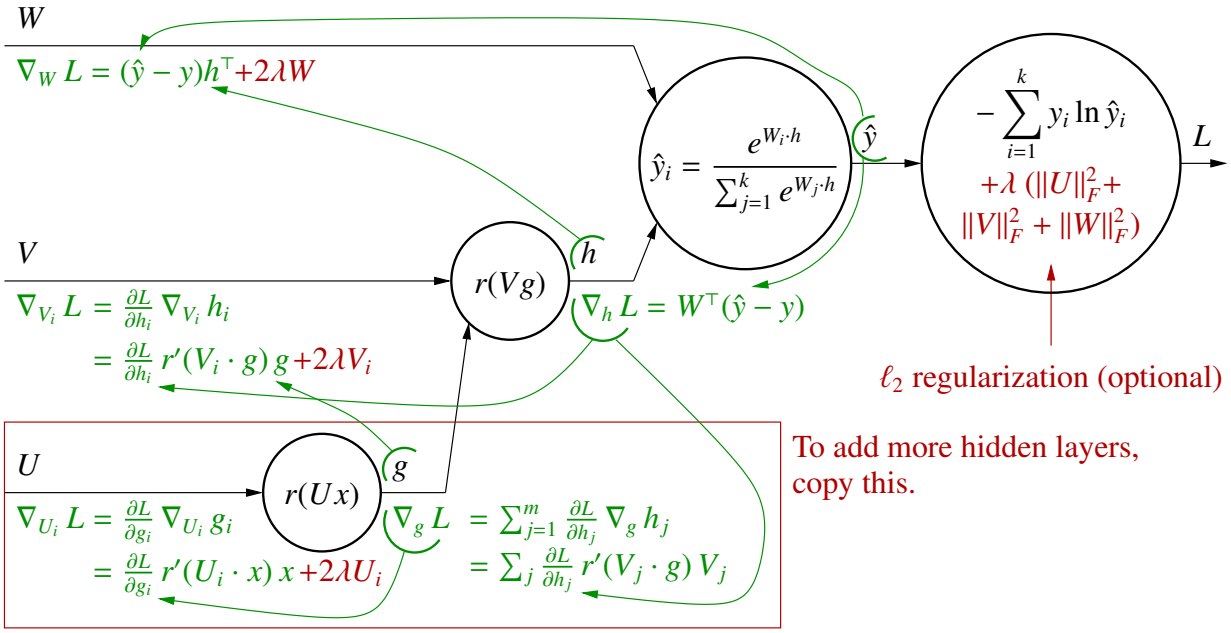
output + loss	linear + squared error	sigmoid + logistic loss	softmax + cross-entropy
$\hat{y} =$	Wh	$s(Wh)$	$\text{softmax}(Wh)$
$L(\hat{y}, y) =$	$\|\hat{y} - y\|^2$	$-\sum_i (y_i \ln \hat{y}_i + (1 - y_i) \ln(1 - \hat{y}_i))$	$-\sum_i y_i \ln \hat{y}_i$
$\nabla_W L =$	$2(\hat{y} - y)\, h^\top$	$(\hat{y} - y)\, h^\top$	$(\hat{y} - y)\, h^\top$
$\nabla_h L =$	$2W^\top(\hat{y} - y)$	$W^\top(\hat{y} - y)$	$W^\top(\hat{y} - y)$
			assuming $\sum_{i=1}^k y_i = 1$

[It's interesting that all three types of outputs produce essentially the same form of gradients for the final layer of the network, except that the predictions $\hat{y}$ are different in each case. This is true even though each linear or sigmoid output unit is independent, but the softmax outputs units are coupled with each other.]

[Observe that even for sigmoid and softmax units, both $\nabla_W L$ and $\nabla_h L$ are linear in the error $\hat{y} - y$. This is a nice outcome, when you consider the exponentials and logarithms we started with. It implies that sigmoid units with logistic loss do not get stuck when the sigmoid derivatives are small. This is related to the fact that the logistic loss goes to infinity as the predicted value $\hat{y}_i$ approaches zero or one. The vanishing gradient of the sigmoid unit is compensated for by the exploding gradient of the logistic loss.]

Note: we don't need to compute $\nabla_{\hat{y}} L$.
Instead, we eliminate $\hat{y}$ by substituting $\hat{y}(W, h)$ into $L(\hat{y}, y)$.
[. . . before taking derivatives. This makes it easier both to derive the math and to write the code.]

[Now I will show you how to perform backpropagation for two hidden layers of ReLU units, a k-class softmax output, the cross-entropy loss function, and ℓ_2 regularization—which may improve test accuracy.]



[Draw this by hand. backpropsoft2.pdf]

[Note that $r(Ux)$ is the ramp function applied component-wise to the vector Ux. The derivative $r'(U_i \cdot x)$ is always zero or one. Observe that we don't need to compute the loss L at all. We also don't compute $\nabla_{\hat{y}} L$, as we said above, but we do need to compute the value of $\hat{y}$ to compute gradients.]

Derivations

[I won't go over this page of derivations in lecture, but I include them here for completeness. Students who want to understand neural networks deeply should spend some time going through these.]

Linear output units ($\hat{y} = Wh$) with squared error loss:

$$\begin{aligned}
L(\hat{y}, y) &= \|\hat{y} - y\|^2 = \|Wh - y\|^2 = \sum_{i=1}^{k} (W_i \cdot h - y_i)^2, \\
\nabla_{W_i} L &= 2(W_i \cdot h - y_i)\, h = 2(\hat{y}_i - y_i)\, h, \\
\nabla_W L &= 2(\hat{y} - y)\, h^\top, \\
\nabla_h L &= 2W^\top (Wh - y) = 2W^\top (\hat{y} - y).
\end{aligned}$$

Sigmoid [logistic] output units ($\hat{y} = s(a) = s(Wh)$) with logistic loss:

$$\begin{aligned}
L(\hat{y}, y) &= -\sum_{i=1}^{k} (y_i \ln \hat{y}_i + (1 - y_i) \ln(1 - \hat{y}_i)) = -\sum_{i=1}^{k} \left(y_i \ln \frac{1}{1 + e^{-a_i}} + (1 - y_i) \ln \left(1 - \frac{1}{1 + e^{-a_i}} \right) \right) \\
&= \sum_{i=1}^{k} \left(y_i \ln(1 + e^{-a_i}) - (1 - y_i) \ln \frac{e^{-a_i}}{1 + e^{-a_i}} \right) \\
&= \sum_{i=1}^{k} \left(y_i \ln(1 + e^{-a_i}) - (1 - y_i)(-a_i - \ln(1 + e^{-a_i})) \right) = \sum_{i=1}^{k} \left((1 - y_i) a_i + \ln(1 + e^{-a_i}) \right), \\
\frac{\partial L}{\partial a_i} &= 1 - y_i - \frac{e^{-a_i}}{1 + e^{-a_i}} = \frac{1}{1 + e^{-a_i}} - y_i = \hat{y}_i - y_i, \\
a_i &= W_i \cdot h, \qquad \nabla_{W_i} a_i = h, \qquad \nabla_h a_i = W_i, \\
\nabla_{W_i} L &= \frac{\partial L}{\partial a_i} \nabla_{W_i} a_i = (\hat{y}_i - y_i)\, h, \\
\nabla_W L &= (\hat{y} - y)\, h^\top, \\
\nabla_h L &= \sum_{i=1}^{k} \frac{\partial L}{\partial a_i} \nabla_h a_i = \sum_{i=1}^{k} (\hat{y}_i - y_i) W_i = W^\top (\hat{y} - y).
\end{aligned}$$

Softmax output units ($\hat{y} = \text{softmax}(a) = \text{softmax}(Wh)$) with cross-entropy loss:

[This derivation uses the assumption that $\sum_{i=1}^{k} y_i = 1$ for each training point's labels.]

$$\begin{aligned}
L(\hat{y}, y) &= -\sum_{i=1}^{k} y_i \ln \hat{y}_i = -\sum_{i=1}^{k} y_i \left(a_i - \ln \sum_{j=1}^{k} e^{a_j} \right) = -\sum_{i=1}^{k} y_i a_i + \ln \sum_{j=1}^{k} e^{a_j}, \\
\frac{\partial L}{\partial a_i} &= -y_i + \left(e^{a_i} \Big/ \sum_{j=1}^{k} e^{a_j} \right) = \hat{y}_i - y_i.
\end{aligned}$$

From here, we repeat the last four lines of the sigmoid derivation.

NEUROBIOLOGY

[The field of artificial intelligence started with some wrong premises. The early AI researchers attacked problems like chess and theorem 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 beat world chess champions, but they still can't play with toys 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 their 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.]

  [In the 1920's, the psychologist Karl Lashley conducted experiments to identify where in the brain memories are stored. He trained rats to run a maze, and then made lesions in different parts of the cerebral cortex, trying to erase the memory trace. Lashley failed; his rats could still find their way through the maze, no matter where he put lesions. He concluded that memories are not stored in any one area of the brain, but are distributed throughout it. Neural networks, properly trained, can duplicate this property.]

  superior for vision, speech, associative memory

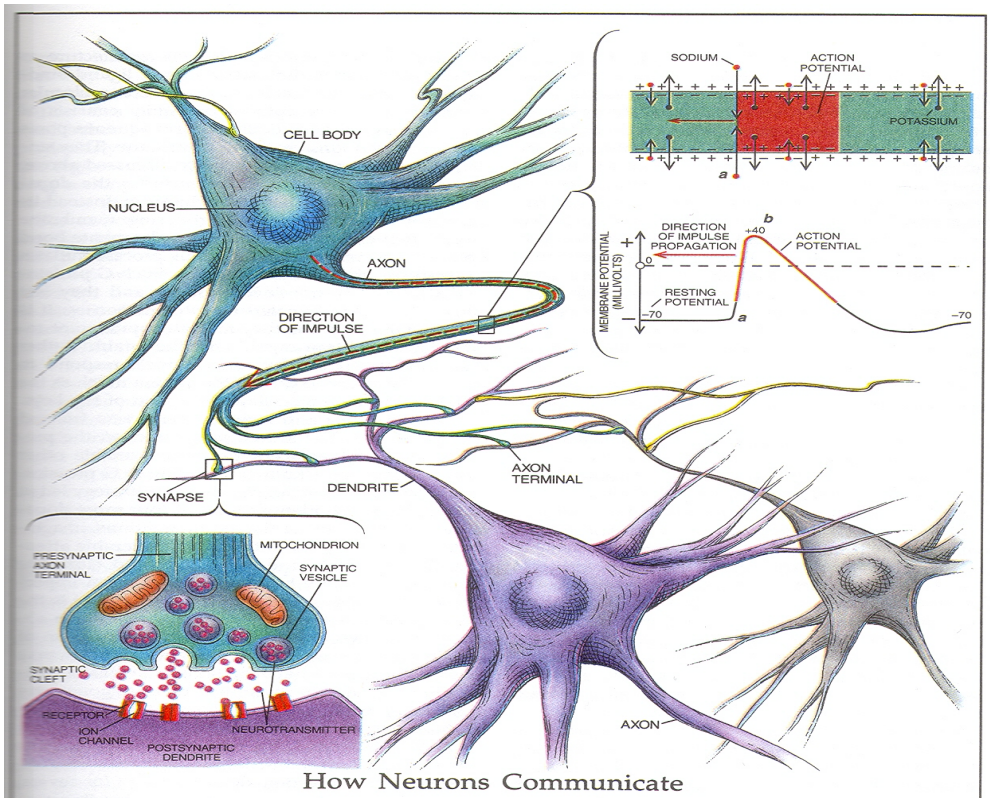
  [By "associative memory," I mean noticing connections between things. One thing our brains are very good at is retrieving a pattern if we specify only a portion of the pattern.]

[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 run in parallel, but also because of their computation style.]

[Neural nets try to emulate the parallel, associative thinking style of brains, and they are the best techniques we have for many fuzzy problems, including most problems in vision and speech. Not coincidentally, neural nets are also inferior at many traditional computer tasks such as multiplying 10-digit numbers or compiling source code.]

18 Neurobiology; Faster Neural Network Training

NEUROBIOLOGY (cont'd)



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 squids have a very large axon they use for fast propulsion by expelling jets of water. The mathematics of action potentials was first characterized in these squid axons, and that work won a Nobel Prize in Physiology in 1963.]
- Dendrite: Smaller limb 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.

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 a "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 overpower downstream neurons. We accomplish the same thing with the sigmoid function.]
- Weight change/learning ↔ synaptic plasticity
  [Donald] 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. They can work, but they're generally not nearly as fast or effective as backpropagation.]

[Backpropagation is one part of artificial neural networks for which any analogy is doubtful. There have been some proposals that the brain might do something vaguely like backpropagation,[7] but it seems tenuous. Learning in brains is still not well understood.]

[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 the things we do.]

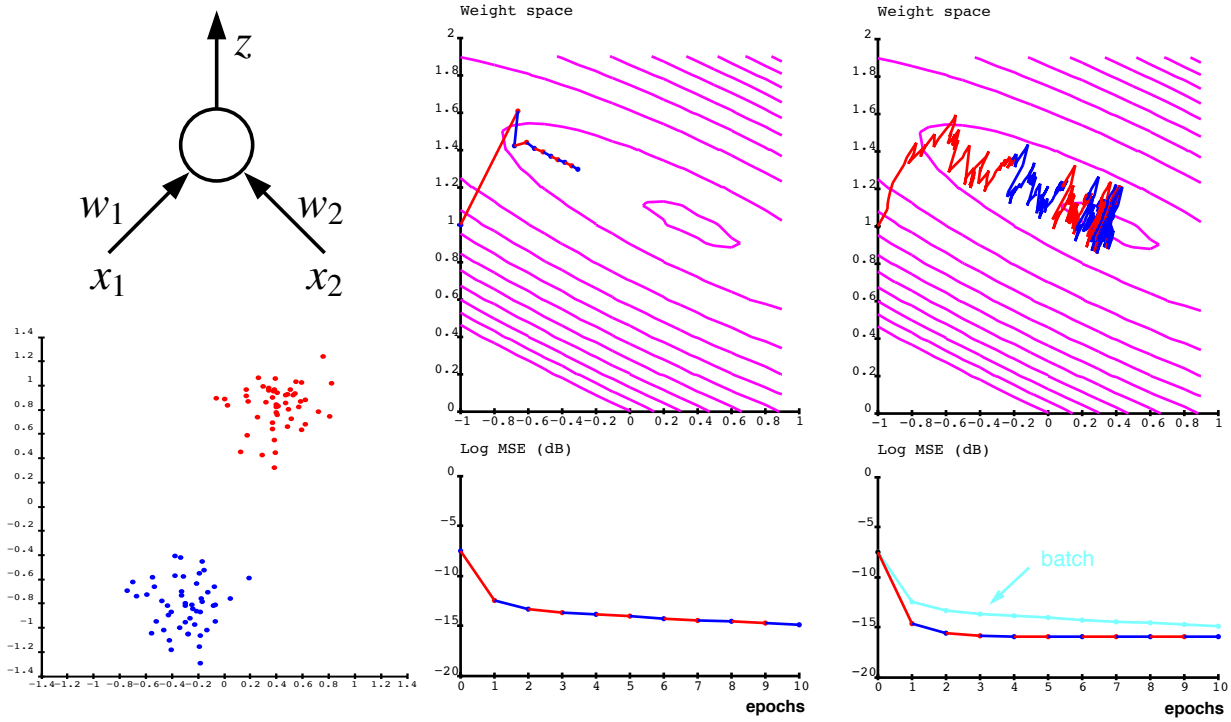
[7] See Lillicrap et al., "Backpropagation and the Brain," *Nature Reviews Neuroscience* **21**, pages 335-–346, April 2020.

HEURISTICS FOR FASTER TRAINING

[A big disadvantage of neural nets is that they take a long, long time to train compared to other classification methods we've studied. Here are some ways to speed them up. Unfortunately, you usually have to experiment with techniques and hyperparameters to find which ones will help with your particular application. I suggest you implement vanilla backpropagation first, usually in combination with stochastic gradient descent and intelligent weight initialization, and experiment with fancy heuristics only after you get that working.]

(1) ReLUs. [To fix the vanishing gradient problem, as described in the previous lecture.]

(2) Stochastic gradient descent (SGD): faster than batch on large, redundant data sets.
[Whereas batch gradient descent walks downhill on one cost function, stochastic descent takes a very short step downhill on one point's loss function and then another short step on another point's loss function. The cost function is the sum of the loss functions over all the sample points, so one batch step is akin to n stochastic steps and does roughly the same amount of computation. But if you have many different examples of the digit "9", they contain much redundant information, and stochastic gradient descent learns the redundant information more quickly—often *much* more quickly. Conversely, if the data set is so small that it encodes little redundant information, batch gradient descent is typically faster.]



batchvsstochmod.pdf (LeCun et al., "Efficient BackProp") [Left: a perceptron with only two weights trained to minimize the mean squared error cost function, and its 2D training data. Center: batch gradient descent makes only a little progress each epoch. Epochs alternate between red and blue. Right: stochastic descent decreases the error much faster than batch descent. Again, epochs alternate between red and blue.]

One epoch presents every training point once. Training usually takes many epochs, but if sample is huge [and carries lots of redundant information], SGD can take less than one epoch.

(3) SGD with minibatches.
Choose a minibatch size b; e.g. 256.
Repeatedly perform gradient descent on the sum of the loss functions of b randomly chosen points.

[Although we perform gradient descent on a minibatch of training points all at once, we don't call it *batch* gradient descent. We still call it *stochastic* gradient descent.]

Advantages [compared to SGD done just one point at a time]:
- Less "bouncy"; usually converges more quickly.
  [SGD bounces around wildly. Minibatches reduce the variance of the steps by a factor of $\sqrt{b}$ while maintaining the advantages of SGD.]
- Can use parallelism, vectorization, GPUs efficiently.
  [The backpropagation computations are fully independent from one training point to another, so it's very easy to compute gradients for multiple points in parallel or through vectorization.]
- Better speed because of memory hierarchy.
  [You should lay out the activations for the training points in the minibatch next to each other in memory. With the right memory layout and minibatch size, your use of the caches and memory hierarchy can be very efficient. Performing SGD on 64 training points might be almost as fast as performing SGD on one. The bottleneck in neural network training is memory latency, not arithmetic.]

[Minibatches nearly always work faster than processing just one training point at a time. They are standard in implementations of neural network training.]

Typically, we shuffle training pts, partition into $\lceil n/b \rceil$ minibatches.
An epoch presents each minibatch once. [Reshuffling for each epoch is optional.]
[It is important to randomize well so your minibatch is a representative subsample of the training points. Sometimes practitioners store each class in a separate list, shuffle each class separately, and build minibatches with a proportional number of training points from each class.]

[Be forewarned that the best learning rate ϵ will be different for different values of the minibatch size b, and there isn't always a predictable relationship between b and the best ϵ.]

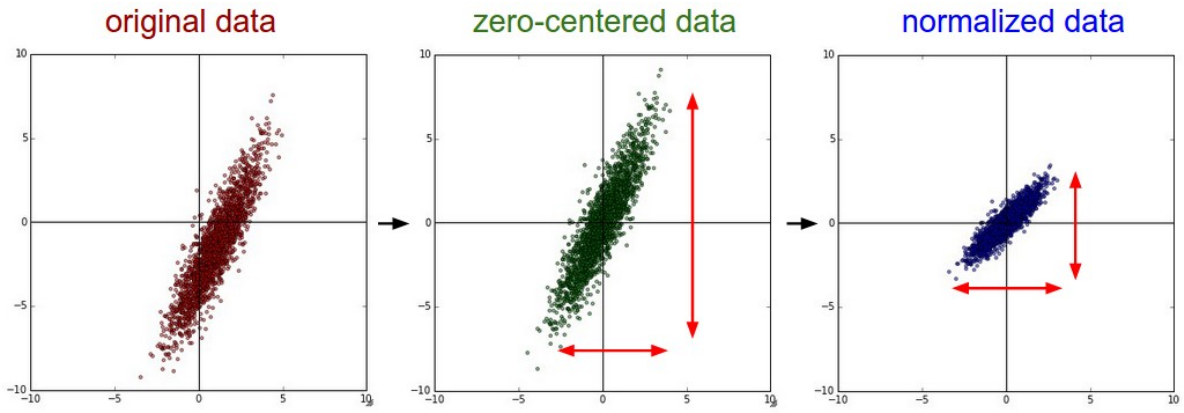
(4) To choose learning rate ϵ, use a small random subsample of training data.
[Practitioners have found that the size of the training set has only a weak effect on the best choice of ϵ. So use a subsample to quickly estimate a good learning rate, then apply it to your whole training set. This is very easy to do, and it can save you a lot of time!]

(5) Emphasizing schemes. [Neural networks learn the most redundant examples quickly, and the most rare examples slowly. This motivates emphasizing schemes, which repeat the rare examples more often.]
- Stochastic: present examples from rare classes more often, or w/bigger ϵ.
- Batch: examples from rare classes have bigger losses.

[Emphasizing schemes are a natural way to incorporate an asymmetric loss function. Suppose you're developing a test for a rare disease, and you have a small number of examples of people with the disease, but many examples of people without it. The classifier might decide to always return negative, "no disease," because it can achieve 99% accuracy that way. So we'd like to impose larger losses on examples with positive labels. For batch gradient descent, we make their losses bigger in the cost function. For SGD there are two options: use a bigger loss for rare examples, equivalent to presenting them with a larger learning rate ϵ; or present them more often. It's not clear which of these two options will train faster, as those extra presentations take more computation but shorter steps have better convergence properties.]

[Emphasizing schemes can also be used to emphasize misclassified training points, like the Perceptron Algorithm does, but that can backfire if those points are bad outliers.]

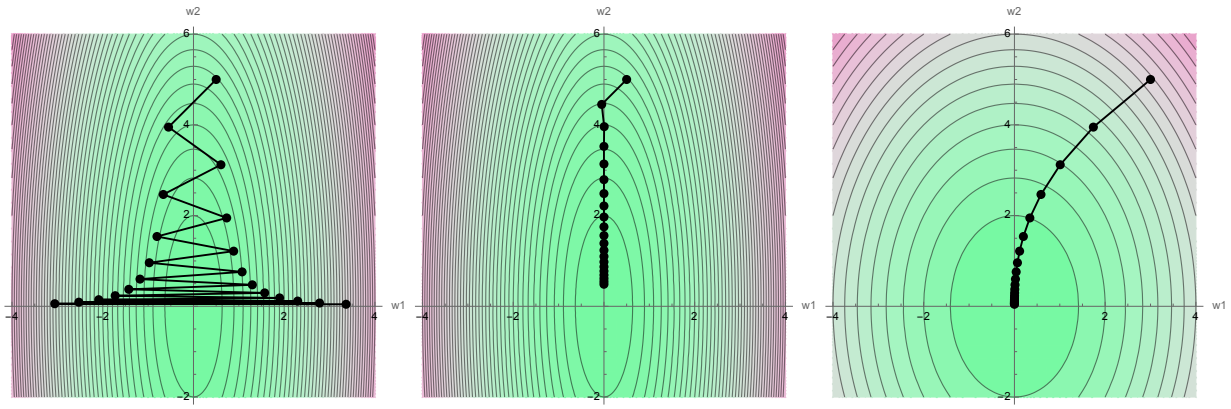
(6) Normalizing the training pts.
- Center each feature so mean is zero.
- Then scale each feature so variance ≈ 1.



normalize.jpg [A 2D example of normalizing points.]

[Remember that the power of neural networks comes from the nonlinearity of the activation function, and the nonlinearity of a sigmoid or ReLU unit falls where the linear combination of values coming in is close to zero. Centering makes it easier for the first layer of hidden units to be in the nonlinear operating region.]

[Neural networks are an example of an optimization algorithm whose cost function tends to have better-conditioned Hessians if the input features are normalized, so it may converge to a local minimum faster.]



illcondition105.pdf, illcondition055.pdf, goodcondition.pdf

[Recall these illustrations from Lecture 5. Gradient descent on a function with an ill-conditioned Hessian matrix can be slow because a large step size diverges in one direction (left) while a small step size converges slowly in another direction (center). Normalizing the data might improve the conditioning of the Hessian (right), thus speeding up gradient descent. Moreover, if you use ℓ_2-regularization, normalization makes it penalize the features more equally.]

[You could go even further and whiten the data, as we discussed in Lecture 9, but whitening takes $\Theta(nd^2+d^3)$ time for n training points with d features, so it takes too much time if d is very large; whereas normalizing takes $\Theta(nd)$ time.]

[Remember that whatever linear transformation you apply to the training points, you *must* later apply the same linear transformation to the test points you want to classify!]

(7) Initializing weights. [Proper initialization of weights is very important, especially for deep networks. Consider this carefully for Homework 6!]

[Recall that we initialize a neural network with random weight values to break the symmetry between hidden units. If we make those random values too small, they might never grow enough, especially if the network is deep. If we make them too large, we may cause the exploding gradient problem in ReLUs, or the vanishing gradient problem in sigmoid units. Here are some rules of thumb for initializing random weights.]

Consider the variance of each unit's output, given random weights.
Principle: output of unit should have same variance as each of its inputs.[8]

For a unit with fan-in η (not counting bias term), initialize each incoming edge to ...
[The fan-in of a unit is the number of connections entering the unit.]
For a ReLU unit, a weight in $\mathcal{N}(0, 2/\eta)$ or $\mathcal{U}\left(-\sqrt{6/\eta},\ \sqrt{6/\eta}\right)$.
[This is called He initialization, after Kaiming He.]
For a sigmoid unit, make it $\mathcal{N}(0, 12.8/\eta)$ or $\mathcal{U}\left(-\sqrt{38.4/\eta},\ \sqrt{38.4/\eta}\right)$.
For a linear or tanh unit, make it $\mathcal{N}(0, 1/\eta)$ or $\mathcal{U}\left(-\sqrt{3/\eta},\ \sqrt{3/\eta}\right)$.
[This initialization is sometimes called Xavier initialization, but it isn't quite what Xavier Glorot originally proposed. A tanh unit is very similar to a sigmoid unit, but its output is centered at zero, whereas sigmoid outputs are centered at 0.5. I don't recommend you use either sigmoid or tanh units as hidden units, but if you do, the tanh is preferable for that reason.]

[The reason we divide by the fan-in is because the more inputs a unit has, the greater its incoming signal is. So we must make the weights smaller to match the unit's output variance to the variance of each input.]

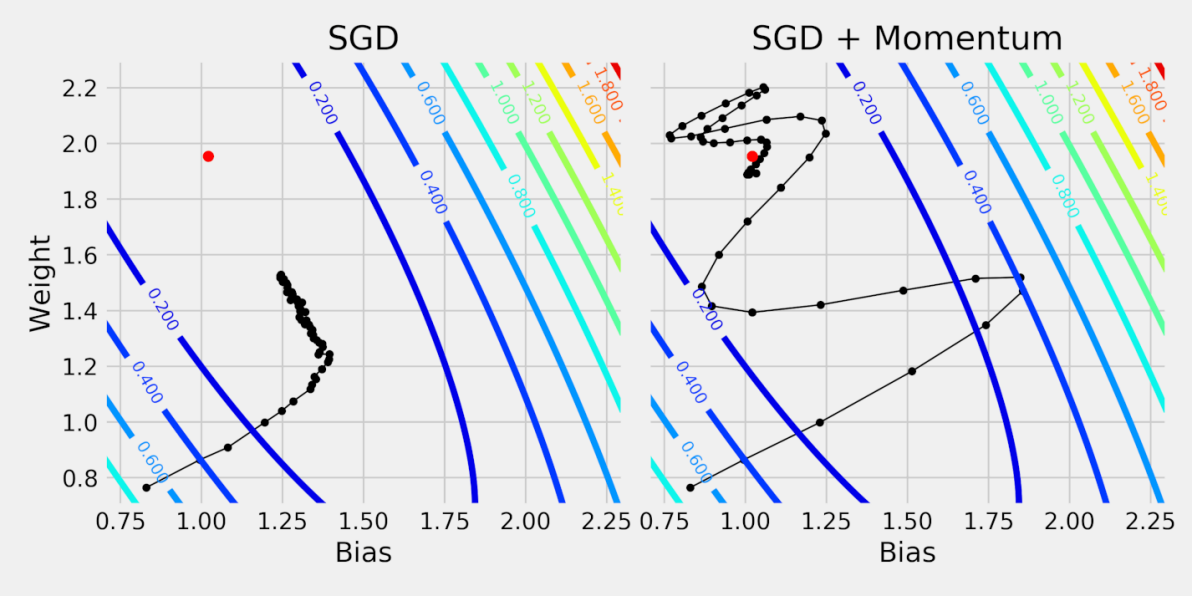
Set bias terms to zero. [Bias terms can too easily overpower signals coming from earlier layers. For ReLU units, some people suggest setting the bias terms to a small positive constant so they're more likely to be turned on at first, but other people say it gives worse performance in practice.]
Linear output unit: set bias term to the mean label.
Sigmoid output unit: set bias term so default output is the mean label.
[E.g., if 90% of your training points are in class C, set the bias so the sigmoid output defaults to 0.9. Andrej Karpathy says that if you don't initialize the output unit biases, the first few minibatches are largely wasted learning the mean labels.]

(8) Momentum. Gradient descent changes "momentum" m slowly. [The intuition is that if you've taken many steps in roughly the same direction, you should go faster in that direction.]

$m \leftarrow -\epsilon' \nabla J(w)$
repeat
    $w \leftarrow w + m$
    $m \leftarrow \beta m - \epsilon \nabla J(w)$

Good for both batch & stochastic. Choose hyperparameter $\beta < 1$.
[Here, J is the cost for the minibatch, which could be anything from a single training point to the whole training set. The hyperparameter β specifies how much 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. Geoff Hinton suggests starting at 0.5 and slowly increasing it to 0.9 or higher as the gradients get small.]
[If β is large, you should usually choose ϵ small to compensate, but you might still use a large ϵ' in the first line so the initial velocity is reasonable.]

[8] For an explanation of these suggestions, see Siddharth Krishna Kumar, "On Weight Initialization in Deep Neural Networks."



sgdmomentumgodoy.png (Daniel Godoy) [Left: 50 steps of SGD (with 16-point mini-batches) don't get very close to the minimum (red). Right: 50 steps with momentum do get close to the minimum, but overshoot it several times.]

[A problem with momentum is that once it gets close to a good minimum, it overshoots the minimum, then oscillates around it. But it often gets us close to a good minimum sooner. We see both phenomena above.]

pretzelwaterpark.jpg [How I imagine a neural network's cost function. It does not resemble a parabolic bowl. The downhill paths from the start to the local minima are sinuous. The swimmers here employ gradient descent with momentum with great success.]

19 Convolutional Neural Networks

CONVOLUTIONAL NEURAL NETWORKS (ConvNets; CNNs)

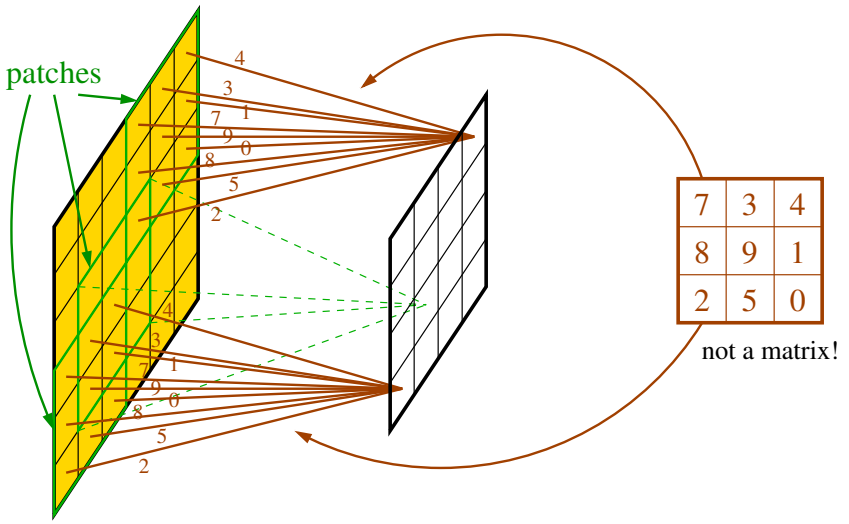
[Convolutional neural nets drove a big resurgence of interest in neural nets starting in 2012. Often you'll hear the buzzword deep learning, which refers to neural nets with many layers. Most image recognition networks are deep and convolutional. In 2018, the Association for Computing Machinery gave the Alan M. Turing Award to Geoff Hinton, Yann LeCun, and Yoshua Bengio for their work on deep neural networks.]

Vision: inputs are images. 400×400 image = 160,000 pixels.
If we connect them all to 160,000 hidden units $\rightarrow$ 25.6 billion connections.
[With so many weights, the network is very slow to train or even to use once trained.]

[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. Let's apply these two properties to neural net design, plus one new idea: we'll learn the edge detectors instead of hard-coding them.]

ConvNet ideas:

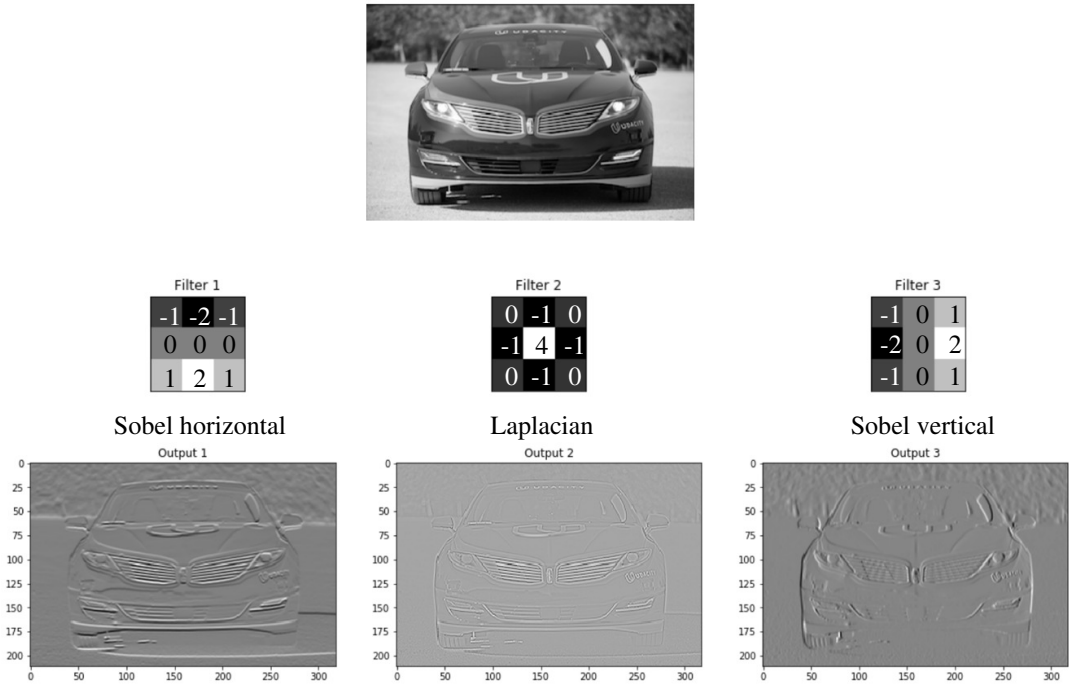
- Local connectivity: A hidden unit connects only to a small patch of units in previous layer.
  [A unit in the first hidden layer doesn't look at the whole image. It looks only at a small number of pixels—typically 9, 25, or 49 pixels. This speeds up both training and classification considerably.]
- Shared weights: Groups of hidden units share same set of weights, called a filter aka mask aka kernel.
  Each filter operates on every patch of image.



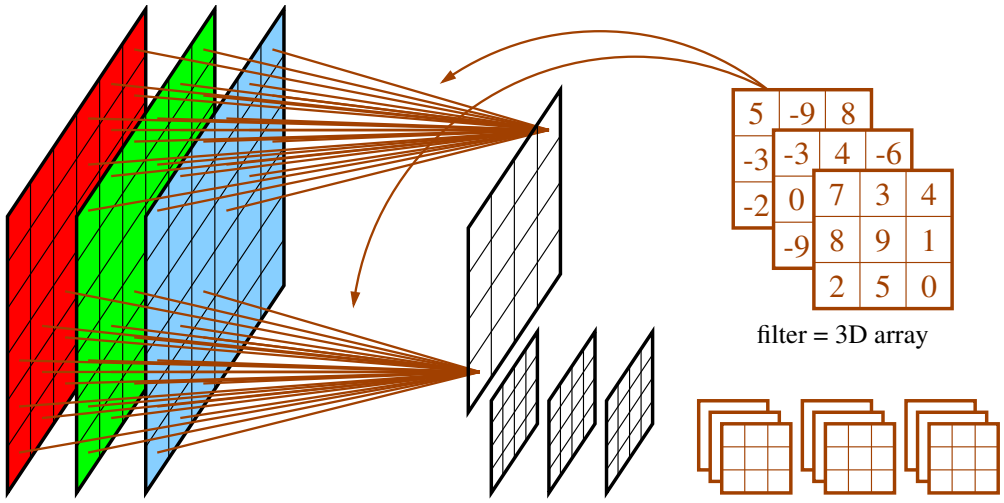
convlayer.pdf [Applying a filter to an image. Every hidden unit uses the same nine shared weights. In this example, a 6×6 image is covered by 16 overlapping 3×3 patches, yielding a 4×4 activation map of hidden units. We learn the filter by backpropagation.]

If image size is $J \times K$ and filter size is $M \times M$, the activation map is $(J - M + 1) \times (K - M + 1)$ hidden units—one for each patch.

A convolutional layer learns multiple filters. There is one activation map per filter. A channel is an activation map, OR another dimension such as the red/green/blue channels of an input image.



caredges.pdf (Cezanne Camacho) [Two Sobel filters (one horizontal, one vertical) and a Laplacian filter applied to an image, yielding three activation maps (three channels). Note that these filters were not learned by a CNN (but they could be).]



convchannels.pdf [A convolutional layer (of edges). If a layer's input has more than one channel, then each filter is represented by a three-dimensional matrix. The set of all filters is represented by a four-dimensional matrix.]

[The output of a convolutional layer has multiple channels, and usually so does the input. The layer's output has one channel per filter, and these channels becomes inputs to downstream convolutional layers. The input to the neural network often has multiple channels too; most commonly, the color channels of a color image.]

If edge layer l has $C^{(l-1)}$ channels in and $C^{(l)}$ channels out ($C^{(l)}$ filters),
- # of weights/filter = $C^{(l-1)} \times M \times M$;
- # of weights in layer = $C^{(l-1)} \times C^{(l)} \times M \times M$;
- # of units out = $C^{(l)}\times$ # of patches = $C^{(l)} \times (J - M + 1) \times (K - M + 1)$.

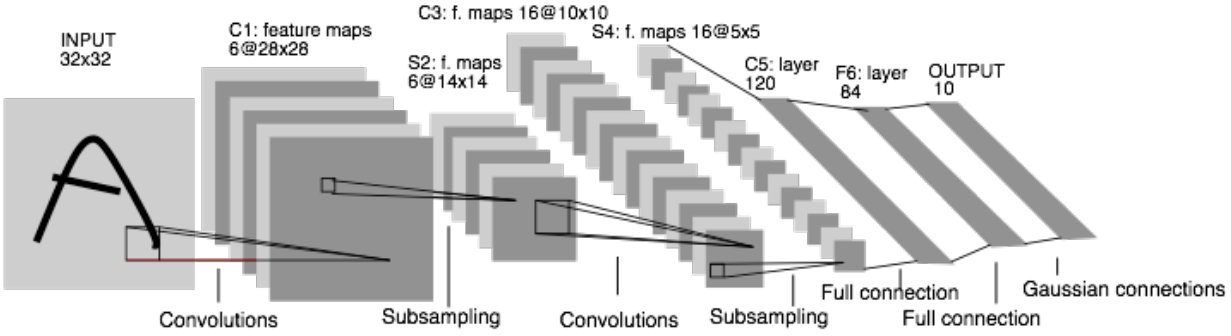
Typically, each convolutional hidden unit ends with a ReLU activation.
[But there are exceptions in many modern CNNs.]

Options for bias terms:
- Untied bias: $C^{(l)} \times (J - M + 1) \times (K - M + 1)$ bias terms—one per unit out.
- Tied bias: $C^{(l)}$ bias terms—one per filter/channel out.
- No bias terms. [This option is usually *not* immediately followed by a ReLU.]

[Untied bias terms are a lot of extra weights, sometimes more weights than the filters! Sometimes they give better test accuracy, but not always, so try validating both ways.]

Benefits of shared weights:
- Much less memory needed. [Better cache behavior too.]
- Regularization. [It's unlikely that a weight will become spuriously large if it's used in many places.]
- If one filter learns to detect edges, *every* patch has an edge detector.
  [Because the filter that detects edges is applied to every patch.]
  ConvNets exploit repeated structure in images, audio.
- A filter destined to become an edge detector learns on edges in *every* part of every image.
  [So it can learn the idea faster.]

[In a neural net, you can think of hidden units as added features that we learn, as opposed to added 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.]



LeNet5.png Architecture of LeNet5.

[ConvNets were first popularized by the success of Yann LeCun's "LeNet 5" handwritten digit recognition software. LeNet 5 has six hidden layers! Hidden layers 1 and 3 are *convolutional layers* with shared weights. Layers 2 and 4 are *pooling layers* that make the image smaller, with no weights at all. Layers 5 and 6 are fully-connected layers of hidden units with no shared weights. A great deal of experimentation went into

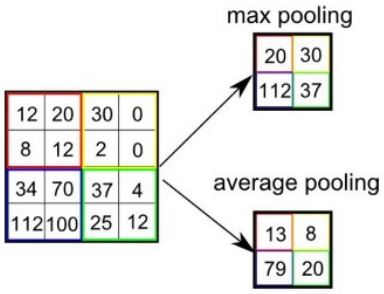
figuring out the number of layers and their sizes. At its peak, LeNet 5 was responsible for reading the zip codes on 10% of US Mail. Another Yann LeCun system was deployed in ATMs and check reading machines and was reading 10 to 20% of all the checks in the US by the late 90's. LeCun is one of the Turing Award winners I told you about earlier.]

[Show Yann LeCun's video LeNet5.mov, illustrating LeNet 5.]

Downsampling

[At the output of LeNet 5, we have to compress the information down to a single 10-unit output. Experience shows that this is best done by slowly compressing the information in the image through a sequence of layers, rather than connecting a very large layer of hidden units directly to the output. This observation echoes classic image processing techniques that were developed before neural networks. The two popular methods of downsampling are called *pooling* and *strided convolution*.]

Max pooling: Reduce a $J \times K$ image to $\lceil J/2 \rceil \times \lceil K/2 \rceil$ or $\lceil J/3 \rceil \times \lceil K/3 \rceil$ [as illustrated].

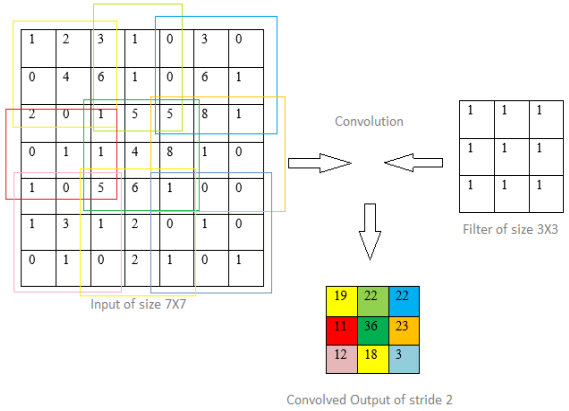


maxavgpool.pdf (Ekpenyong et al.) [Max pooling and average pooling.]

Average pooling: likewise, but each unit out is the average of four units in.
[Average pooling was used in LeNet 5, but max pooling seems more popular now.]

Pooling layers have *no weights*. Nothing to train!
[But you still have to think carefully about how to do backpropagation through them.]

Strided convolution: Patches overlap less (or not at all).



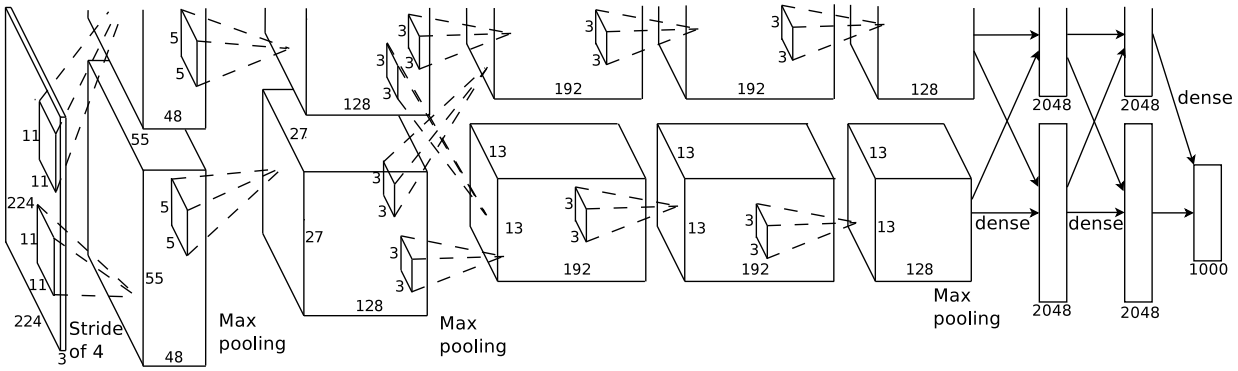
stride.pdf (Vadlamani & Patel) [Strided convolution.]

AlexNet

[I told you three lectures ago that neural net 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 much faster to train, and they more or less have only one hyperparameter, whereas neural nets take a lot of work to tune.]

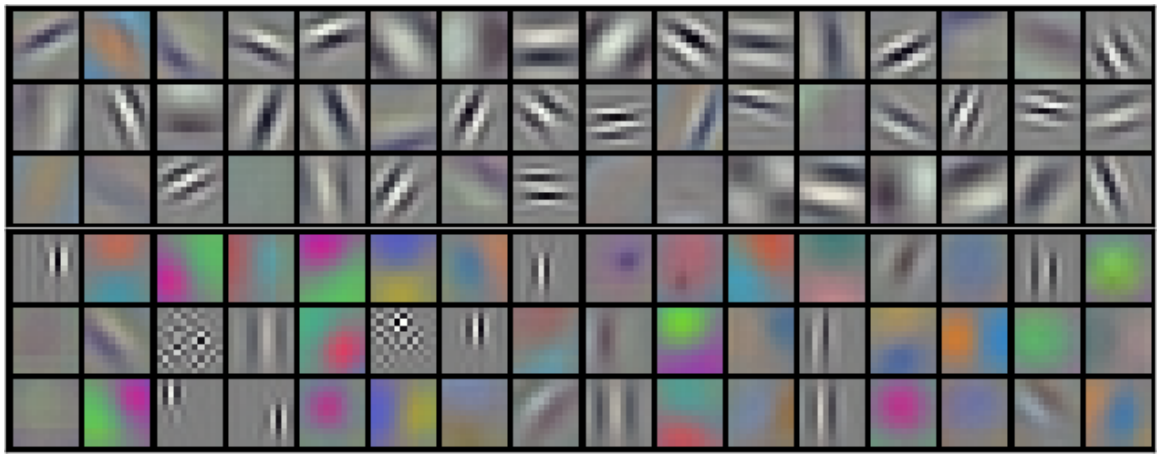
[Neural nets are now in their third wave of popularity. The single biggest factor in bringing them back is probably big data. Thanks to the internet, we now have absolutely huge collections of images to train neural nets with, and researchers have discovered that neural nets often give better performance 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.]

[The event that brought attention back to neural nets was the ImageNet Image Classification Challenge in 2012. 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 recent 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, dropout, data augmentation, and GPUs for training.]



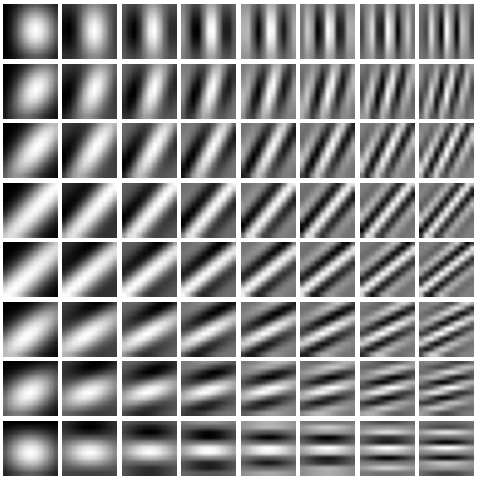
alexnet.pdf (Krizhevsky et al., 2012, 2017) [Architecture of AlexNet.]

[When ConvNets were first applied to image analysis, researchers found that some of the learned filters are edge detectors! Here are the first layers of filters learned by AlexNet.]



filtersalex.png (Krizhevsky et al., 2012, 2017) [Filters learned by the first layer of AlexNet.]

[Not all of the features are edge detectors; many of them are more concerned with color. But more than half of them resemble mathematical functions called Gabor filters, which detect edges and also textures.]

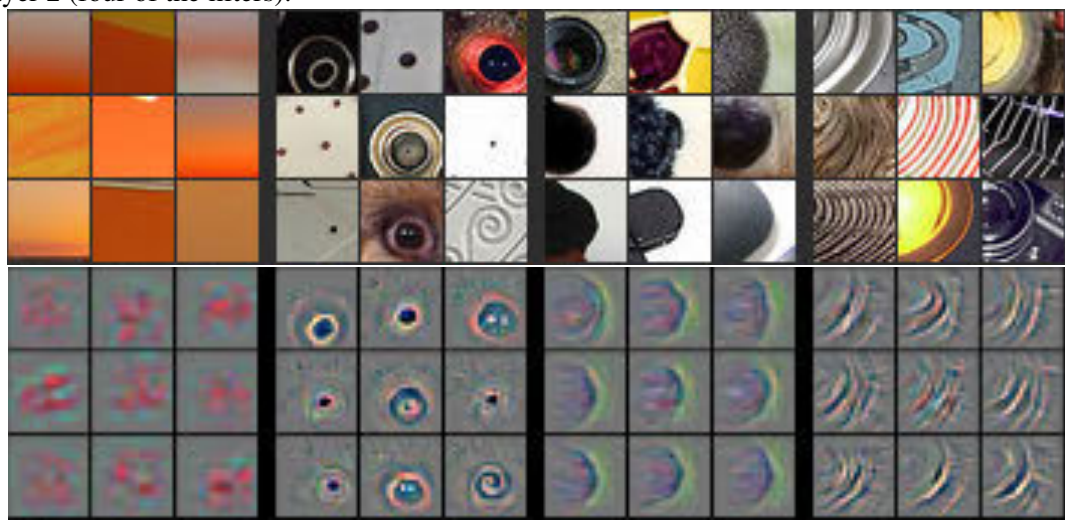


gabor.pdf (Bishop, Figure 10.11) [Gabor filters. *Not* learned; these are math functions.]

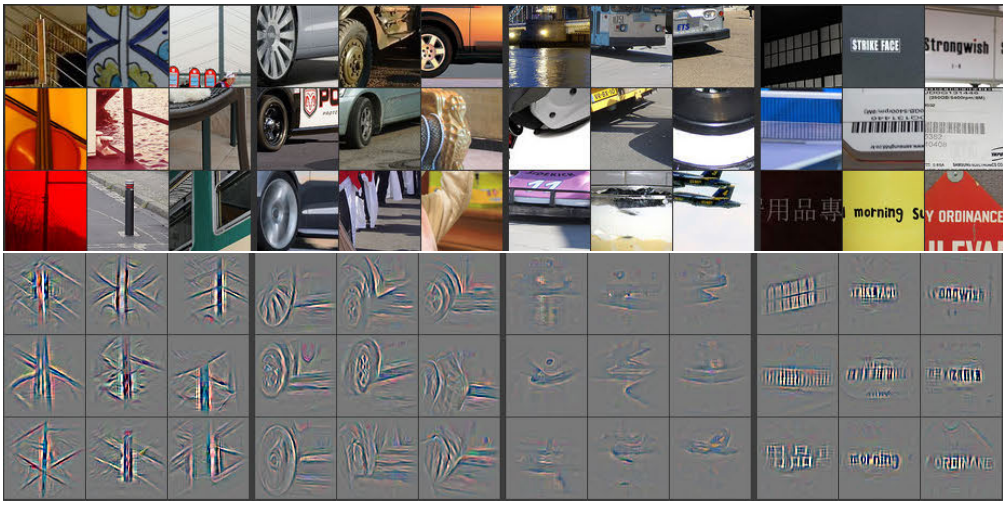
[AlexNet learned some simple color-specific edge detectors, but I find it noteworthy that the higher-frequency texture detectors are not sensitive to color at all. Apparently, the CNN decided it can separate fine texture from color.]

[Unfortunately, we can't just draw the filters learned by subsequent convolutional layers, because they're 3D arrays that don't carry much visual information. Instead, Zeiler and Ferguson (2013) have a technique where they determine which patches from the training set most trigger a particular filter, and they draw nine of those. They also have a technique for determining which pixels of those patches are most relevant to triggering the filter, and they plot the relevance of each patch pixel. This reveals that in the third example for convolutional layer 4, the filter is primarily responding to the grass in these images.]

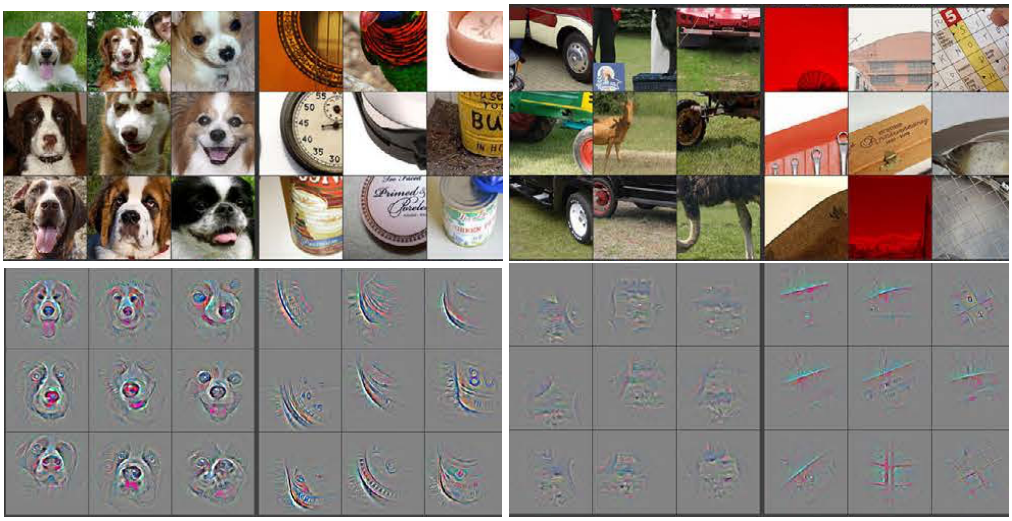
Conv layer 2 (four of the filters):



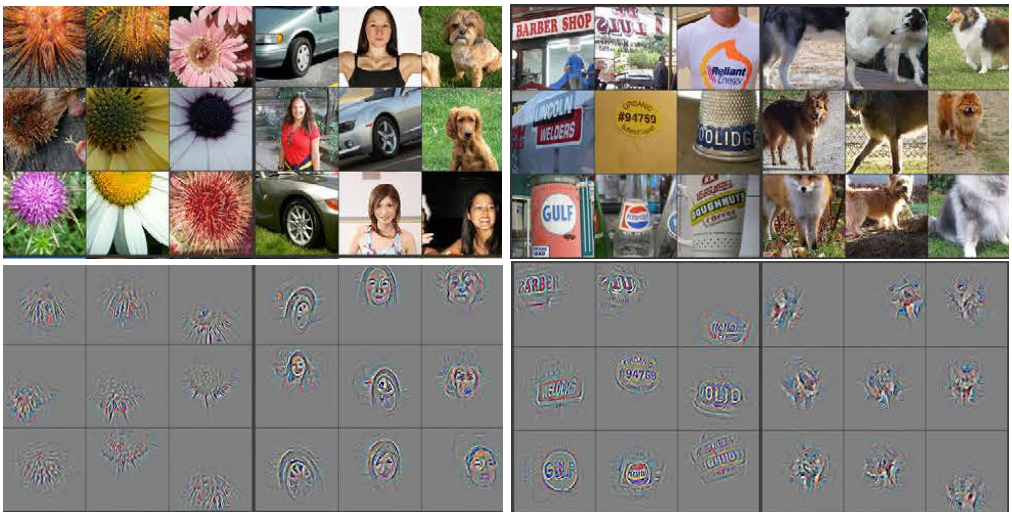
Conv layer 3:



Conv layer 4:



Conv layer 5:



The V1 Visual Cortex

[The idea to exploit local connectivity in CNNs was inspired by the human visual system, as well as by techniques used in image processing.]

[Show slides on the visual cortex, 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 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. It's fascinating, and surely not a coincidence, that humans and CNNs for vision both have edge detectors in their early layers.]

[The complex cells act as location-independent line detectors by taking inputs from many simple cells, which are location dependent. It's reminiscent of max pooling.]

[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., locality preserving. This is a big part of the inspiration for convolutional neural networks!]

[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.]

20 Unsupervised Learning: Principal Components Analysis

UNSUPERVISED LEARNING

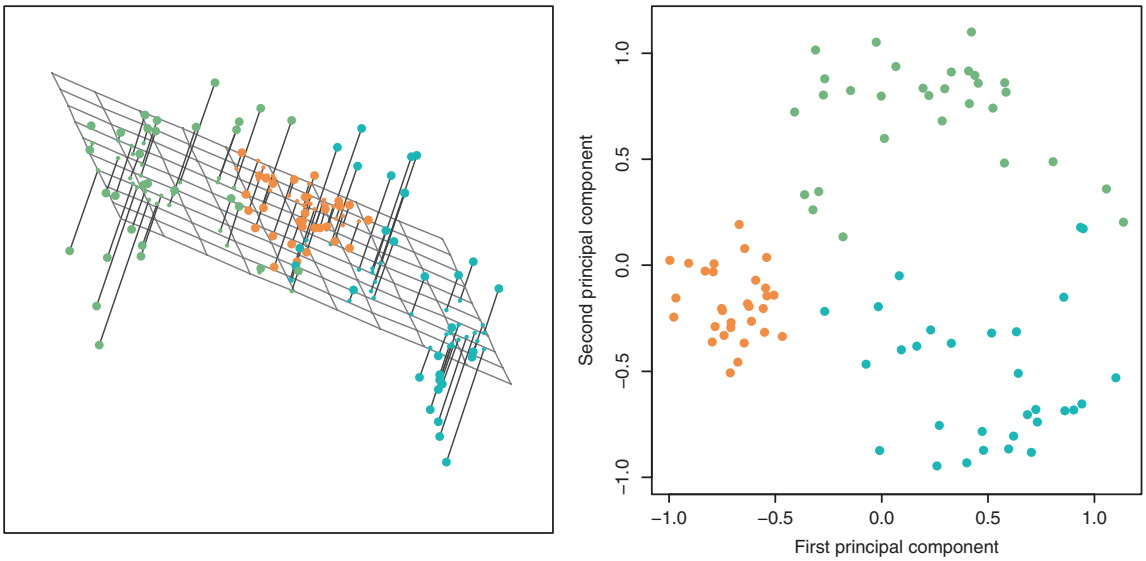
We have sample points, but no labels!
No classes, no y-values, nothing to predict.
Goal: Discover structure in the data.

Examples:

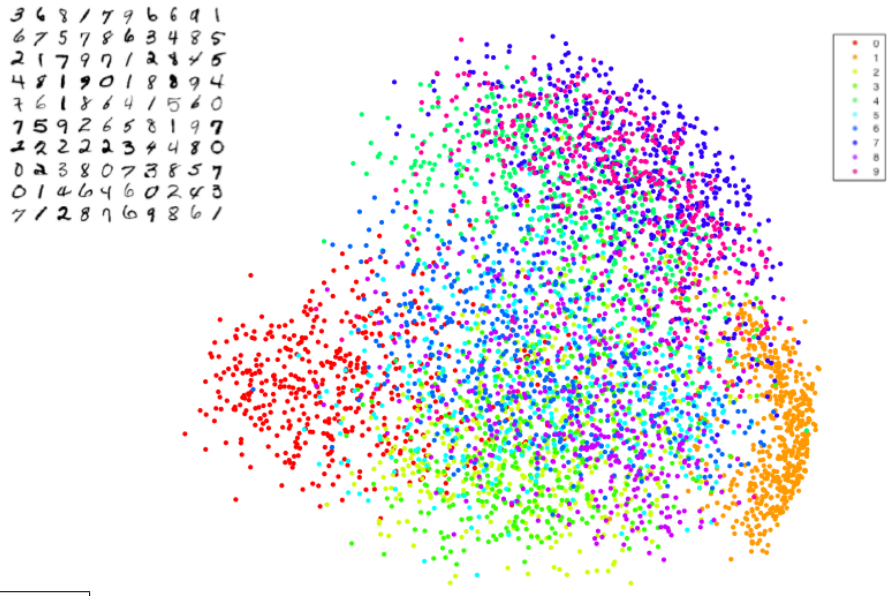
- Clustering: partition data into groups of similar/nearby points.
- Dimensionality reduction: data often lies near a low-dimensional subspace (or manifold) in feature space; matrices have low-rank approximations.
  [Whereas clustering is about grouping similar sample points, dimensionality reduction is 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.]

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 (ISL, Figure 12.2) [Example of 3D points projected to 2D by PCA. The 3D space on the left is the feature space, and the 2D space on the right is the principal component space.]



pcadigits.pdf [The (high-dimensional) MNIST digits projected to a 2D subspace (from 784D). 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.]

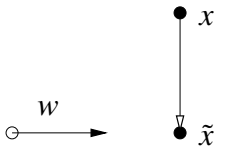
Why?

- Reducing # of dimensions makes some computations cheaper, e.g., regression.
- Remove irrelevant dimensions to reduce overfitting in learning algs.
  Like subset selection, but the "features" aren't axis-aligned;
  they're linear combos of input features.
- Find a small basis for representing variations in complex things, e.g., faces, genes.

[Sometimes PCA is used as a preprocess before regression or classification for the first two reasons.]

Let X be $n \times d$ design matrix. [No fictitious dimension.]
From now on, assume X is centered: $\sum_i X_i = 0$. (Replace X with $\dot{X}$.)
[We center the data in the usual way: by computing the sample mean, then subtracting the sample mean from each sample point.]

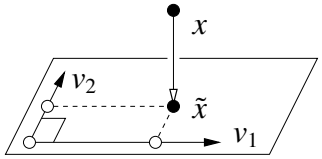
[Let's start by seeing what happens if we pick just one principal direction.]
Let w be a unit vector.
The orthogonal projection of point x onto vector w is $\tilde{x} = (x \cdot w)\, w$.



If w not unit, $\tilde{x} = \dfrac{x \cdot w}{\|w\|^2} w$.

[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.]

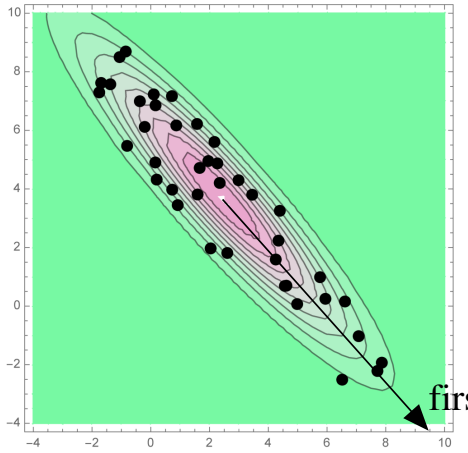
Given orthonormal directions $v_1, \ldots, v_k$, $\tilde{x} = \sum_{i=1}^{k} (x \cdot v_i)\, v_i$.
[The word "orthonormal" means they're all mutually orthogonal and all have length 1.]



Often we want just the k principal coordinates $x \cdot v_i$ in principal component space.
[Often we don't actually want the projected point in $\mathbb{R}^d$. Sometimes we do, but often we just want the principal coordinates.]

$X^\top X$ is square, symmetric, positive semidefinite, $d \times d$ matrix. [As it's symmetric, its eigenvalues are real.]
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. These are the principal components.
[. . . and the most important principal components 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.



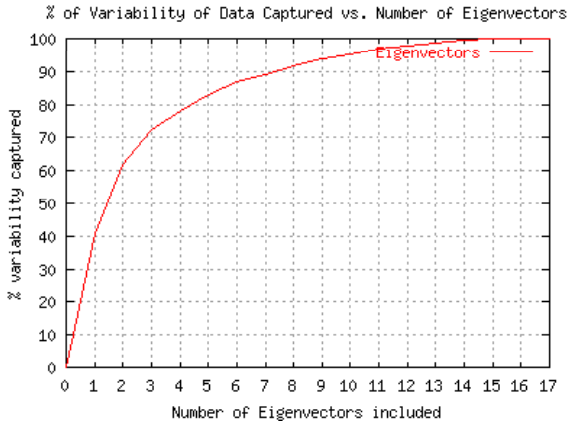
MLEPCA.pdf [A Gaussian fitted to sample points. The principle component with the greatest eigenvalue is drawn.]

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. In particular, you should never normalize image pixels individually.]
- Compute unit eigenvectors/values of $X^\top X$.

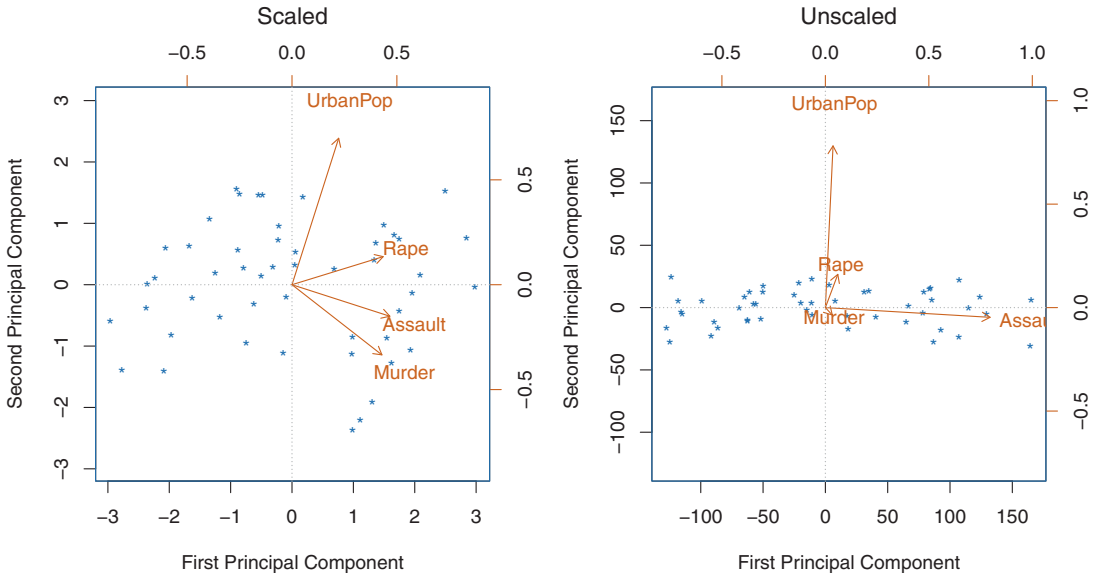
– Choose k. (Optional: based on the eigenvalue sizes.)
– For the best k-dimensional subspace, pick eigenvectors $v_{d-k+1}, \ldots, v_d$.
– Compute the k principal coordinates $x \cdot v_i$ of each training/test point.
[When we do this projection, we have two choices: we can project the original, un-centered training data OR we can project the centered training data. But if we do the latter, we have to translate the test data by the same vector we used to translate the training data when we centered it.]

[End of algorithm.]



$$\% \text{ of variability } = \frac{\sum_{i=d-k+1}^{d} \lambda_i}{\sum_{i=1}^{d} \lambda_i}$$

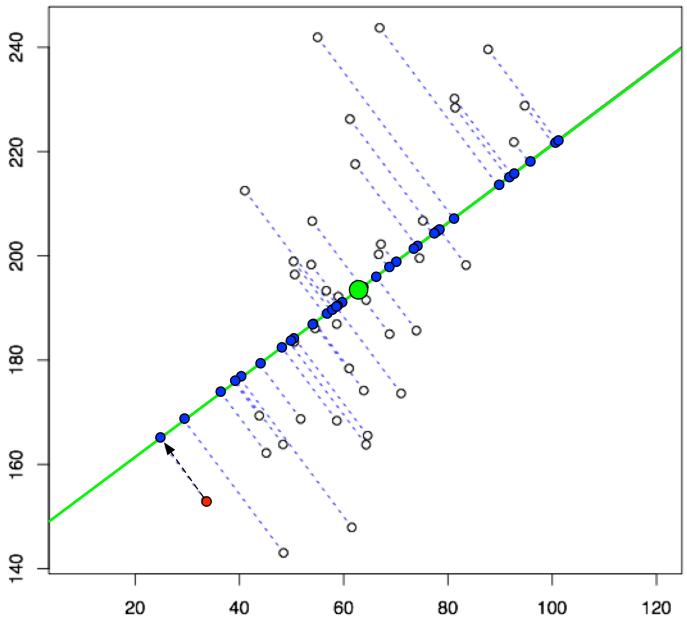
variance.pdf [Plot of # of eigenvectors vs. percentage of sample variance captured for a 17D data set. In this example, just 3 eigenvectors capture 70% of the variance.]



normalize.pdf (ISL, Figure 12.4) [Projection of 4D data onto a 2D subspace. Each point represents one metropolitan area. Normalized data at left; unnormalized data at right. The arrows show the four original axes projected on the 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 larger influence.]

[If you are using PCA as a preprocess for a supervised learning algorithm, there's a more effective way to choose k: validation.]

PCA derivation 2: Find direction w that maximizes sample 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.]

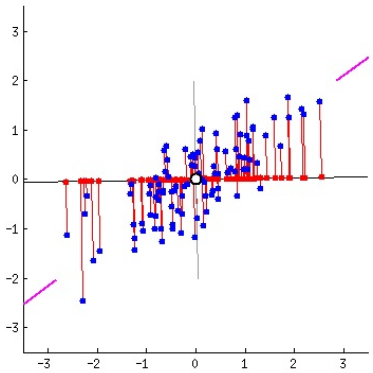


project.jpg [Points projected on a line. We wish to choose the orientation of the green line to maximize the sample variance of the blue points.]

$$\text{Find } w \text{ that maximizes } \mathrm{Var}(\{\tilde{X}_1, \tilde{X}_2, \ldots, \tilde{X}_n\}) = \frac{1}{n}\sum_{i=1}^{n}\left(X_i \cdot \frac{w}{\|w\|}\right)^2 = \frac{1}{n}\frac{\|Xw\|^2}{\|w\|^2} = \frac{1}{n}\underbrace{\frac{w^\top X^\top X w}{w^\top w}}_{\text{Rayleigh quotient of } X^\top X \text{ and } w}$$

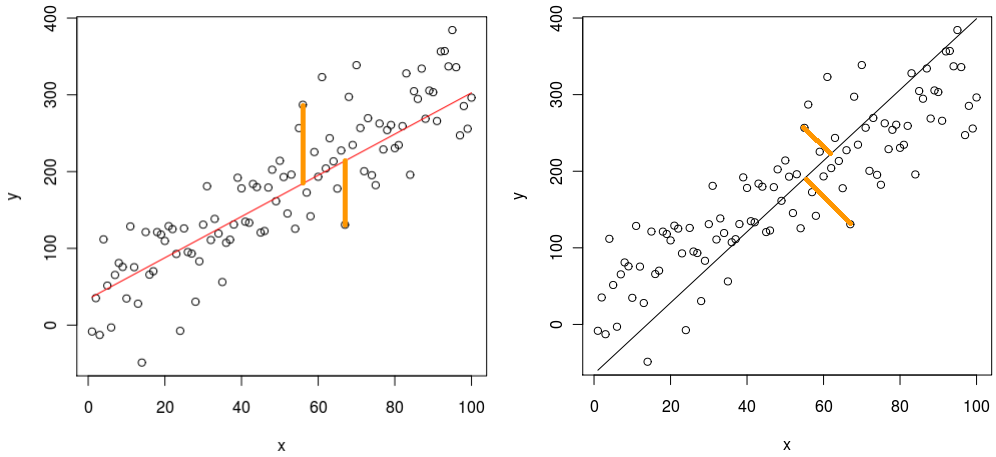
[This fraction is a well-known construction called the Rayleigh quotient. When you see it, you should smell eigenvectors nearby. How do we maximize this?]
If w is an eigenvector v_i of $X^\top X$, Ray. quo. $= \lambda_i$
$\rightarrow$ of all eigenvectors, v_d achieves maximum variance λ_d/n.
One can 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 have 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 second direction that's orthogonal to the best direction. But subject to that constraint, we again pick the direction that maximizes the sample variance.]
What if we constrain w to be orthogonal to v_d? Then v_{d-1} is optimal.
[And if we need a third direction orthogonal to v_d and v_{d-1}, the optimal choice is v_{d-2}. And so on.]

PCA derivation 3: Find direction w that minimizes mean squared projection distance.



PCAanimation.gif [This is an animated GIF; unfortunately, the animation doesn't work 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.]

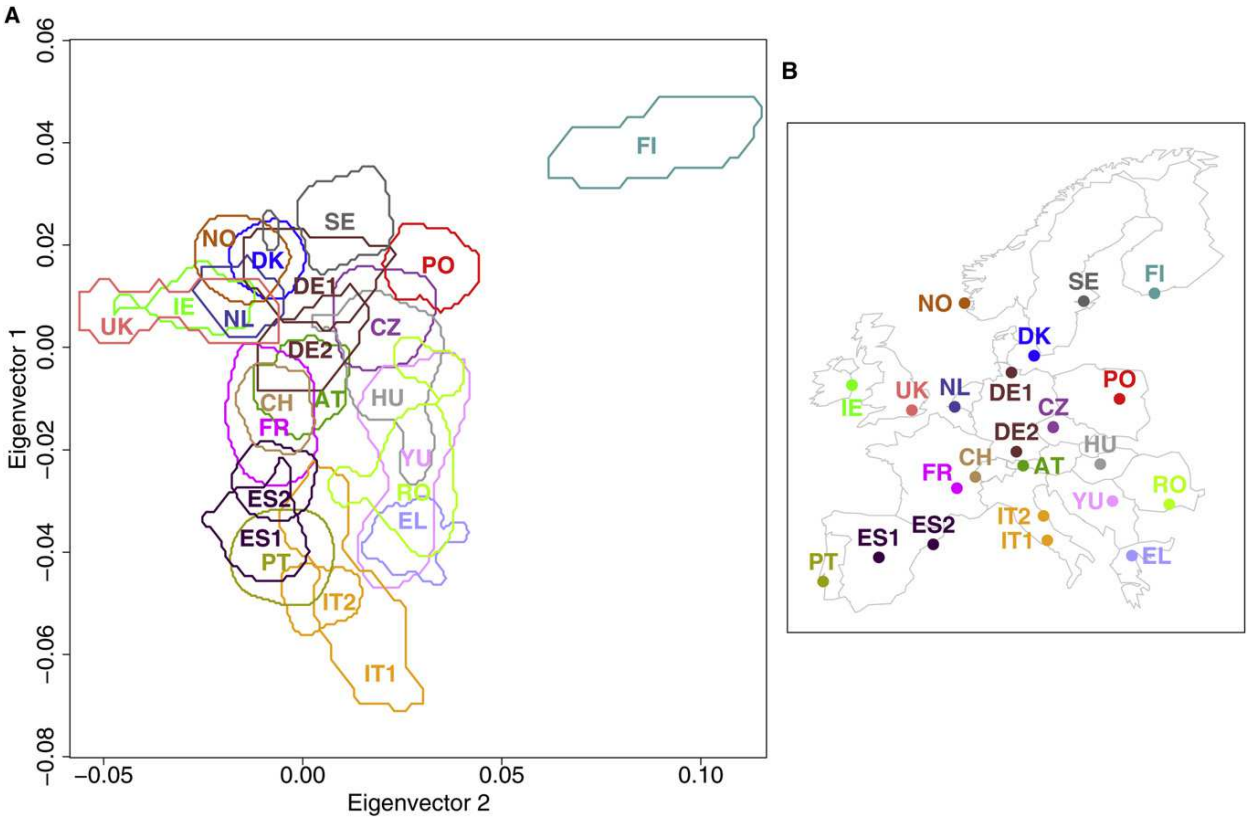
[You can think of this as a sort of least-squares linear regression, with one subtle but important change. Instead of measuring the error in a fixed vertical direction, we're measuring the error in a direction orthogonal to the principal component direction we choose.]



projlsq.png, projpca.png [Least-squares linear regression vs. PCA. In linear regression, the projection direction is always "vertical" (measured in the label coordinate); 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.]

$$\text{Find } w \text{ that minimizes } \sum_{i=1}^{n} \left\|X_i - \tilde{X}_i\right\|^2 = \sum_{i=1}^{n} \left\|X_i - \frac{X_i \cdot w}{\|w\|^2} w\right\|^2 = \sum_{i=1}^{n} \left(\|X_i\|^2 - \left(X_i \cdot \frac{w}{\|w\|}\right)^2\right)$$
$$= \text{constant} - n\,(\text{variance from derivation 2}).$$

Minimizing mean squared projection distance = maximizing variance.
[From this point, carry on with the same reasoning as derivation 2.]



europegenetics.pdf (Lao et al., Current Biology, 2008.) [Illustration of the first two principal components of the single nucleotide polymorphism (SNP) matrix for the genes of various Europeans. The design matrix has 2,547 people from these locations in Europe (right), and 309,790 SNPs per person. Each SNP is binary, so think of it as 309,790 dimensions of zero or one. The output (left) shows spots on the first two principal components where there was a high density of projected people from a particular national type. What's amazing about this is how closely the projected genotypes resemble the geography of Europe.]

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.]

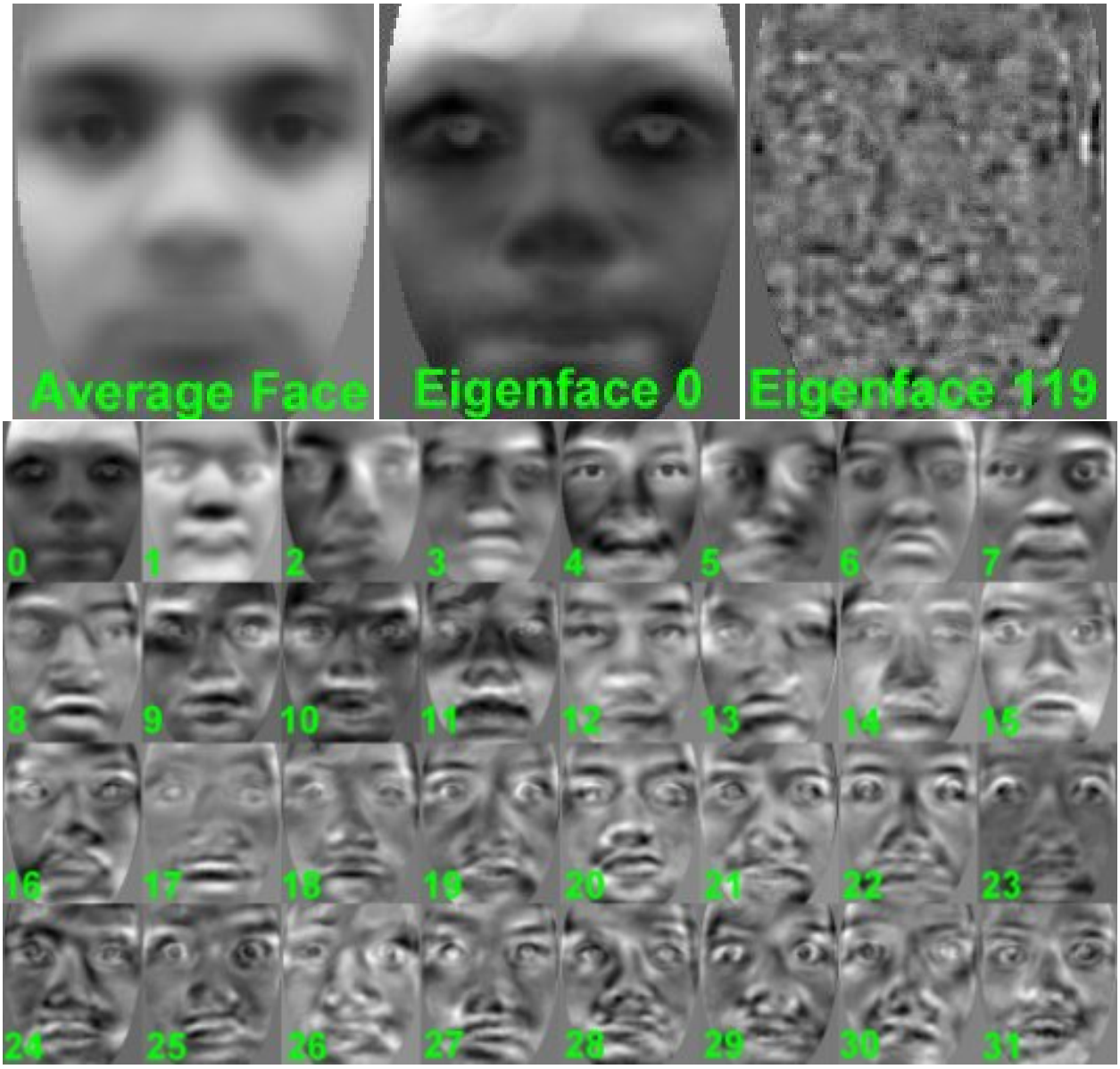
Face recognition: Given a query face, compare it to all training faces; find nearest neighbor in $\mathbb{R}^d$.

[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 neighbors 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.]

[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 principal coordinates instead of 20 million pixels.]



facerecaverage.jpg, facereceigen0.jpg, facereceigen119.jpg, facereceigen.jpg [Images of the the eigenfaces with the 32 largest eigenvalues. The "average face" is the mean used to center the data.]

eigenfaceproject.pdf [Images of a face (left) projected onto the first 4 and 50 eigenvectors, with the average face added back. The 50-eigenvector image is blurry but good enough for face recognition. (These projections are in feature space, not in principle components space. The principle coordinates are what you would use in the nearest neighbor classifier.)]

For best results, equalize the intensity distributions first.

facerecequalize.jpg [Image equalization.]

[Eigenfaces encode both face shape *and* lighting. 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.]

[Eigenfaces are not a state-of-the-art face recognition algorithm, not even close. But inspecting these images can give you some intuition about PCA.]

[Optional: 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 with PCA: 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 talked about the eigendecomposition of a square, symmetric matrix. Unfortunately, nonsymmetric matrices don't have nice eigendecompositions, 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: Every $X \in \mathbb{R}^{n \times d}$ has a singular value decomposition $X = UDV^\top$ of the form

$$X = U D V^\top = \sum_{i=1}^{\min\{n,d\}} \underbrace{\delta_i u_i v_i^\top}_{\text{rank 1 outer product matrix}}$$

X ($n \times d$) $=$ U ($n \times n$, columns $u_1, \ldots, u_n$) D ($n \times d$, diagonal: $\delta_1, \delta_2, \ldots, \delta_d$, zeros elsewhere) $V^\top$ ($d \times d$, rows $v_1, \ldots, v_d$)

$U^\top U = I = UU^\top$
orthonormal u_i's are left singular vectors of X

$V^\top V = I = VV^\top$
orthonormal v_i's are right singular vectors of X

[Draw this by hand; write summation at the right last. fullsvd.pdf]

Diagonal entries $\delta_1, \ldots, \delta_{\min\{n,d\}}$ of D are nonnegative singular values of X.

[By convention, singular values are never negative, but some of them might be zero.]

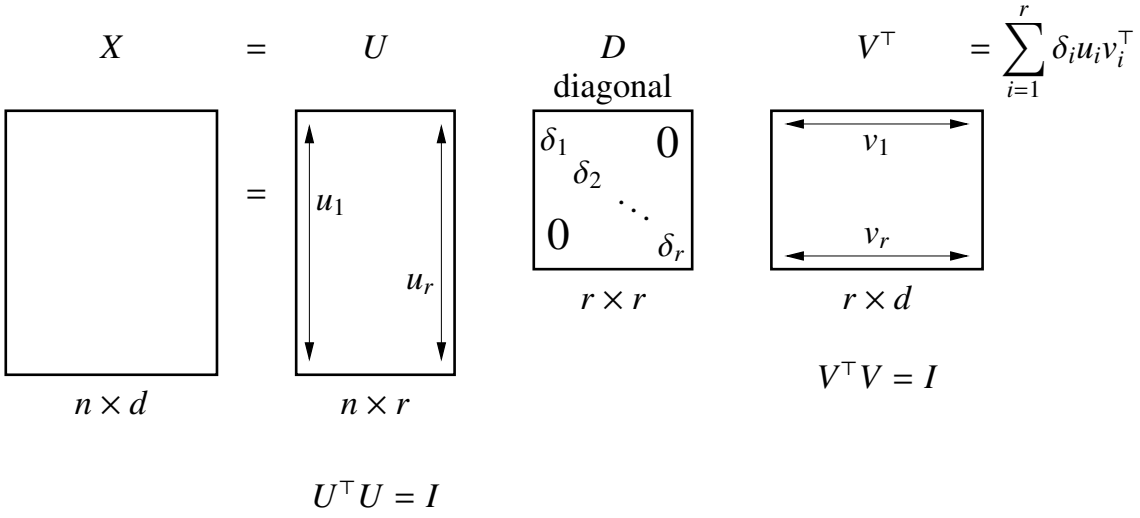
Fact: v_i is an eigenvector of $X^\top X$ w/eigenvalue δ_i^2, so the SVD solves PCA.
Proof: $X^\top X = VD^\top U^\top UDV^\top = VD^\top DV^\top$
which is an eigendecomposition of $X^\top X$, with each δ_i^2 on the diagonal of $D^\top D$.

[The columns of V are the eigenvectors of $X^\top X$, which are the principal components we need for PCA. The SVD also tells us their eigenvalues, which are the squares of the singular values of X. That's related to why the SVD is more numerically stable: the ratios between singular values are smaller than the ratios between eigenvalues, so it's easier to stably compute the singular values of X than the eigenvalues of $X^\top X$.]

Important: Row i of UD gives the principal coordinates of sample point X_i (i.e., $\forall i, \forall j, X_i \cdot v_j = \delta_j U_{ij}$).
[So we don't need to explicitly compute the inner products $X_i \cdot v_j$; the SVD has already done it for us.]
Proof: $XV = UDV^\top V = UD$, so $(XV)_{ij} = (UD)_{ij}$.

The Compact SVD

Singular vectors with singular value zero are useless for PCA [and many other applications], motivating the compact SVD. Let r be the rank of X.



[Draw this by hand. compsvd.pdf]

$\delta_1, \ldots, \delta_r$ are nonzero singular values of the full SVD, & have the same left/right singular vectors.

[There are r nonzero singular values, and we can express X with their singular vectors alone. A nice consequence is that D is invertible. If X is a centered design matrix for sample points that all lie on a line, then X has rank 1 and there is only one nonzero singular value. If the centered sample points span a subspace of dimension r, X has rank r and there are r nonzero singular values.]

[We might save a fair amount of time by not computing the left and right singular vectors with singular value zero. Observe that the columns of U are still orthogonal, but it is no longer true that $UU^\top = I$. The same goes for V.]

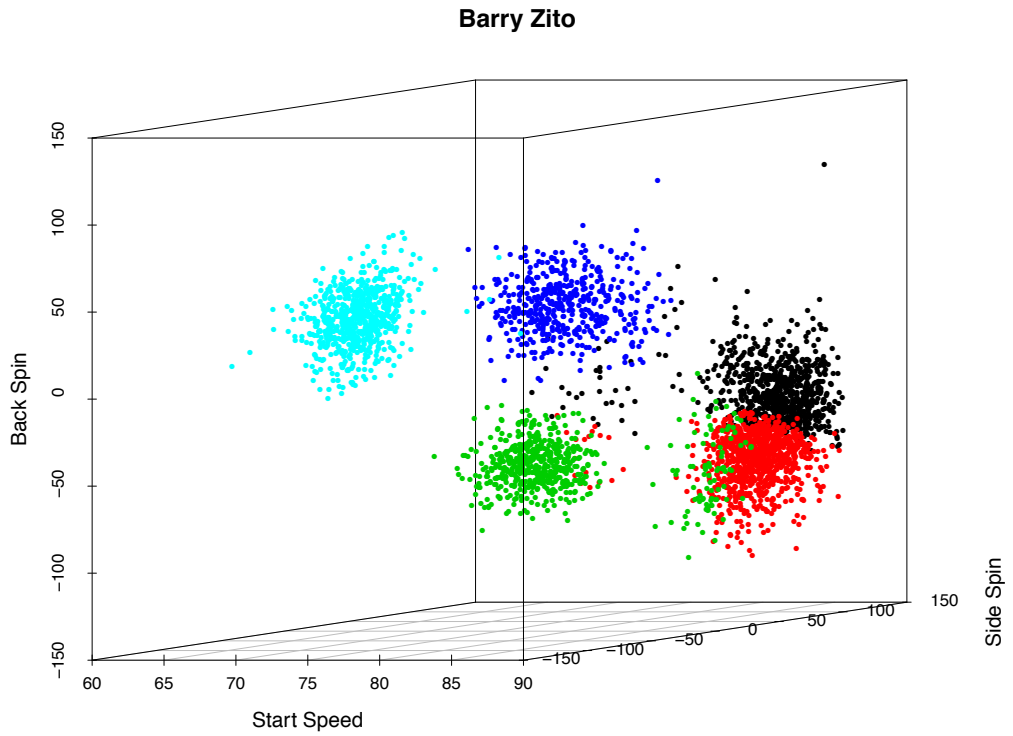
Fact: We can find the k greatest singular values & corresponding vectors in $O(ndk)$ time.
[So we can save time by computing some of the singular vectors without computing all of them.]
[There are approximate, randomized algorithms that are even faster, producing an approximate SVD in $O(nd \log k)$ time. These are starting to become popular in algorithms for very big data.]
[https://code.google.com/archive/p/redsvd/]

[In the next lecture, we will use the compact SVD to help us understand the Moore-Penrose pseudoinverse and its application to least-squares linear regression.]

CLUSTERING

Partition data into clusters so points in 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 (from a talk by Michael Pane) [k-means 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 sample point X_i a cluster label $y_i \in [1, k]$.
Cluster i's mean is $\mu_i = \dfrac{1}{n_i} \sum_{y_j=i} X_j$, given n_i points in cluster i.

Find y that minimizes $\displaystyle\sum_{i=1}^{k} \sum_{y_j=i} \|X_j - \mu_i\|^2$. [Sum of squared distances from points to their cluster means.]

NP-hard. Solvable in $O(nk^n)$ time. [Try every partition.]

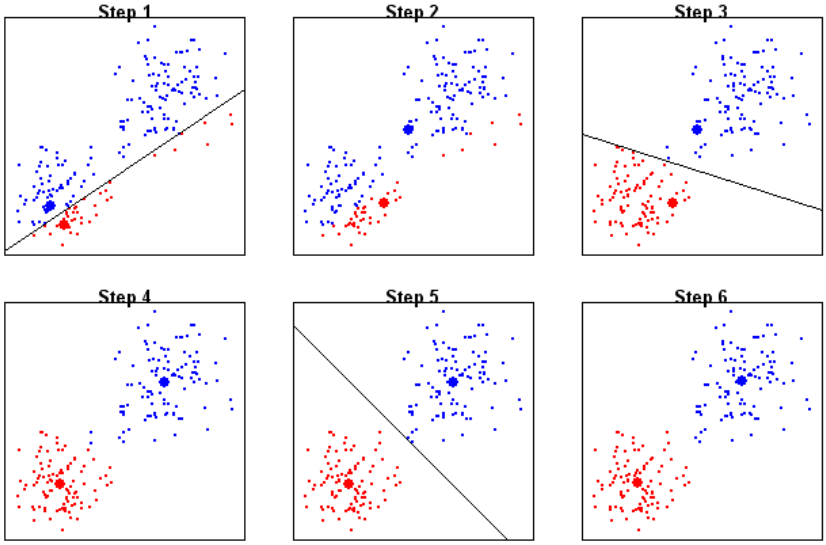
k-means heuristic: Alternate between
(1) y_j's are fixed; update μ_i's
(2) μ_i's are fixed; update y_j's
Halt when step (2) changes no assignments.

[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.]

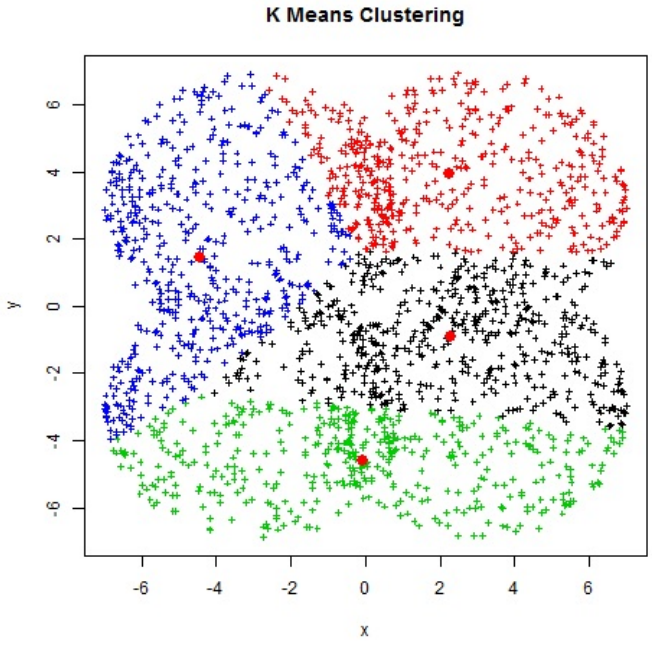
Step (1): One can show (calculus) the optimal μ_i is the mean of the points in cluster i.
[This is easy calculus, so I leave it as a short exercise.]

Step (2): The optimal y assigns each point X_j to the closest mean μ_i.
[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.]

[. . . so both steps minimize the cost function, but they don't optimize all the variables at once.]

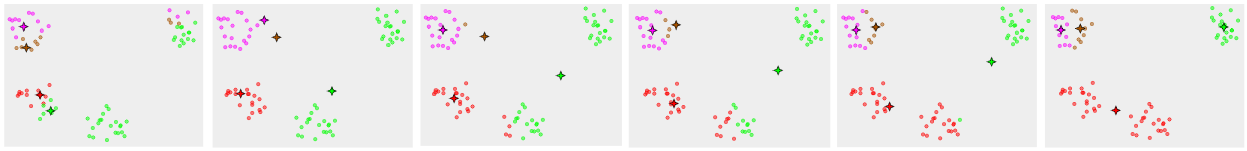


2means.png [An example of 2-means. Odd-numbered steps reassign the data points. Even-numbered steps compute new means.]



4meansanimation.gif [This is an animated GIF of 4-means with many points. Unfortunately, the animation doesn't work 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 says that Lloyd's algorithm never loops forever. But it 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 don't 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.]



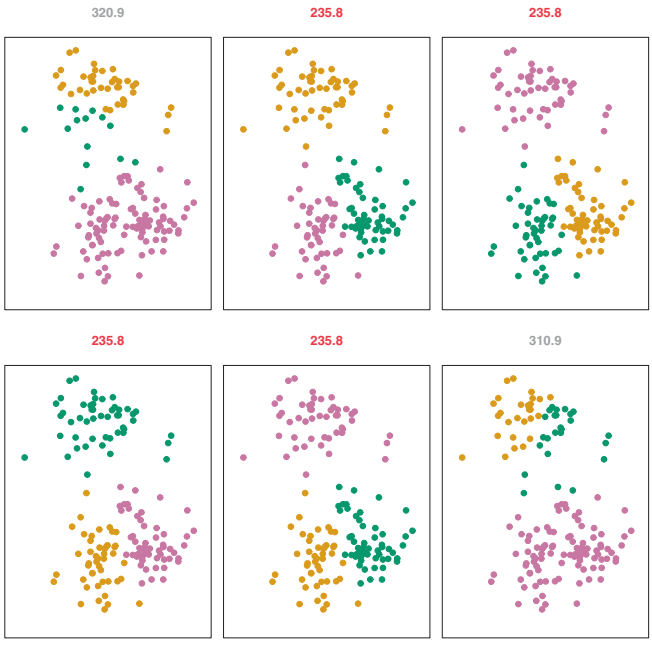
4meansbad.png [An example where 4-means clustering fails.]

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).
- k-means++: like Forgy, but biased distribution.
  [Each μ_i is chosen with a preference for points far from previous μ_i's.]

[k-means++ is a little more work, but it works well in practice and theory. Forgy seems to be better than random partition, 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, each time starting with a different random partition. The algorithm finds three different local minima.]

[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 $\displaystyle\sum_{i=1}^{k} \frac{1}{n_i} \sum_{y_j=i} \sum_{y_m=i} \|X_j - X_m\|^2$

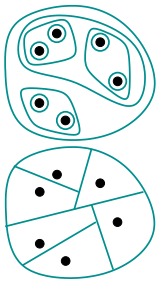
[At the minimizer, this objective function is equal to twice the previous one. It's a worthwhile exercise to show that—it's harder than it looks. The nice thing about this expression is that it doesn't include the means; it's a function purely of the sample points and the clusters we assign them to. So it's more compelling.]

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 usually shouldn't, but it depends on context.]

[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.

Top-down, aka divisive clustering:
start with all pts in one cluster; repeatedly split it.

[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.]

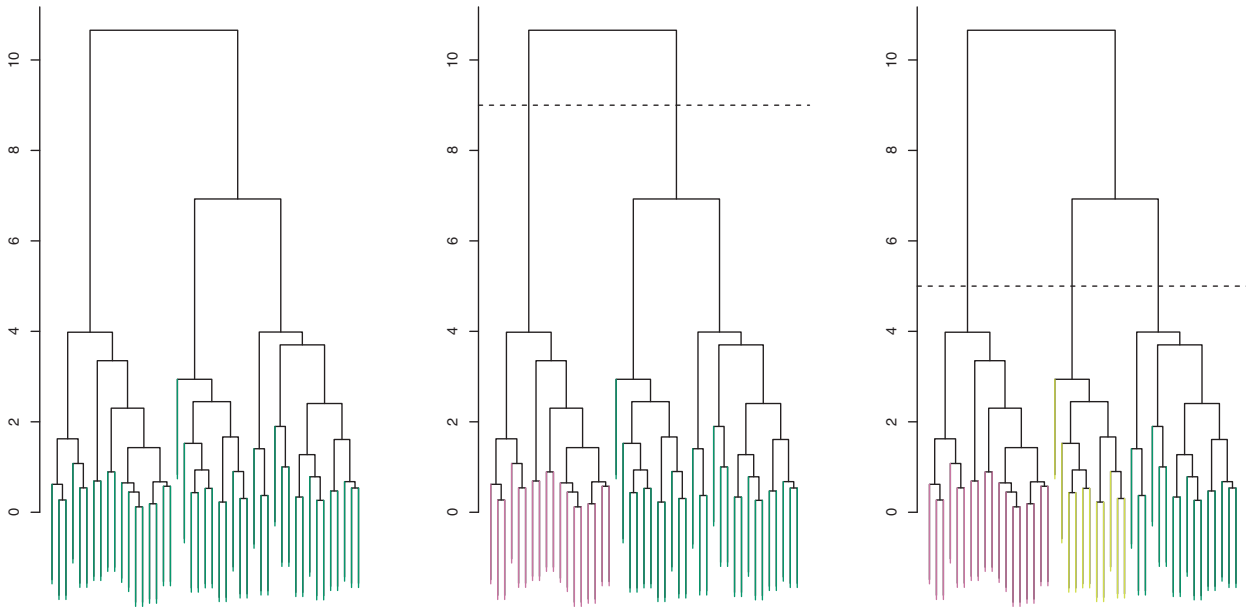
We need a distance fn for clusters A, B:

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 sample points. The centroid linkage only really makes sense if we're using the Euclidean distance.]

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. A package called ELKI has publicly available implementations.]

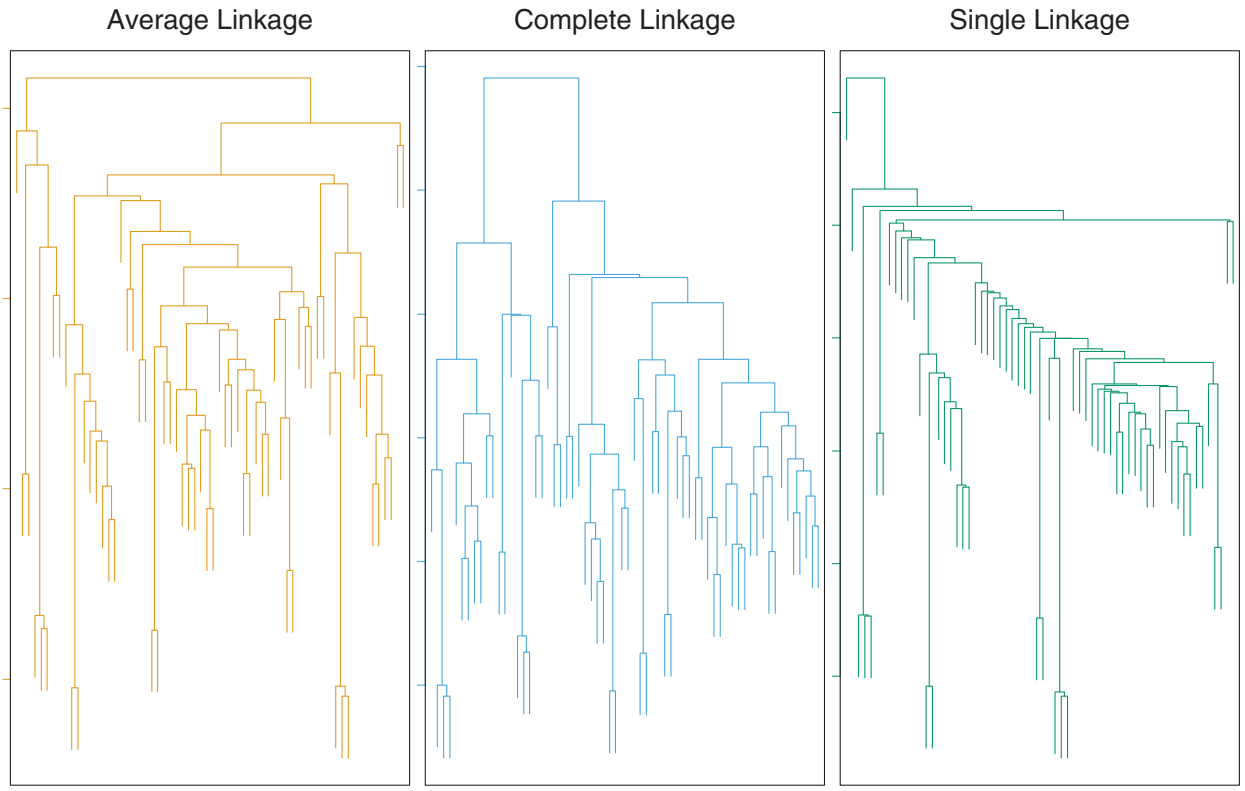
Dendrogram: Illustration of the cluster hierarchy (tree) in which the vertical axis encodes all the linkage distances.



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 treenode's left subtree and right subtree and it would still be the same dendrogram. It doesn't 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 sample point. 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 farthest 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 applications 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 sample 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 The Pseudoinverse; Better Generalization for Neural Nets

THE PSEUDOINVERSE AND THE SVD

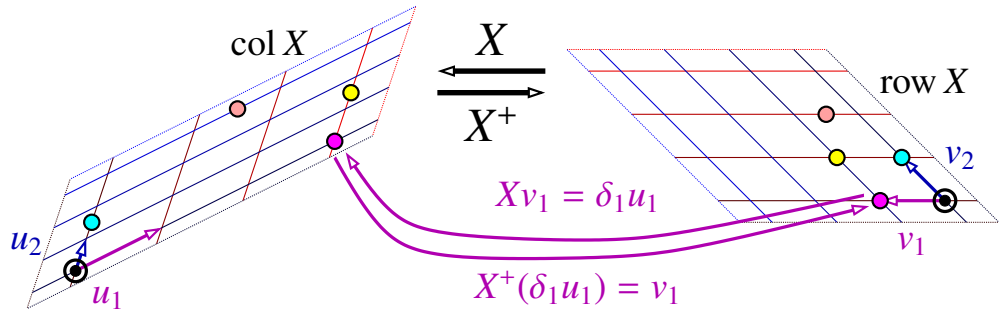
[We're done with unsupervised learning. For the rest of the semester, we go back to supervised learning.]

[The singular value decomposition can give us insight into the pseudoinverse and its use in least-squares linear regression. If you attended Discussion Section 6, you worked through an explanation of this, but now that I've introduced the compact SVD in Lecture 21, I'd like to summarize it.]

Let X be any $n \times d$ matrix. Let $X = UDV^\top$ be its *compact* SVD. Let $r = \operatorname{rank} X$.
Recall that $U \in \mathbb{R}^{n\times r}$, $D \in \mathbb{R}^{r\times r}$ is diagonal & invertible, $V \in \mathbb{R}^{d\times r}$, $U^\top U = I$, $V^\top V = I$.
The Moore–Penrose pseudoinverse of X is $X^+ = VD^{-1}U^\top$. It's $d \times n$.
[This is a better pseudoinverse than the one I defined in Lecture 10, not least because it's always defined.]

Observe:
(1) $XX^+ = UU^\top$, which is symmetric. Proof: $XX^+ = UDV^\top VD^{-1}U^\top = UDD^{-1}U^\top = UU^\top$.
(2) $X^+X = VV^\top$. [The proof is analogous to (1).]
(3) If $r = n$, then $XX^+ = I_{n\times n}$ and X^+ is a right inverse. Proof: U is square, $U^\top U = I \to UU^\top = I$; use (1).
(4) If $r = d$, then $X^+X = I_{d\times d}$ and X^+ is a left inverse. [The proof is analogous to (3) and uses (2).]
(5) By (3), **if X is invertible ($r = n = d$), $X^+ = X^{-1}$. [The pseudoinverse is the inverse when one exists.]**
(6) These are compact SVDs: $X^+ = VD^{-1}U^\top$, $X^\top = VDU^\top$, $(X^+)^\top = UD^{-1}V^\top$.
[If a factorization has the form of a compact SVD, it *is* a compact SVD.]
X^+ is like $X^\top$ with the nonzero singular values inverted.
(7) Given a compact SVD $X = UDV^\top$, $\operatorname{null} X = \operatorname{null} V^\top$.
Proof: $V^\top w = 0 \to Xw = UDV^\top w = 0 \to D^{-1}U^\top UDV^\top w = 0 \to V^\top w = 0$.
(8) By (6) & (7), $\operatorname{null} X^+ = \operatorname{null} U^\top = \operatorname{null} X^\top$ and $\operatorname{null}(X^+)^\top = \operatorname{null} V^\top = \operatorname{null} X$.
So row X^+ = col X and col X^+ = row X. **X^+ has the same four fundamental subspaces as $X^\top$.**
(9) (1) & (2) give eigendecompositions: $XX^+ = \begin{bmatrix} U & U_{\text{null}} \end{bmatrix} \begin{bmatrix} I_{r\times r} & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} U^\top \\ U_{\text{null}}^\top \end{bmatrix}$, $X^+X = \begin{bmatrix} V & V_{\text{null}} \end{bmatrix} \begin{bmatrix} I_{r\times r} & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} V^\top \\ V_{\text{null}}^\top \end{bmatrix}$.
[U_{null} and V_{null} have orthonormal column vectors spanning the null spaces of XX^+ and X^+X.]
(10) By (8) & (9), all have rank r: X, U, V, X^+, XX^+, X^+X.
(11) By (9), every $w \in \operatorname{col} U$ is an eigenvector of XX^+ with eigenvalue 1; all other eigenvalues are 0.
As $\operatorname{col} U = \operatorname{col} X$, XX^+ **is identity map on col X.** [Symmetrically,] X^+X **is identity map on row X.**

[In summary, the psuedoinverse is as close to an inverse of X as anything can be. Let's visualize what the pseudoinverse does. When you apply X to a vector in $\operatorname{row} X$, you get a vector in $\operatorname{col} X$; then when you apply X^+ to the result, you get the original vector back.]



rowcol.pdf [The singular vectors are perpendicular, but we are viewing the planes from oblique angles.]

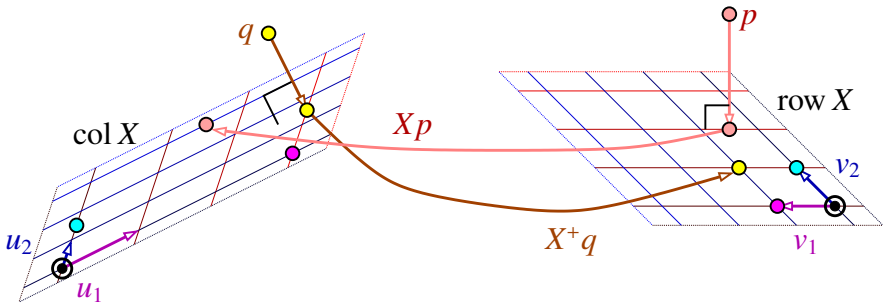
[If we think of X as a linear function that maps row X to col X, and we ignore the other dimensions of $\mathbb{R}^d$ and $\mathbb{R}^n$, then that linear function is a bijection. The inverse of that bijection is the pseudoinverse X^+.]

Linear function $f : \text{row}\, X \to \text{col}\, X, p \mapsto Xp$ is a bijection.
Its inverse is $f^{-1} : \text{col}\, X \to \text{row}\, X, q \mapsto X^+q$.

The r right singular vectors v_i are an orthonormal basis for row X.
The r left singular vectors u_i are an orthonormal basis for col X.
$Xv_i = \delta_i u_i$. $\quad X^+u_i = \frac{1}{\delta_i} v_i$.
[X maps each right singular vector to some scalar multiple of the corresponding left singular vector. The corresponding singular value tells us how much longer the vector gets when we map it. X^+ maps each left singular vector to some scalar multiple of a right singular vector.]

[Usually we don't think of X as a function from row space to column space. Usually we think of X as a function from some bigger space $\mathbb{R}^d$ to a bigger space $\mathbb{R}^n$. In our figure above, X might be a 4×3 matrix, but its rank is only two. Then X isn't a bijection any more, and neither is its pseudoinverse X^+. So X maps every point in $\mathbb{R}^3$ to a point on the plane col X. When you map a three-dimensional space down to two dimensions, it can't be a bijection, so X doesn't have an inverse. Just a pseudoinverse.]

[You can think of X as a function that orthogonally projects a three-dimensional point down onto the row space of X, then uses the bijection above to finish the mapping. Symmetrically, you can think of X^+ as a function that orthogonally projects a four-dimensional point down onto the column space of X, then uses the inverse bijection. Here's an illustration of mapping p to Xp and q to X^+q.]



rowcolpr.pdf

[With the compact SVD, we can show that the pseudoinverse always gives a solution to least-squares linear regression, even when $X^\top X$ is singular.]

Theorem: A solution to the normal equations $X^\top Xw = X^\top y$ is $w = X^+y$.

Proof: $X^\top Xw = X^\top XX^+y = VDU^\top UDV^\top VD^{-1}U^\top y = VD^2D^{-1}U^\top y = VDU^\top y = X^\top y$.

If the normal eq'ns have multiple solutions, $w = X^+y$ is the least-norm solution; i.e., it minimizes $\|w\|$ among all solutions. [If you attended Discussion Section 6, you might have proven this yourself.]

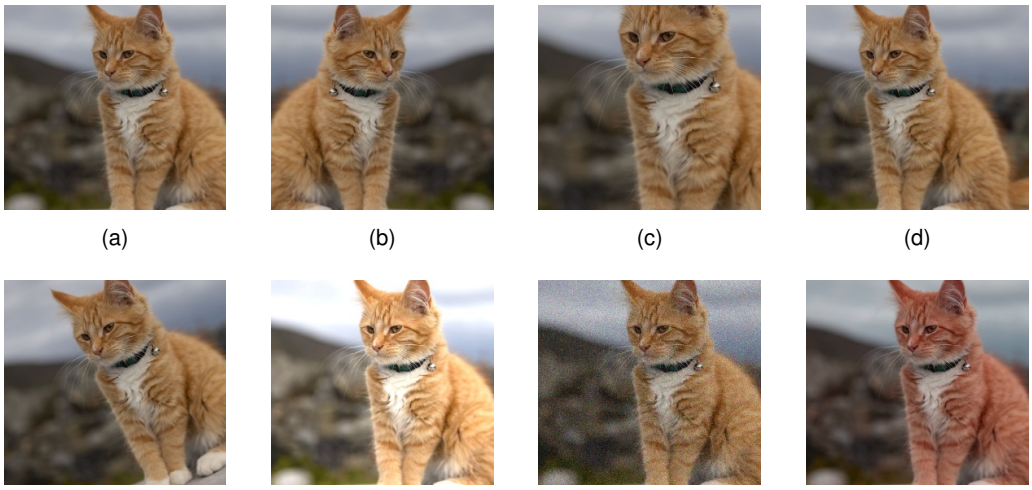
[This way of solving the normal equations is very helpful when $X^\top X$ is singular because $n < d$ or the sample points lie on a subspace of the feature space. But observe that if X has a very small singular value, the reciprocal of that singular value will be very large and have a very large effect on w; but when that singular value is exactly zero, it has no effect on w! So when we have a really tiny singular value, should we pretend it is zero? Ridge regression implements this policy to some degree; review Discussion Worksheet 12 for details.]

BETTER GENERALIZATION FOR NEURAL NETWORKS

[Classic methods for preventing overfitting, such as subset selection, ℓ_2 regularization, and ensembles of learners, sometimes help neural networks to generalize better to points they haven't been trained on.]

(1) Get more data. [This is the best method. Andrej Karpathy writes that "It is a very common mistake to spend a lot of engineering cycles trying to squeeze juice out of a small dataset when you could instead be collecting more data."]

(2) Data augmentation. Augment data set with modified versions of training points.



augmentation.pdf, (Bishop, Figure 9.1) [Examples of data augmentation applied to an original image (a). (b) Reflection. (c) Scaling. (d) Translation. (e) Rotation. (f) Changing brightness and contrast. (g) Added noise. (h) Color shift.]

[You can see that these augmentations do not change the fact that the image should be classified as a cat.]

pixmix.pdf, (Hendrycks et al., "PixMix", 2022) [More varieties of data augmentation.]

[Hendrycks et al. note that "For state-of-the-art models, data augmentation can improve clean accuracy [on the test set] comparably to a 10× increase in model size. Further, data augmentation can improve out-of-distribution robustness [on images from a distribution different than the training set] comparably to a 1,000× increase in labeled data."]

[One point they make is that while adding Gaussian noise is one augmentation that helps improve generalization to new images, it's even more effective to add artifacts that stimulate hidden units, such as the hidden units that detect edges in an image. So their augmentation methods mix images with other images that introduce spurious structure, not just Gaussian noise.]

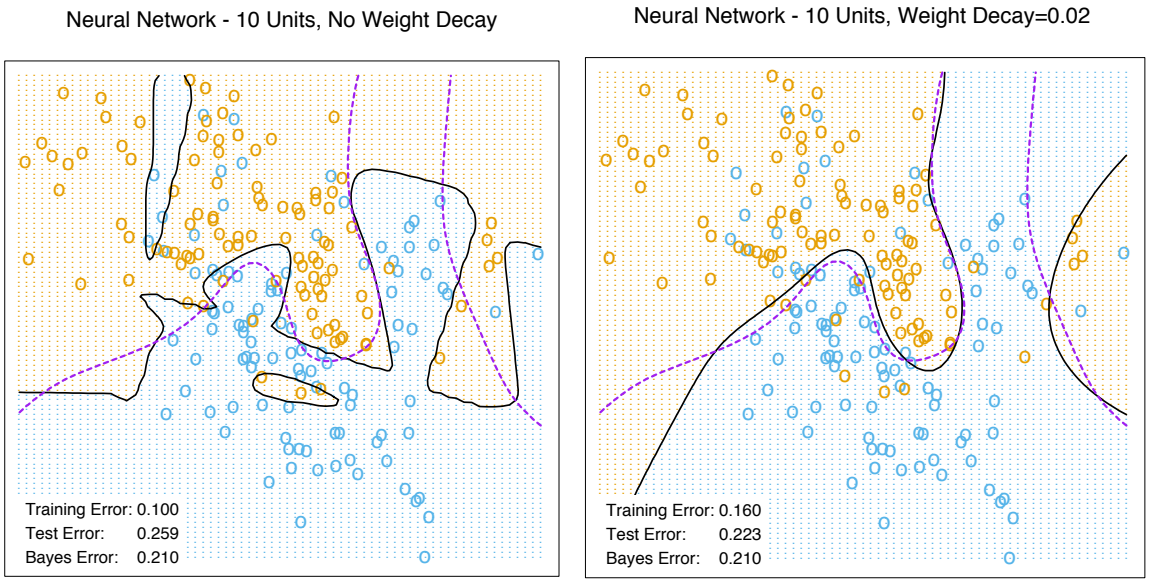
(3) Subset selection. [Recall Lecture 13.]

(4) ℓ_2 regularization, aka weight decay.
Add $\lambda \|w\|^2$ to the cost/loss fn, where w is vector of all weights in network.
[w includes all the weights in all the weight matrices, rewritten as a vector.]

[We regularize for the same reason we do it in ridge regression: we suspect that overly large weights are spurious.]

[With a neural network, it's not clear whether penalizing the bias terms is bad or good. Penalizing the bias terms has the effect, potentially positive, of drawing each ReLU or sigmoid unit closer to the center of its nonlinear operating region. I would suggest to try both ways and use validation to decide whether you should penalize the bias terms or not. Also, you could try using a different hyperparameter for the bias terms than the λ you use for the other weights.]

Effect: step $\Delta w_i = -\epsilon \dfrac{\partial J}{\partial w_i}$ has extra term $-2\epsilon\lambda\, w_i$
Weight w_i decays by factor $1 - 2\epsilon\lambda$ if not reinforced by training.



weightdecayoff.pdf, weightdecayon.pdf (ESL, Figure 11.4) Write "10 hidden units + softmax + cross-entropy loss." [Examples of 2D classification without (left) and with (right) weight decay. Observe that in the second example, the decision boundary (black) better approximates the Bayes optimal boundary (dashed purple curve).]

[AlexNet is a famous example of a network that used both ℓ_2 regularization and momentum. Just add the ℓ_2 penalty to the cost function J and plug that cost function into the momentum algorithm from Lecture 18. AlexNet set $\lambda = 0.0005$ and the momentum decay term to $\beta = 0.9$. They adjusted ϵ manually throughout training.]

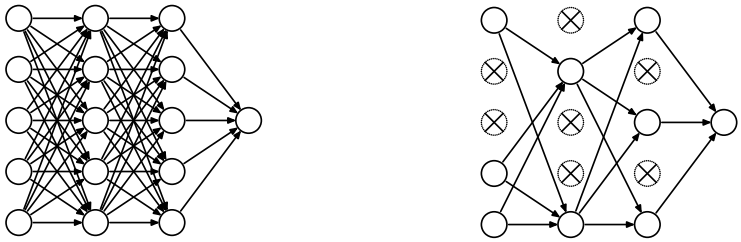
(5) Train for a very long time. [Andrej Karpathy: "I've often seen people tempted to stop the model training when the validation loss seems to be leveling off. In my experience networks keep training for an unintuitively long time. One time I accidentally left a model training during the winter break and when I got back in January it was SOTA ("state of the art").[9]]

[9] http://karpathy.github.io/2019/04/25/recipe/

(6) Ensemble of neural nets. Random initial weights + SGD + (optionally) bagging.
[Ensembles work well for neural nets, reportedly improving test accuracy by 2–3%. Random initial weights and random minibatches ensure that each neural net finds a different local minimum. If you have finite training data, validate to see if bagging helps or not. Obviously, ensembles of neural nets are very slow.]

For speed, sometimes the ensembles share the same early layers.
[Then only the last layers of each neural network are trained separately.]

(7) Dropout emulates an ensemble in one network.



dropout1.pdf, dropout2.pdf

During training, temporarily disable a random subset of the units, along with all edges in and out.
- No forward signal, no weight updates for edges in or out of disabled unit.
- Disable each hidden unit with probability (typically) 0.5.
- Disable each input unit with probability (typically) 0.2.
- Disable a **different** random subset for each SGD minibatch.

After training, before testing: enable all units. If units in a layer were disabled with probability p, multiply all edge weights out of that layer by p.
[When we disabled units, the edge weights had to grow large to make up for their disabled neighbors. At test time, all units and edges are enabled, so we have to reduce the weights to compensate.]

[Dropout gives an effect similar to averaging over multiple neural networks, but it's faster to train. Dropout usually gives better generalization than ℓ_2 regularization. Geoff Hinton and his co-authors give an example where they trained a network to classify MNIST digits. Their network without dropout had a 1.6% test error; it improved to 1.3% with dropout on the hidden units only; and further improved to 1.1% with dropout on the input units too.]

[Recall Karl Lashley's rat experiments, where he tried to make rats forget how to run a maze by introducing lesions in their cerebral cortexes, and it didn't work. He concluded that the knowledge is distributed throughout their brains, not localized in one place. Dropout is a way to force neural networks to distribute knowledge throughout the weights.]

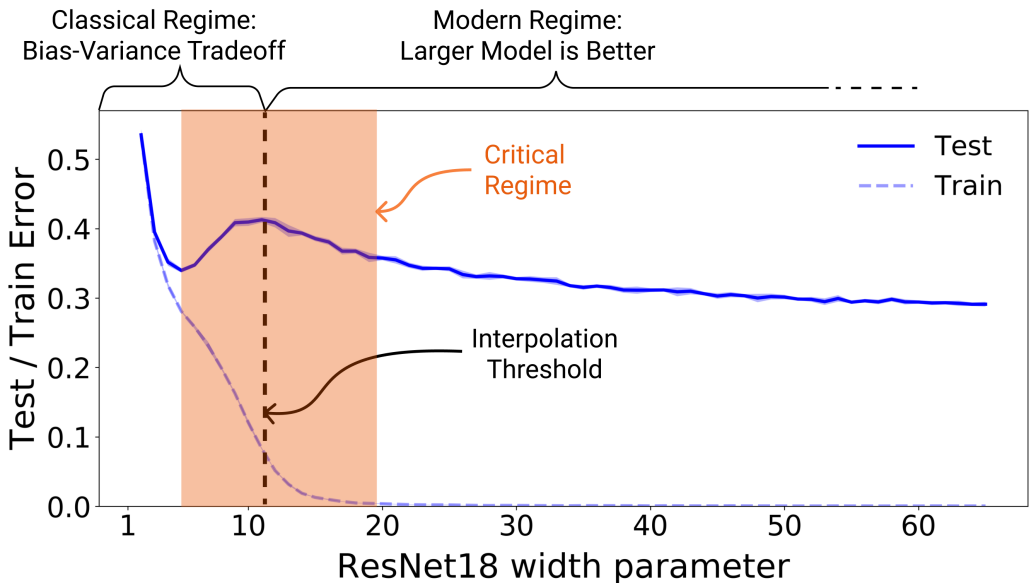
Double Descent

[Early neural network researchers sometimes struggled with their networks falling into bad local minima and failing to achieve low training errors. But with experience and greater computational power, we've discovered that these problems can usually be solved simply by adding more units to every hidden layer. We call this "making the network wider." Sometimes you also have to add more layers. But if your layers are wide enough, and there are enough of them, a well-designed neural network can typically output exactly the correct label for every training point, which implies that you're at a *global* minimum of the cost function.]

Hidden layers are wide enough + numerous enough ⇒ network output can interpolate the label ($\hat{y} = y$) for every training pt ⇒ find global minimum of cost fn.

[I have pointed out that if you use sigmoid or softmax output units, you can't set the labels to exactly 1 or 0 and achieve interpolation, as sigmoid and softmax outputs are strictly between 0 and 1; but with labels like 0.1 and 0.9, interpolating the labels is a realistic goal! And the linear output units used for regression can interpolate arbitrary numbers. Bottom line: if you fall into a bad local minimum, your network is too small.]

[One reason it took so long to make this discovery is that researchers believed that having too many weights in a neural network would cause overfitting. It turns out that's only half true. Empirically, we sometimes observe a phenomenon called "double descent," illustrated below.]



doubledescent.pdf (Nakkiran et al., "Deep Double Descent") [A classic double descent curve (solid blue) for test error. The horizontal axis indicates the number of units in each hidden layer of a residual neural network used for image recognition, and the vertical axis measures the the test error (solid curve) and training error (dashed curve).]

[Consider the solid blue curve, showing the test error as the width of a network increases. The horizontal axis is the number of units per hidden layer. As that number increases, at left the test error exhibits the classic U-shaped bias-variance "tradeoff." But when we pass the point where the network is interpolating the labels and continue to add more weights, we sometimes see a second "descent," where the test error starts to decrease again and ultimately gets even lower than before! The peak in the middle of the curve tends to be larger when there is more noise in the labels. Observe that the test error continues to fall even after the training error is zero. The takeaway is, "**bigger models are often better.**"]

[The currently accepted explanation for double descent, per Nakkiran et al., is that "at the interpolation threshold . . . the model is just barely able to fit the training data; forcing it to fit even slightly-noisy or mis-specified labels will destroy its global structure, and result in high test error. However for over-parameterized models, there are many interpolating models that fit the training set, and SGD is able to find one that 'absorbs' the noise while still performing well on the distribution."]

[Double descent has also been observed in decision trees and even in linear regression where we add random features to the training points (thereby adding more weights to the linear regression model).]

23 Residual Networks; Batch Normalization; AdamW

TRAINING DEEP NETWORKS

Most influential ideas: ResNets, batch normalization, layer normalization.
[These ideas enable deep networks to train. They reduce the likelihood of encountering the vanishing gradient or exploding gradient problems, but don't quite eliminate them.]

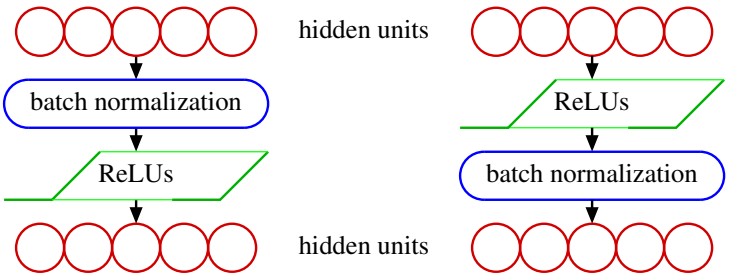
Batch Normalization

[Batch normalization has played a huge role in making it easier to train very deep neural networks since its introduction in 2015, and it's still a mainstay today. It seems to make the cost function uglier, though; when you can train a network without it, it might generalize better.]

Recall batch normalization from Homework 6: For a vector a of activations,

- a batch-norm layer learns parameters β_i and γ_i for each activation a_i;
- calculate the sample mean μ of a and the sample variances σ_i^2 of a_i over a minibatch;
- for some small ϵ, the layer outputs vector z with $z_i = \beta_i + \gamma_i \dfrac{a_i - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}}$.

[We didn't say much in the homework about where batch normalization layers are used. There is some disagreement over whether it's best to place them before or after a nonlinear activation function. Both ways are commonly used. The only clear rules are to never use batch normalization for outputs, and never use ReLUs for inputs.]



batchrelu.pdf [The original authors proposed the left version.]

[Batch normalization is often applied to image data in convolutional neural networks, but not quite like this. We do not normalize each pixel separately. Remember that in a CNN, we rely heavily on relationships between adjacent pixels. Normalizing each pixel separately could introduce spurious edges and ruin the network's ability to recognize images. But we can normalize an entire image as a whole, and we can normalize each channel separately.]

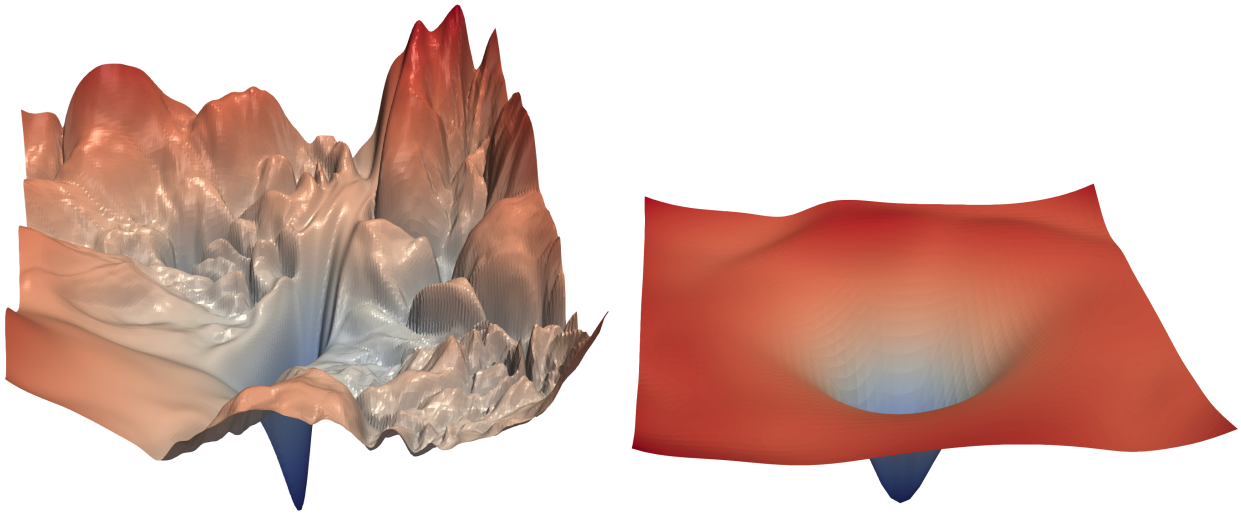
Batch-norm for images: compute mean & variance over all images in minibatch AND all pixels in each image. One β and one γ **per channel**.

Layer normalization: compute mean & variance over all hidden units (all pixels and all channels), but not (necessarily) over images. One β and one γ **per image**.

[Layer normalization ensures that for any one image and any one layer of hidden units, the hidden unit vector will lie on a sphere with center β and radius γ. Layer normalization is easier to parallelize than batch normalization, and it is particularly useful in recurrent neural networks.]

Residual Neural Networks (ResNets)

[Look at this famous figure depicting two-dimensional cross sections through the cost function of deep convolutional neural networks. At right is the cost for a residual neural network. At left is the cost if we remove the "residual connections" that characterize residual networks. You can guess which one is easier to optimize.]



(a) without skip connections (b) with skip connections

lossskip.pdf, (Hao Li et al., 2018)

Idea: design a network with layers that can easily represent the identity fn, e.g., when all weights are zero.
[There are two observations that help to motivate this idea.]

Motivation 1: Networks with nonlinear activations have great difficulty representing the identity. Let's fix that.
[If a hidden layer has negative unit values, a subsequent hidden layer with ReLUs cannot replicate those values. You might be able to replicate them at a linear output layer if you're very clever. Instead, let's redesign our networks to make it easy.]

Motivation 2: Consider a linear neural network $\hat{y} = W_L W_{L-1} \cdots W_1 x$ with square matrices W_i.

Given an $n \times d$ design matrix X and an $n \times d$ label matrix Y, solve this linear regression problem by gradient descent [batch or stochastic].

Find matrices that minimize $J(W_L, W_{L-1}, \ldots, W_1) = \|W_L W_{L-1} \cdots W_1 X^\top - Y^\top\|_F^2$.

The cost fn is very smooth around $(I, I, \ldots, I)$, but much more complicated in regions close to $(0, 0, \ldots, 0)$.
[There's a paper showing that near the identity matrices, every critical point of the cost function is a global minimum—much like in the figure at top right. So even a simple linear neural network suffices to show some of the behavior in the figure.]

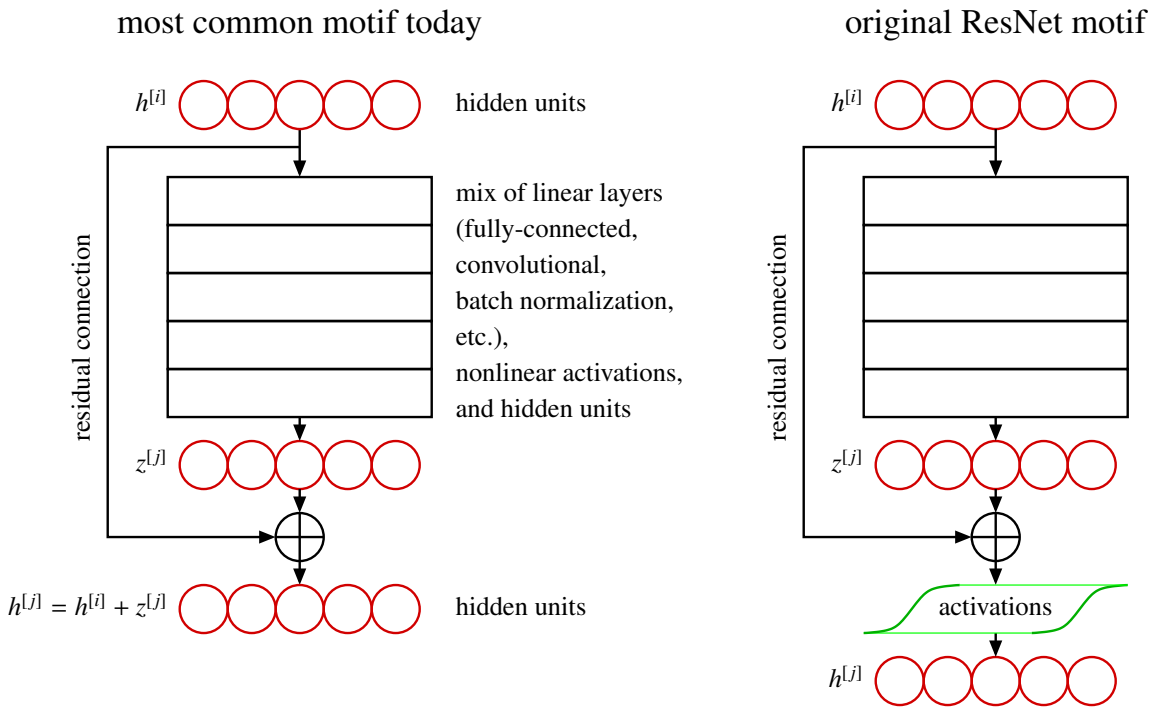
Any network satisfying $W_L W_{L-1} \cdots W_1 = Y^\top (X^\top)^+$ is a solution.
If L is sufficiently large, there is a solution or approximate solution where each W_i is "close" to I.
[If you multiply together enough matrices that are close to the identity matrix, you can obtain any square, invertible matrix, and you can get very close to any square matrix. So for sufficiently many layers, there are solutions in the "nice" region of the cost function. SGD starting from the identity network will find them.]

[Of course, you would never do linear regression this way. But when we add nonlinear activation functions such as ReLUs between the matrices, the phenomenon persists that the cost function is ugly near the origin and can be relatively nice where the network is computing an function near the identity function at each layer.]

Takeaways:

- Initializing near I (plus small random weights) is better than initializing near zero.
- More layers makes it more likely there's a solution in a "nice" part of the cost fn.

ResNets use residual connections aka skip connections to add hidden layer values to subsequent layers. Networks are constructed by repeating one of these motifs.



motifs.pdf

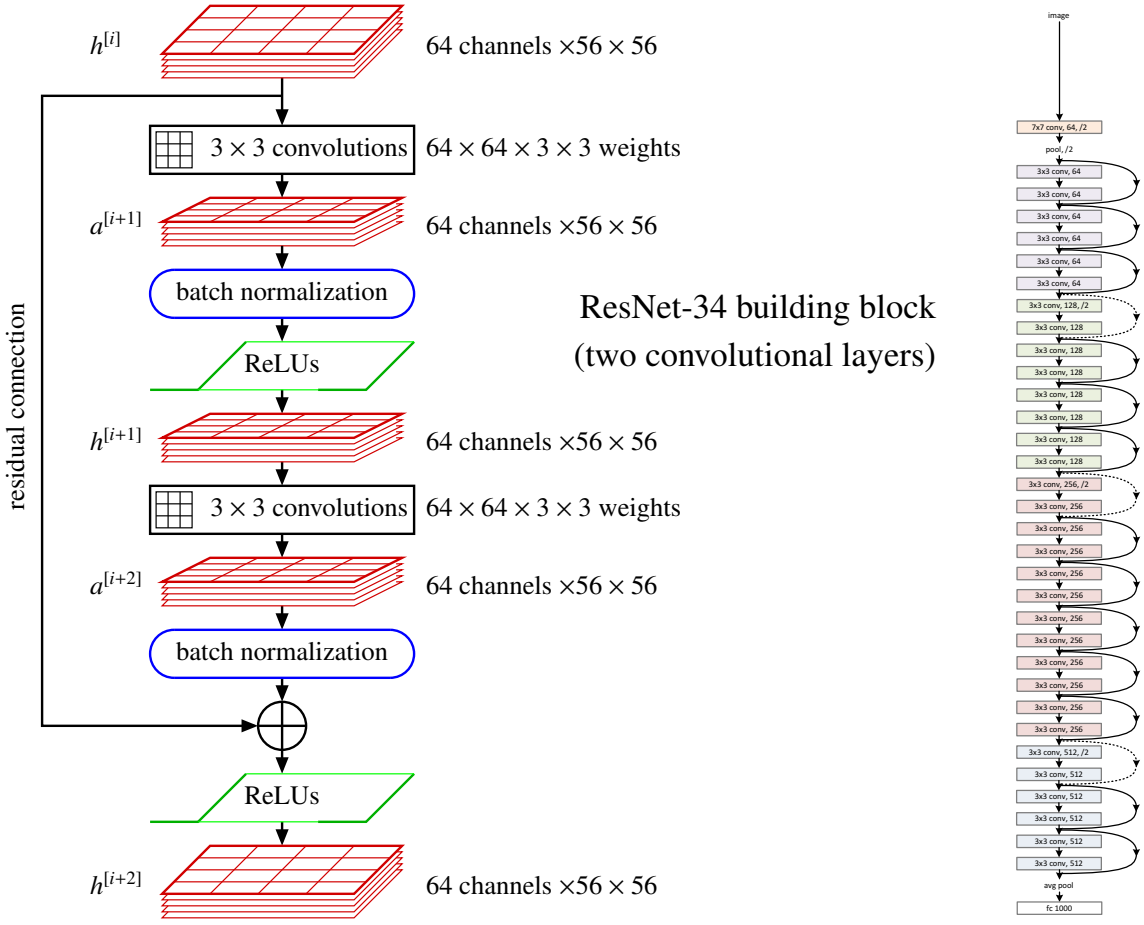
[The motif on the right was used by the original ResNet paper, with ReLU activations after the residual connection. But the motif on the left seems to be more popular now.]

If all weights are zero, left motif sets $h^{[j]} = h^{[i]}$ by default.
Right motif with ReLUs copies all positive units but zeros out negative ones.

[A big advantage of the motif on the left is that if it is advantageous to send some hidden unit values from an early layer to a later layer unchanged, the network has the opportunity to learn to do that. The motif on the right can do that with positive hidden unit values if it uses ReLU activations.]

If the "ideal" mapping from $h^{[i]}$ to $h^{[j]}$ is expressed by a function f, the left motif is trying to learn $f(h) - h$ (which we hope is small).

[The first ResNets were CNNs for image classification. The authors won first place in the 2015 ImageNet Large Scale Visual Recognition Challenge with an ensemble of six ResNets, two of which had 152 layers. This was the biggest advance in neural network vision performance since the AlexNet paper that changed modern computer vision in 2012. Here is the building block for one of their smaller ResNets, a 34-layer model. At right is the authors' schematic of the whole ResNet-34 network with normalizations omitted.]

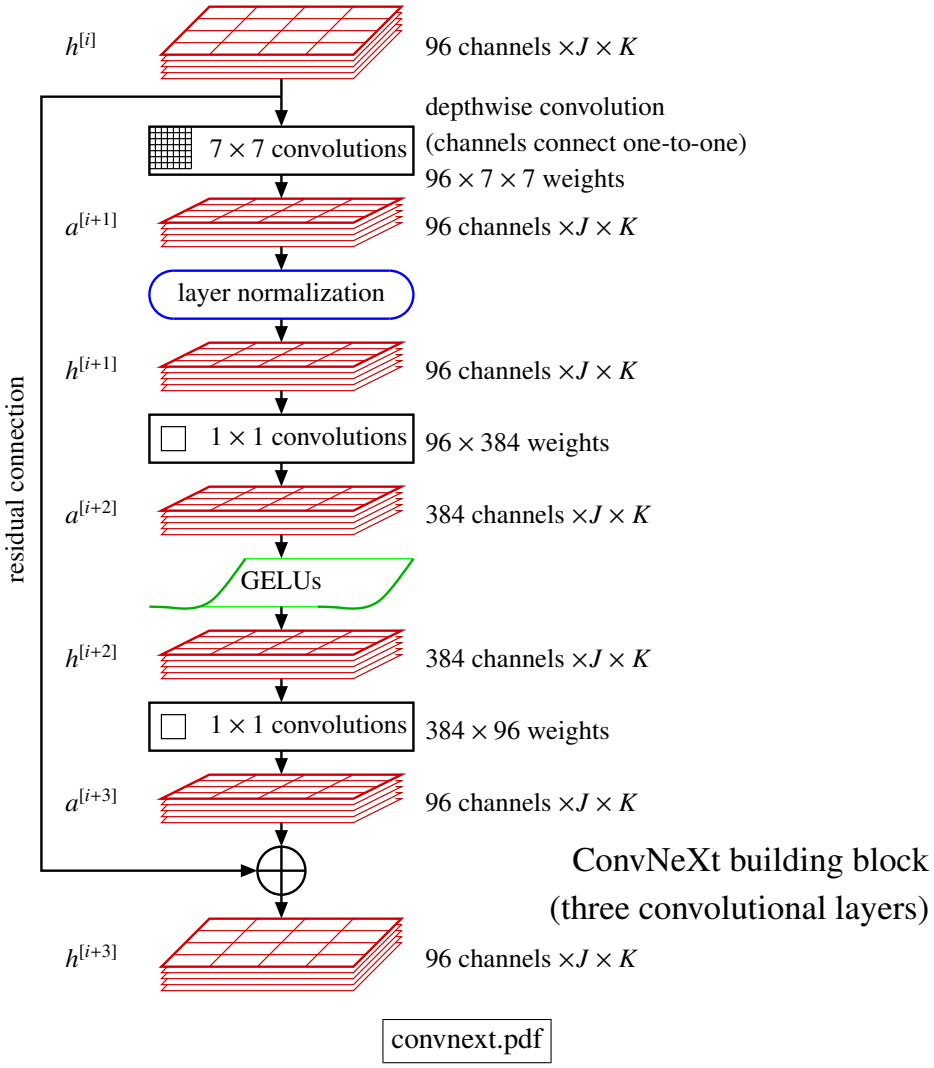


resnet34.pdf, resnet34he.pdf

[Observe that there is a layer of ReLU activations in the middle of the motif, with linear convolutions before and after it. Each convolution is followed by a batch normalization layer. The biggest ResNet in the competition-winning ensemble, with 152 layers, uses a motif with three convolution layers, three batch normalization layers, and three ReLU layers—two within the residual connection and one after.]

[Below is an example of the building block for a more modern convolutional ResNet, called ConvNeXt, produced by a collaboration between Facebook and Berkeley. Unlike the original ResNet, it does not place an activation function after the residual connection.]

[The authors found that one layer normalization step per three convolutional layers suffices, whereas the original ResNets used a batch normalization step after every convolutional layer. Instead of a ReLU, they use a GELU, which stands for Gaussian Error Linear Unit. It's similar in shape to a ReLU but it's smooth, with no discontinuity. It appears to give better test accuracy than ReLUs in some circumstances, though not all. It is popular in transformers for speech generation.]



[The authors found that they get better test accuracy if they use 7×7 convolution filters, not just smaller ones. But these filters use depthwise convolution, which means that each output channel receives input from only one output channel. By contrast, traditional convolutional layers connect every output channel to every input channel. The advantage of depthwise convolution is a big savings in weights and time.]

[Another interesting design choice is the use of an inverted bottleneck, where they temporarily increase the number of channels from 96 to 384 and then decrease it again to 96. This creates very wide layers that lead to better generalization to test data. To prevent the number of weights from exploding, they use the odd idea of a 1×1 convolution going into and out of the 384 channels. This simply means that there is an all-to-all connection from 96 channels to 384 channels, but there are no connections between adjacent "pixels" in the activation maps. Note that the 1×1 convolutions are not depthwise convolutions! A 1×1 depthwise convolution would not permit any mixing of information at all.]

[The combination of depthwise convolutions and 1×1 convolutions causes channel mixing to be separated from spatial mixing. Both kinds of mixing take place, but they take place in different convolutional layers. This permits us to greatly reduce the number of weights while still allowing information to eventually flow everywhere.]

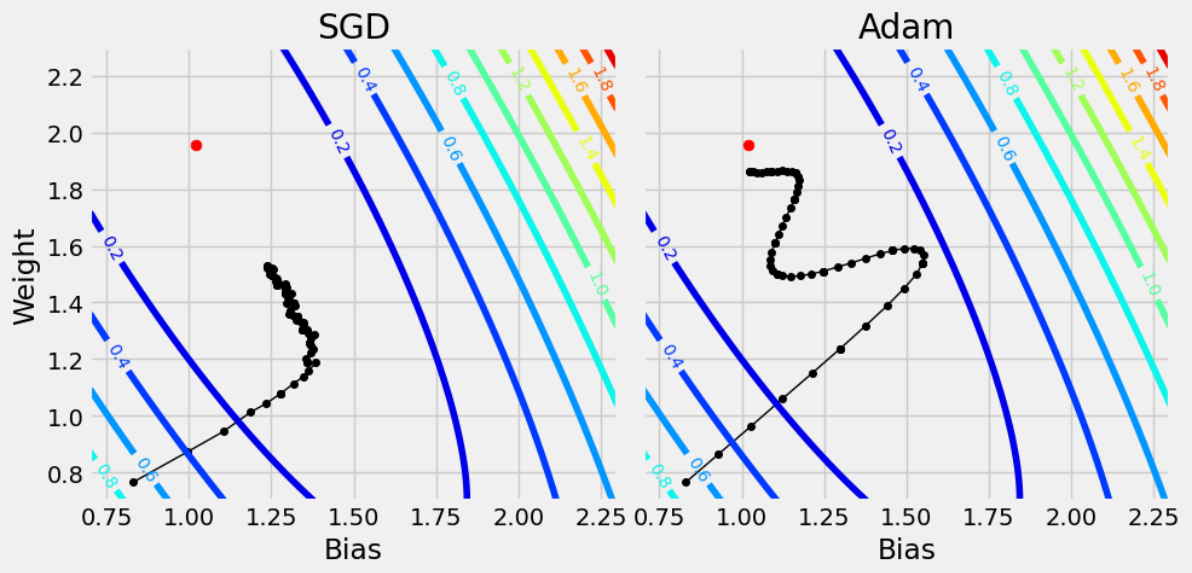
AdamW

"Adaptive moment estimation with weight decay."
An optimization method faster than SGD. Warning: may reduce test accuracy.
Let J be losses summed over a minibatch. Let w_i be a weight. Intuition:

- Sign of $\frac{\partial J}{\partial w_i}$ gives more useful information than its relative magnitude.
- We change w_i slowly if $\frac{\partial J}{\partial w_i}$ changes sign frequently; otherwise stay fast.
  [Roughly speaking, each weight has its own learning rate.]
- Keep exponential moving averages m_i of $\frac{\partial J}{\partial w_i}$ [first moment] and r_i of $\left(\frac{\partial J}{\partial w_i}\right)^2$ [second raw moment].
  Each step has $\Delta w_i \propto -m_i/\sqrt{r_i}$. Typically $m_i/\sqrt{r_i} \approx \pm 1$, but smaller if sign $(\partial J/\partial w_i)$ changes often.

$m \leftarrow 0$; $r \leftarrow 0$; $t = 0$
repeat
  $t \leftarrow t + 1$
  $g \leftarrow \nabla J(w)$
  $m \leftarrow \beta m + (1-\beta)\, g$
  $r \leftarrow \tau r + (1-\tau) g \odot g$ [elementwise multiplication; square each component of g]
  $\hat{m} \leftarrow m/(1-\beta^t)$ [correcting bias, as m was initialized to zero]
  $\hat{r} \leftarrow r/(1-\tau^t)$ [correcting bias]
  $\forall i, w_i \leftarrow w_i - \epsilon\left(\frac{\alpha \hat{m}_i}{\sqrt{\hat{r}_i} + \delta} + \lambda w_i\right)$

Typical parameters: $\alpha = 0.001, \beta = 0.9, \tau = 0.9999, \delta = 10^{-8}$.
Weight decay term λ regularizes; set by validation. If $\lambda = 0$, it's just called Adam.



sgdadamgodoy.png (Daniel Godoy) [Left: 50 steps of SGD (with 16-point minibatches) don't get very close to the minimum (red). Right: 50 steps with Adam.]

24 Boosting; Nearest Neighbor Classification

ADABOOST (Yoav Freund and Robert Schapire, 1997)

AdaBoost ("adaptive boosting") is an ensemble method for classification (or regression) that
- **reduces bias** [compare with random forests and other ensembles, which reduce variance];
- trains learners on weighted sample points [like bagging];
- uses different weights for each learner;
- increases weights of misclassified training points;
- gives bigger votes to more accurate learners.

Input: $n \times d$ design matrix X, vector of labels $y \in \mathbb{R}^n$ with $y_i = \pm 1$.

Ideas:
- Train T classifiers $G_1, \ldots, G_T$. ["T" stands for "trees"]
- Weight for training point X_i in G_t grows according to how many of $G_1, \ldots, G_{t-1}$ misclassified it.
  [Moreover, if X_i is misclassified by very accurate learners, its weight grows even more.]
  [And, the weight shrinks every time X_i is correctly classified by a learner.]
- Train G_t to try harder to correctly classify training pts with larger weights.
- Metalearner is a linear combination of learners. For test point z, $M(z) = \sum_{t=1}^{T} \beta_t G_t(z)$.
  Each G_t is ± 1, but M is continuous. Return sign of $M(z)$.

[Remember that in the previous lecture on ensemble methods, I talked briefly about how to assign different weights to training points. It varies for different learning algorithms. For example, in regression we usually modify the risk function by multiplying each point's loss function by its weight. In a soft-margin support vector machine, we modify the objective function by multiplying each point's slack by its weight.]
[Boosting works with many learning algorithms, but it was originally developed for decision trees, and boosted decision trees are very popular and successful. To weight points in decision trees, we use a weighted entropy where instead of computing the proportion of points in each class, we compute the proportion of weight in each class.]

In iteration T, what classifier G_T and coefficient β_T should we choose? Pick a loss fn L(prediction, label).

Find G_T & β_T that minimize

$$\text{Risk} = \frac{1}{n} \sum_{i=1}^{n} L(M(X_i), y_i), \qquad M(X_i) = \sum_{t=1}^{T} \beta_t G_t(X_i).$$

AdaBoost metalearner uses exponential loss function

$$L(\hat{\lambda}, \lambda) = e^{-\hat{\lambda}\lambda} = \begin{cases} e^{-\hat{\lambda}} & \lambda = +1 \\ e^{\hat{\lambda}} & \lambda = -1 \end{cases}$$

Important: label λ is binary, G_t is binary, but $\hat{\lambda} = M(X_i)$ is continuous!
[This loss function is for the metalearner only. We will discover later that the ideal cost function for each individual learner G_t is just the total weight of the misclassified points. In practice, our individual learners are often classification algorithms like decision trees that don't explicitly try to minimize any loss function at all. Even when the practice doesn't match the theory, boosting still usually works quite well.]

[The exponential loss function has the advantage that it pushes hard against badly misclassified points. So it's usually better than the squared error loss function for classification in a metalearner. It's similar to why in neural networks we prefer the cross-entropy loss function over the squared error for sigmoid outputs.]

$$
\begin{aligned}
n \cdot \text{Risk} &= \sum_{i=1}^{n} L(M(X_i), y_i) = \sum_{i=1}^{n} e^{-y_i M(X_i)} \\
&= \sum_{i=1}^{n} \exp\left(-y_i \sum_{t=1}^{T} \beta_t G_t(X_i)\right) = \sum_{i=1}^{n} \prod_{t=1}^{T} e^{-\beta_t y_i G_t(X_i)} \qquad \Leftarrow \begin{array}{l} y_i G_t(X_i) = \pm 1 \\ \text{if } -1,\ G_t \text{ misclassifies } X_i \end{array} \\
&= \sum_{i=1}^{n} w_i^{(T)} e^{-\beta_T y_i G_T(X_i)}, \quad \text{where } w_i^{(T)} = \prod_{t=1}^{T-1} e^{-\beta_t y_i G_t(X_i)} \\
&= e^{-\beta_T} \sum_{y_i = G_T(X_i)} w_i^{(T)} + e^{\beta_T} \sum_{y_i \neq G_T(X_i)} w_i^{(T)} \qquad \text{[correctly classified and misclassified]} \\
&= e^{-\beta_T} \sum_{i=1}^{n} w_i^{(T)} + (e^{\beta_T} - e^{-\beta_T}) \sum_{y_i \neq G_T(X_i)} w_i^{(T)}.
\end{aligned}
$$

What G_T minimizes the risk? The learner that minimizes the sum of $w_i^{(T)}$ over all misclassified pts X_i!

[This is interesting. By manipulating the formula for the risk, we've discovered what weight we should assign to each training point. To minimize the risk, we should find the classifier that minimizes the sum of the weights $w_i^{(T)}$, as specified above, over the misclassified points. It's a complicated function, but we can compute it. A useful observation is that each learner's weights are related to the previous learner's weights:]

Recursive definition of weights:

$$
w_i^{(T+1)} = w_i^{(T)} e^{-\beta_T y_i G_T(X_i)} = \begin{cases} w_i^{(T)} e^{-\beta_T} & y_i = G_T(X_i), \\ w_i^{(T)} e^{\beta_T} & y_i \neq G_T(X_i). \end{cases}
$$

[This recursive formulation is a nice benefit of choosing the exponential loss function. Notice that a weight shrinks if the point was classified correctly by learner T, and grows if the point was misclassified.]
[Now, you might wonder if we should just pick a learner that classifies all the training points correctly. But that's not always possible. If we're using a linear classifier on data that's not linearly separable, some points must be classified wrongly. Moreover, it's NP-hard to find the optimal linear classifier, so in practice G_T will be an approximate best learner, not the true minimizer of the weighted training error. But that's okay.]
[You might ask, if we use decision trees, can't we get zero training error? Usually we can. But interestingly, boosting is usually used with short, imperfect decision trees instead of tall, pure decision trees, for reasons I'll explain later.]
[Now, let's derive the optimal value of β_T.]

To choose β_T, set $\frac{\mathrm{d}}{\mathrm{d}\beta_T}\text{Risk} = 0$.

$$
\begin{aligned}
0 &= -e^{-\beta_T} \sum_{i=1}^{n} w_i^{(T)} + (e^{\beta_T} + e^{-\beta_T}) \sum_{y_i \neq G_T(X_i)} w_i^{(T)}; \qquad \text{[now divide both sides by the first term]} \\
0 &= -1 + (e^{2\beta_T} + 1)\,\text{err}_T, \quad \text{where } \text{err}_T = \frac{\sum_{y_i \neq G_T(X_i)} w_i^{(T)}}{\sum_{i=1}^{n} w_i^{(T)}}; \qquad \Leftarrow G_T\text{'s weighted error rate} \\
\beta_T &= \frac{1}{2} \ln\left(\frac{1 - \text{err}_T}{\text{err}_T}\right).
\end{aligned}
$$

[So now we have derived the optimal metalearner!]

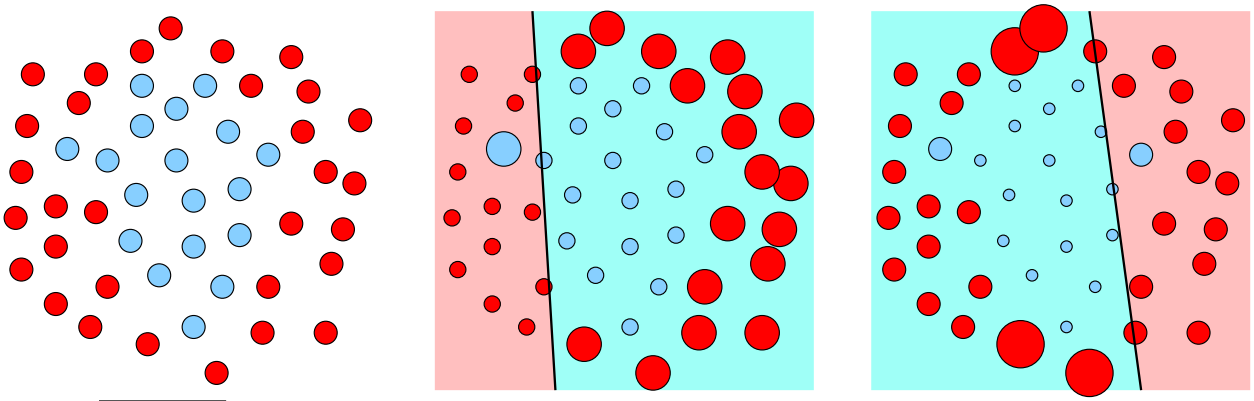
- If $\text{err}_T = 0, \beta_T = \infty$. [So a perfect learner gets an infinite vote.]
- If $\text{err}_T = 1/2, \beta_T = 0$. [So a learner with 50% weighted training error gets no vote at all.]

[More accurate learners get bigger votes in the metalearner. Interestingly, a learner with training error worse than 50% gets a negative vote. A learner with 60% error is just as useful as a learner with 40% error; the metalearner just reverses the signs of its votes. It's so bad, it's good.]

[Now we can state the AdaBoost algorithm.]

AdaBoost alg:

1. Initialize weights $w_i \leftarrow \frac{1}{n}, \forall i \in [1, n]$.
2. for $t \leftarrow 1$ to T
   a. Train G_t with weights w_i.
   b. Compute weighted error rate $\text{err} \leftarrow \dfrac{\sum_{\text{misclassified}} w_i}{\sum_{\text{all}} w_i}$; coefficient $\beta_t \leftarrow \dfrac{1}{2} \ln\left(\dfrac{1-\text{err}}{\text{err}}\right)$.
   c. Reweight pts: $w_i \leftarrow w_i \cdot \begin{cases} e^{\beta_t}, & G_t \text{ misclassifies } X_i \\ e^{-\beta_t}, & \text{otherwise} \end{cases} = w_i \cdot \begin{cases} \sqrt{\frac{1-\text{err}}{\text{err}}}, \\ \sqrt{\frac{\text{err}}{1-\text{err}}}. \end{cases}$
3. return metalearner $h(z) = \text{sign}\left(\sum_{t=1}^{T} \beta_t G_t(z)\right)$.



boost.pdf [At left, all the training points have equal weight. After choosing a first linear classifier, we increase the weights of the misclassified points and decrease the weights of the correctly classified points (center). We train a second classifier with these weighted points, then again adjust the weights of the points according to whether they are misclassified by the second classifier.]

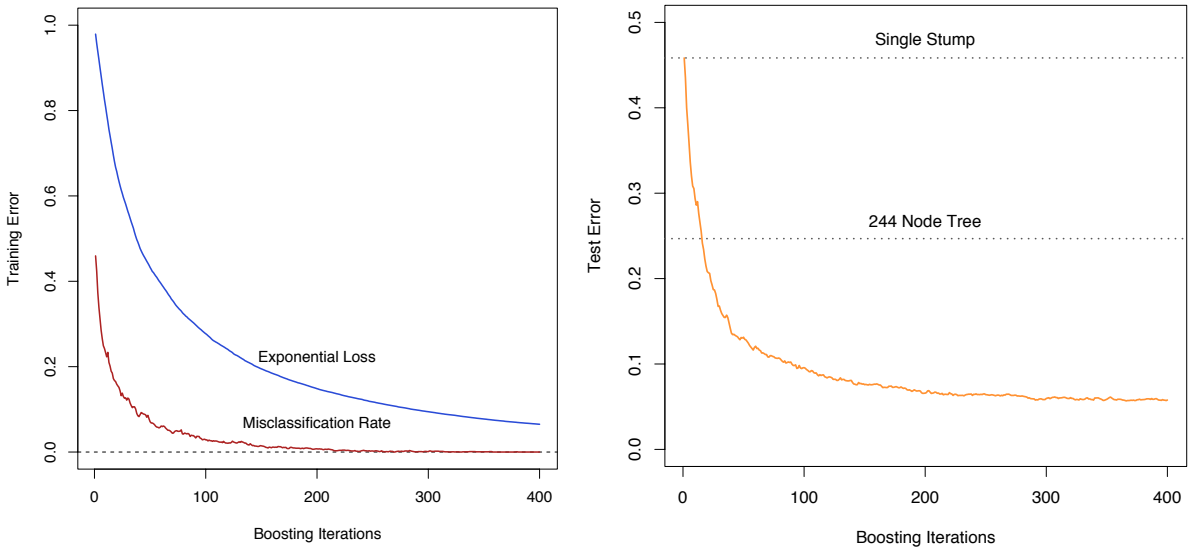
Why boost decision trees? [As opposed to other learning algorithms?] Why **short** trees?

- **Boosting reduces bias reliably, but not always variance.** AdaBoost trees are impure to reduce overfitting. [Recall again that random forests use ensembles to reduce variance, but AdaBoost uses them to reduce bias. The AdaBoost variance is more complicated: it often decreases at first, because successive trees focus on different features, but often it later increases. Sometimes boosting overfits after many iterations, and sometimes it doesn't; it's hard to predict when it will and when it won't.]
- Fast. [We're training many learners, and running many learners at classification time too. Short decision trees that only look at a few features are very fast at both training and testing.]
- No hyperparameter search needed. [Unlike SVMs, neural nets, etc.] [UC Berkeley's Leo Breiman called AdaBoost with decision trees "the best off-the-shelf classifier in the world."]
- Easy to make a tree beat 45% training error [or some other threshold] consistently.

– AdaBoost + short trees is a form of subset selection.
[Features that don't improve the metalearner's predictive power enough aren't used at all. This helps reduce overfitting and running time, especially if there are a lot of irrelevant features.]
– Linear decision boundaries don't boost well.
[It takes a lot of boosting to make linear classifiers model really nonlinear decision boundaries well, so SVMs aren't a great choice. Recall from Discussion Section 9 that ensembles of depth-one stumps are an even worse choice, because they can't even do XOR. Methods with nonlinear decision boundaries benefit more from boosting, because they allow boosting to reduce the bias faster much. Even depth-two decision trees boost substantially better than depth-one decision trees.]

More about AdaBoost:

– Posterior prob. can be approximated: $P(Y = 1|x) \approx \dfrac{1}{1 + e^{-2M(x)}}$.
– Exponential loss is vulnerable to outliers; for corrupted data, use other loss.
[Loss functions have been derived for dealing with outliers. Unfortunately, they have more complicated weight computations.]
– If every learner beats error μ for $\mu < 50\%$, metalearner training error will eventually be zero. [You will prove this in Homework 7.]
– [The AdaBoost paper and its authors, Freund and Schapire, won the 2003 Gödel Prize, a prize for outstanding papers in theoretical computer science.]



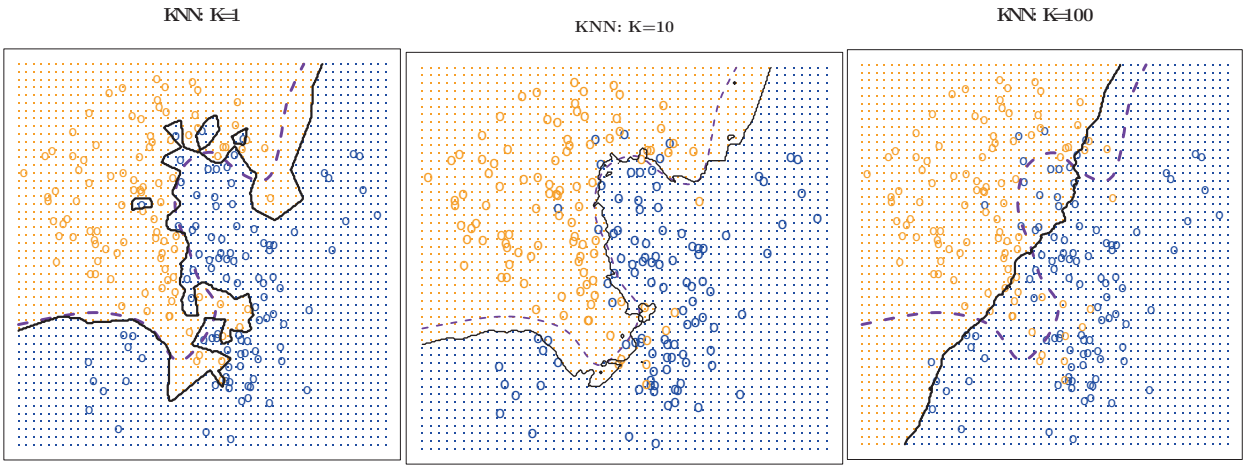
trainboost.pdf, testboost.pdf (ESL, Figures 10.2, 10.3) [Training and testing errors for AdaBoost with stumps, depth-one decision trees that make only one decision each. At left, observe that the training error eventually drops to zero, and even after that the average loss (which is continuous, not binary) continues to decay exponentially. At right, the test error drops to 5.8% after 400 iterations, even though each learner has an error rate of about 46%. AdaBoost with more than 25 stumps outperforms a single 244-node decision tree. In this example no overfitting is observed, but there are other datasets for which overfitting is a problem.]

NEAREST NEIGHBOR CLASSIFICATION

[I saved the simplest classifier for the end of the semester.]

Idea: Given query point q, find the k training pts nearest q.
Distance metric of your choice.
Regression: Return average label of the k pts.
Classification: Return class with the most votes from the k pts OR
return histogram of class probabilities.

[The histogram of class probabilities tries to estimate the posterior probabilities of the classes. Obviously, the histogram 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. The histogram 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 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 (purple). Generally, the ideal k depends on how dense your data is. As your data gets denser, the best 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 $< 2B - B^2$ where B = Bayes risk.
if only 2 classes, $\leq 2B - 2B^2$

[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. Here, we are using the 0-1 loss to define the Bayes risk; so the Bayes risk is the smallest possible error rate over that 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 Bayes optimal.]

25 Nearest Neighbor Algorithms: Voronoi Diagrams and k-d Trees

NEAREST NEIGHBOR ALGORITHMS

Exhaustive k-NN Alg.

Given query point q:
- Scan through all n training pts, computing distances to q.
- Maintain a max-heap with the k shortest distances seen so far.
  [Whenever you encounter a training point closer to q 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 5, but if $k = 99$ a heap will substantially speed up keeping track of the 99th-shortest distance.]

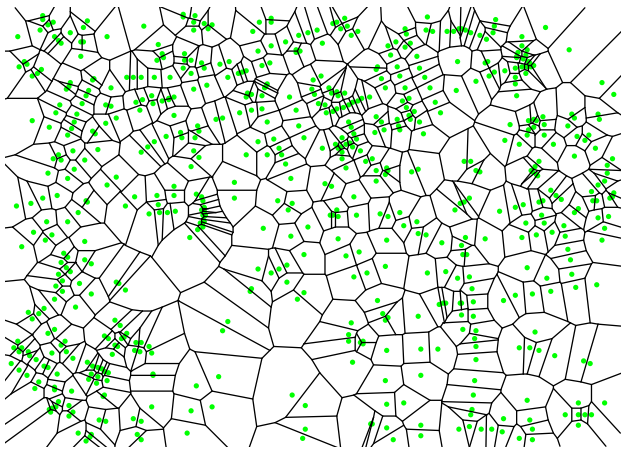
Time to train classifier: 0 [This is the only $O(0)$-time algorithm we'll learn this semester.]
Query time: $O(nd + n \log k)$
expected $O(nd + k \log n \log k)$ if random pt order
[It's a cute theoretical observation that you can slightly improve the expected running time by randomizing the point order so that only expected $O(k \log n)$ heap operations occur. But in practice I can't recommend it; you'll probably lose more from cache misses than you'll gain from fewer heap operations.]

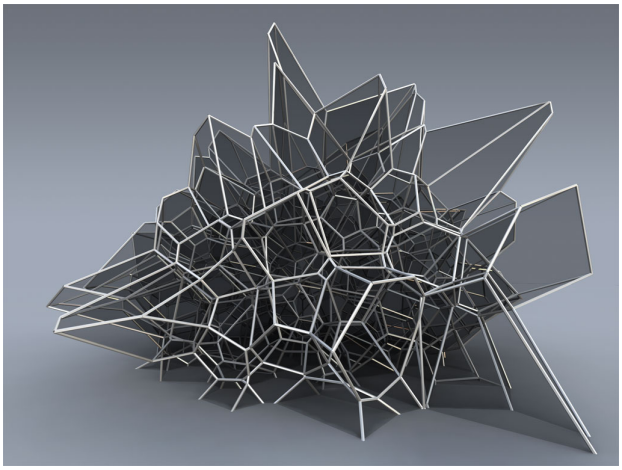
Can we preprocess training pts to obtain sublinear query time?

2–5 dimensions: Voronoi diagrams
Medium dim (up to $\sim$ 30): k-d trees
Large dim: exhaustive k-NN, but can use PCA or random projection
locality sensitive hashing [still researchy, not widely adopted]

Voronoi Diagrams

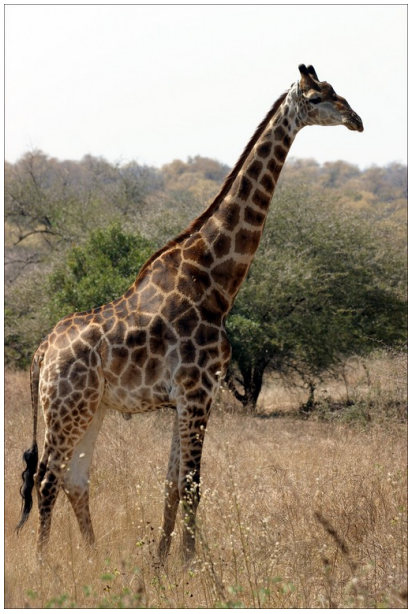
Let X be a point set. The Voronoi cell of $w \in X$ is
Vor $w = \{p \in \mathbb{R}^d : \|p - w\| \leq \|p - v\| \quad \forall v \in X\}$
[A Voronoi cell is always a convex polyhedron or polytope.]
The Voronoi diagram of X is the set of X's Voronoi cells.

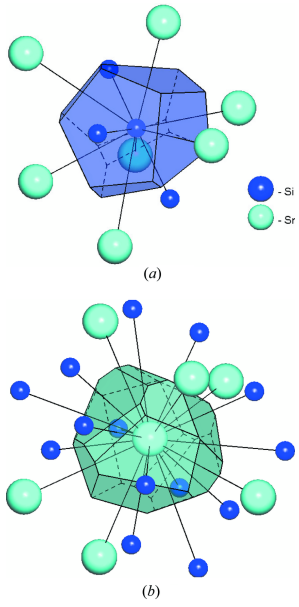


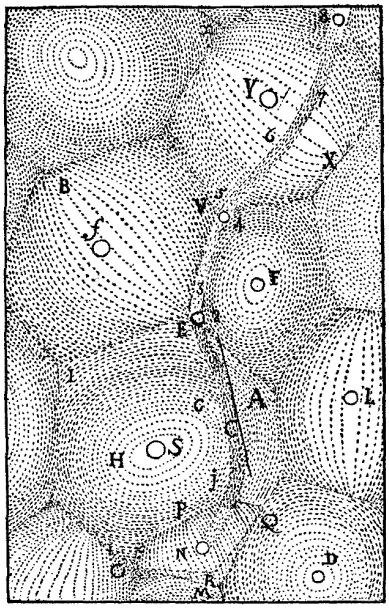


voro.pdf, vormcdonalds.jpg, voronoiGregorEichinger.jpg, saltflat3.jpg

[Voronoi diagrams sometimes arise in nature (salt flats, giraffe, crystallography).]







giraffe-1.jpg, srsi2.png (Vladislav Blatov), vortex.pdf (René Descartes)

[Perhaps the first frequent users of Voronoi cells were crystallographers, who call them Voronoi–Dirichlet polyhedra. Above we see how the polyhedra can clarify the crystal structure of the low-temperature phase of strontium silicide, α-$SrSi_2$.]

[Believe it or not, the first published Voronoi diagram dates back to 1644, in the book "Principia Philosophiae" by the 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 asymptotic 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 may grow exponentially with d.]

Point location: Given query point $q \in \mathbb{R}^d$, find the point $w \in X$ for which $q \in \text{Vor } w$.
[We need a second data structure that can perform this search on a Voronoi diagram efficiently.]

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 often logarithmic in 3–5 dimensions.]

1-NN only! [A standard Voronoi diagram supports only 1-nearest neighbor queries. If you want the k nearest neighbors, there is something called an order-k Voronoi diagram that has a cell for each possible k nearest neighbors. But nobody uses those, for two reasons. First, the size of an order-k Voronoi diagram is $\Theta(k^2 n)$ in 2D, and worse in higher dimensions. Second, there's no reliable software available to compute one.]

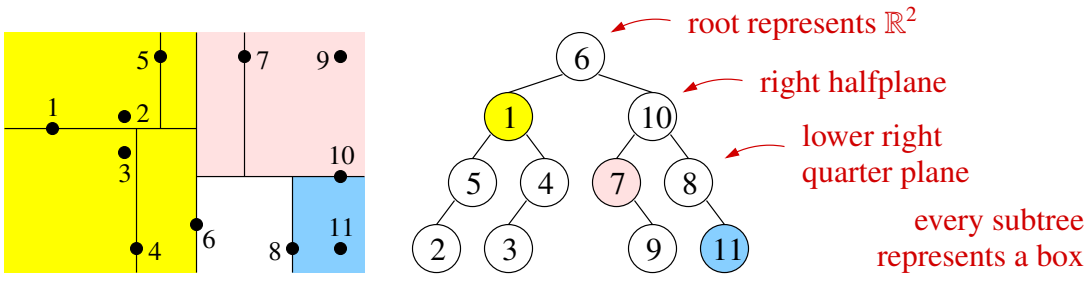
[There are also Voronoi diagrams for other distance metrics, like the ℓ_1 and ℓ_∞ norms.]

[Voronoi diagrams are good for 1-nearest neighbor queries in two dimensions, and maybe up to 5 dimensions, and they're a great concept for understanding the problem of nearest neighbor search. But k-d trees are much simpler, and probably faster in 6 or more dimensions.]

k-d Trees

"Decision trees" for NN search. [Just like in a decision tree, each treenode in a k-d tree represents a rectangular box in feature space, and we split a box by choosing a splitting feature and a splitting value. But we use different criteria for choosing splits.] Differences:

- Choose splitting feature w/greatest width: feature i in $\max_{i,j,k}(X_{ji} - X_{ki})$.
  [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.]
  Cheap alternative: rotate through the features. [At depth 1 we split on the first feature, at depth 2 we split on the second feature, and so on. This builds the tree faster, by a factor of $O(d)$.]
- Choose splitting value: median point for feature i; OR midpoint $\frac{X_{ji}+X_{ki}}{2}$.
  Median guarantees $\lfloor \log_2 n \rfloor$ tree depth; $O(nd \log n)$ tree-building time.
  [...or just $O(n \log n)$ time if you rotate through the features. An alternative to the median is splitting at the box center, which improves the aspect ratios of the boxes, but it could unbalance your tree. A compromise strategy is to alternate between medians at odd depths and centers at even depths, which also guarantees an $O(\log n)$ depth.]
- Each internal node stores a training point. [...that lies in the node's box. Usually the splitting point.]
  [Some k-d tree implementations have points only at the leaves, but it's better to have points in internal nodes too, so when we search the tree, we often stop searching earlier.]



[Draw this by hand. kdtreestructure.pdf]

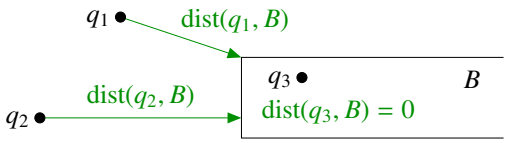
[Just like in a decision tree, each subtree represents an axis-aligned box in feature space. All the training points stored in a subtree are in that box.]

[Once the tree is built, the classification algorithm is very different from decision trees. Most importantly, you usually have to visit multiple leaves of the tree to find the nearest neighbor. To save time, we sometimes use an approximate nearest neighbor algorithm, instead of demanding the exact nearest neighbor.]

After tree is built (training), classify test pts:

Goal: given query pt q, find a training pt w such that $\|q - w\| \leq (1 + \epsilon)\,\|q - u\|$,
where $\|\cdot\|$ is ℓ_p norm for some $p \in [1, \infty]$ [k-d trees are not limited to the Euclidean (ℓ_2) norm.]
& u is the nearest training pt in that norm.
$\epsilon = 0 \quad \Rightarrow \quad$ exact NN; $\quad \epsilon > 0 \quad \Rightarrow \quad$ approximate NN.

Each subtree represents a box $B = [s_1, t_1] \times [s_2, t_2] \times \cdots \times [s_d, t_d]$. [An s_i can be $-\infty$, and a t_i can be ∞.]
[Think of s as the lower left corner of the box, and t as the opposite corner.]
Think of B as an infinite point set.
The distance from q to B is $\text{dist}(q, B) = \min_{z \in B} \|q - z\|$. [This norm is the same norm we seek neighbors in.]

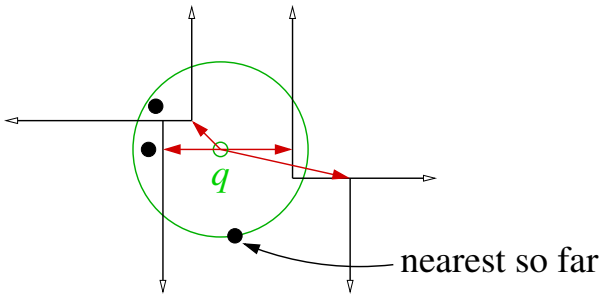


[Draw this by hand. dist.pdf] [Point-to-box distances.]

The minimizer's components are $z_i = \begin{cases} s_i, & q_i < s_i, \\ q_i, & q_i \in [s_i, t_i], \\ t_i, & q_i > t_i. \end{cases}$

Query alg. maintains:

- Nearest neighbor found so far (or k nearest). goes down ↓
- Binary min-heap of unexplored boxes/subtrees, keyed by distance from q. goes up ↑



[Draw this by hand. query.pdf] [A query in progress.]

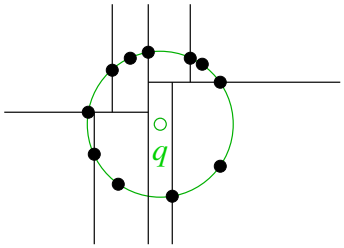
[We search the boxes nearest q first, hoping that we will never need to search most of the boxes or their associated subtrees. The binary heap makes it fast to find the box nearest q, because each box in the heap has a numerical key, the distance from q to the box. The search stops when the distance from q to the kth-nearest neighbor found so far $\leq$ the distance from q to the nearest unexplored box (times $1 + \epsilon$). For example, in the figure above, the query will never visit the box at far lower right, because it doesn't intersect the circle. That's how we avoid searching most of the tree—when we're lucky.]

Alg. for 1-NN query. Interpret each B as both a box and a treenode.

$Q \leftarrow$ min-heap containing root node with key zero
$r \leftarrow \infty$
while Q not empty and $(1+\epsilon) \cdot \text{minkey}(Q) < r$
    $B \leftarrow \text{removemin}(Q)$
    $v \leftarrow B$'s training point
    if $\|q - v\| < r$ then { $w \leftarrow v$; $r \leftarrow \|q - v\|$ }
    $B', B'' \leftarrow$ child boxes of B
    if $(1+\epsilon) \cdot \text{dist}(q, B') < r$ then $\text{insert}(Q, B', \text{dist}(q, B'))$ [The key for B' is $\text{dist}(q, B')$]
    if $(1+\epsilon) \cdot \text{dist}(q, B'') < r$ then $\text{insert}(Q, B'', \text{dist}(q, B''))$
return w

For k-NN, replace "r" and "w" with a max-heap holding the k nearest neighbors.

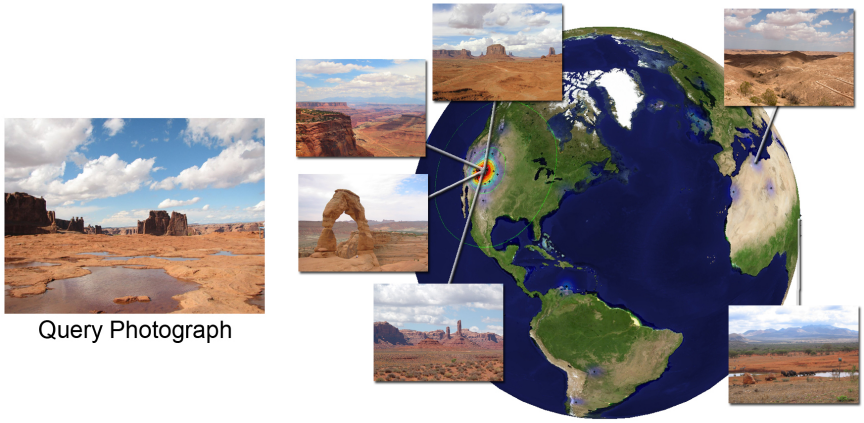
Why ϵ-approximate NN?



[Draw this by hand. kdtreeproblem.pdf] [A worst-case exact NN query.]

[In the worst case, we may have to visit every node in the k-d tree to find the exact nearest neighbor. In that case, the k-d tree is slower than simple exhaustive search. This is an example where an *approximate* nearest neighbor search 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. This is especially true in high dimensions—in a high-dimensional space, the nearest point often isn't much closer than a *lot* of other points.]

[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 research paper that uses a 120-nearest neighbor classifier to solve a problem.]



im2gpspress.pdf

[In 2008, James Hays and our own Prof. Alexei Efros wrote a paper on geolocalization, where the goal is to take a query photograph and determine where on earth the photo was taken. Their training data was 6 million GPS-tagged photos downloaded from Flickr. The bottom line is that by using 120-nearest neighbors, they came within 64 km of the correct location about 50% of the time. That was good for the time, but I think that in 2025, you could do much better with the data available to us today.]

RELATED CLASSES [if you like machine learning, consider these courses in 2024–25]

CS 180/280A (fall): Computer Vision/Photography
CS 182/282A (fall): Deep Neural Networks
EECS 183 (fall?): Natural Language Processing
CS 185/285 (fall?): Deep Reinforcement Learning
CS 194-196/294-196 (fall): Agentic AI (D. Song)
CS C281A (fall): Statistical Learning Theory [C281A is the most direct continuation of CS 189/289A.]
EECS 127 (both), 227AT (both): Numerical Optimization [a core part of ML]
[It's hard to overemphasize the importance of numerical optimization to machine learning, as well as other CS fields like graphics, theory, and scientific computing.]
EECS 126 (both): Random Processes [Markov chains, expectation maximization, PageRank]
EE C106A/B (fall/spring): Intro to Robotics [dynamics, control, sensing]
Math 110 (both): Linear Algebra [but the real gold is in Math 221]
Math 221 (fall): Matrix Computations [how to solve linear systems, compute SVDs, eigenvectors, etc.]
CS C281B (spring): Learning & Decision Making
CS C267 (spring): Scientific Computing [parallelization, practical matrix algebra, some graph partitioning]
CS C280 (spring): Computer Vision (Efros, Kanazawa)
CS 294-162 (fall): ML Systems (Gonzalez/Stoica/Zaharia)
CS 294-286 (fall): Machine Learning in Social Settings (Chang)
NEU 100A (fall): Cellular and Molecular Neurobiology
VS 265 (?): Neural Computation

A Bonus Lecture: Learning Theory

LEARNING THEORY: WHAT IS GENERALIZATION?

[One thing humans do well is generalize. When you were a young child, you only had to see a few examples of cows before you learned to recognize cows, including cows you had never seen before. You didn't have to see every cow. You didn't even have to see $\log n$ of the cows.]

[Learning theory tries to explain how machine learning algorithms generalize, so they can classify data they've never seen before. It also tries to derive mathematically how much training data we need to generalize well. Learning theory starts with the observation that if we want to generalize, it helps to constrain what hypotheses we allow our learner to consider.]

A range space (aka set system) is a pair (P, H), where
P is set of all possible test/training points (can be infinite)
H is hypothesis class, a set of hypotheses (aka ranges, aka classifiers):
each hypothesis is a subset $h \subseteq P$ that specifies which points h predicts are in class C.
[So each hypothesis h is a 2-class classifier, and H is a set of sets of points.]

Examples:

1. Power set classifier: P is a set of k numbers; H is the power set of P, containing all 2^k subsets of P.
   e.g., $P = \{1, 2\}, H = \{\emptyset, \{1\}, \{2\}, \{1, 2\}\}$
2. Linear classifier: $P = \mathbb{R}^d$; H is the set of all halfspaces; each halfspace has the form $\{x : w \cdot x \geq -\alpha\}$.
   [In this example, both P and H are infinite. In particular, H contains every possible halfspace—that is, every possible linear classifier in d dimensions.]

[The power set classifier sounds very powerful, because it can learn every possible hypothesis. But the reality is that it can't generalize at all. Imagine we have three training points and three test points in a row.]

? C C ? N ?

[The power set classifier can classify these three test points any way you like. Unfortunately, that means it has learned nothing about the test points from the training points. By contrast, the linear classifier can learn only two hypotheses that fit this training data. The leftmost test point must be classified class C, and the rightmost test point must be classified class Not-C. Only the test point in the middle can swing either way. So the linear classifier has a big advantage: it can generalize from a few training points. That's also a big disadvantage if the data isn't close to linearly separable, but that's another story.]

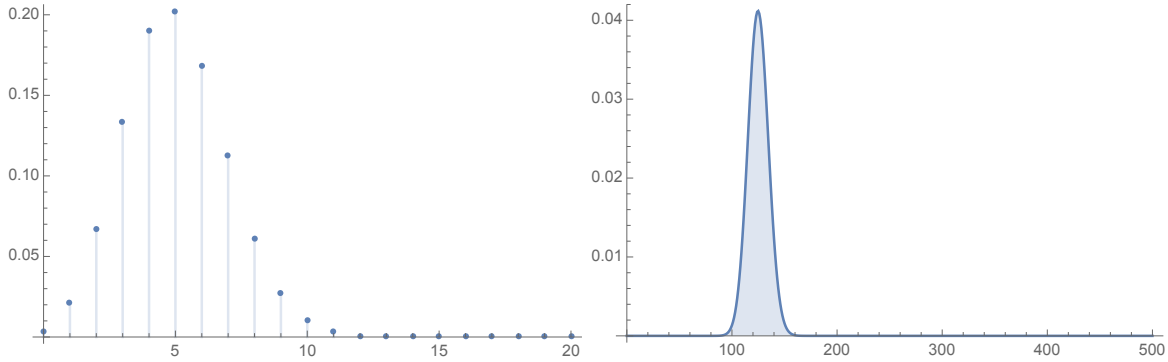
[Now we will investigate how well the training error predicts the test error, and how that differs for these two classifiers.]

Suppose all training pts & test pts are drawn independently from same prob. distribution $\mathcal{D}$ defined on domain P. [$\mathcal{D}$ also determines each point's label. Classes C and Not-C may have overlapping distributions.]

Let $h \in H$ be a hypothesis [a classifier]. h predicts a pt x is in class C if $x \in h$.
The risk aka generalization error $R(h)$ of h is the probability that h misclassifies a random pt x drawn from $\mathcal{D}$—i.e., the prob. that $x \in$ C but $x \notin h$ or vice versa.
[Risk is almost the same as the test error. To be precise, the risk is the mean test error for test points drawn randomly from $\mathcal{D}$. For a particular test set, sometimes the test error is higher, sometimes lower, but on average it is $R(h)$. If you had an infinite amount of test data, the risk and the test error would be the same.]

Let $X \subseteq P$ be a set of n training pts drawn from $\mathcal{D}$
The empirical risk aka training error $\hat{R}(h)$ is % of X misclassified by h.
[This matches the definition of empirical risk I gave you in Lecture 12, if you use the 0-1 loss function.]

h misclassifies each training pt w/prob. $R(h)$, so total misclassified has a binomial distribution.
As $n \to \infty$, $\hat{R}(h)$ better approximates $R(h)$.



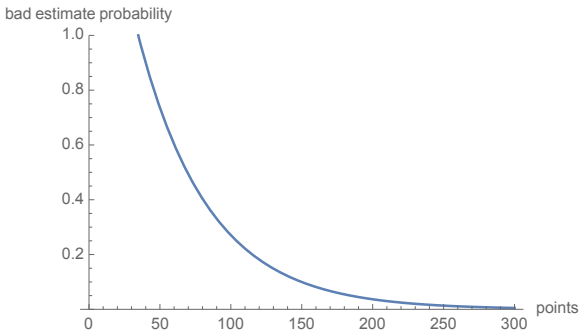
binom20.pdf, binom500.pdf Consider a hypothesis whose risk of misclassification is 25%. [Plotted are probability mass functions of the number of misclassified training points for 20 points and 500 points, respectively. For 20 points, the training error is not a reliable estimate of the risk: the hypothesis might get "lucky" with misleadingly low training error.]

[If we had infinite training data, this distribution would become infinitely narrow and the training error would always be equal to the risk. But we can't have infinite training data. So, how well does the training error approximate the risk?]

Hoeffding's inequality tells us prob. of bad estimate:

$$\Pr(|\hat{R}(h) - R(h)| > \epsilon) \leq 2e^{-2\epsilon^2 n}.$$

[Hoeffding's inequality is a standard result about how likely it is that a number drawn from a binomial distribution will be far from its mean. If n is big enough, then it's very unlikely.]



hoeffding.pdf [Hoeffding's bound for the unambitious $\epsilon = 0.1$. It takes at least 200 training points to have high confidence of attaining that error bound.]

[One reason this matters is because we will try to choose the best hypothesis. If the training error is a bad estimate of the test error, we might choose a hypothesis we think is good but really isn't. So we are happy to see that the likelihood of that decays exponentially in the amount of training data.]

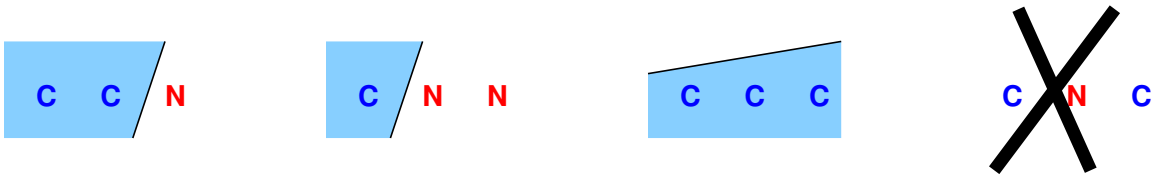
Idea for learning alg: choose $\hat{h} \in H$ that minimizes $\hat{R}(\hat{h})$! Empirical risk minimization.
[None of the classification algorithms we've studied actually do this, but only because it's computationally infeasible to pick the best hypothesis. Support vector machines can find a linear classifier with zero training error when the training data is linearly separable. But when it isn't, SVMs try to find a linear classifier with low training error, but they don't generally find the one with minimum training error. That's NP-hard.]
[Nevertheless, for the sake of understanding learning theory, we're going to pretend that we have the computational power to try every hypothesis and pick the one with the lowest training error.]

Problem: if too many hypotheses, some h with high $R(h)$ will get lucky and have very low $\hat{R}(h)$!

[This brings us to a central idea of learning theory. You might think that the ideal learning algorithm would have the largest class of hypotheses, so it could find the perfect one to fit the data. But the reality is that you can have so many hypotheses that some of them just get lucky and score far lower training error than their actual risk. That's another way to understand what "overfitting" is.]
[More precisely, the problem isn't too many *hypotheses*. Usually we have infinitely many hypotheses, and that's okay. The problem is too many dichotomies.]

Dichotomies

A dichotomy of X is $X \cap h$, where $h \in H$.
[A dichotomy picks out the training points that h predicts are in class C. Think of each dichotomy as a function assigning each training point to class C or class Not-C.]



[Draw this by hand. dichotomies.pdf] [Three examples of dichotomies for three points in a hypothesis class of linear classifiers, and one example (right) that is not a dichotomy.]

[For n training points, there could be up to 2^n dichotomies. The more dichotomies there are, the more likely it is that one of them will get lucky and have misleadingly low empirical risk.]

Extreme case: if H allows all 2^n possible dichotomies, $\hat{R}(\hat{h}) = 0$ even if every $h \in H$ has high risk.
[If our hypothesis class permits all 2^n possible assignments of the n training points to classes, then one of them will have zero training error. But that's true even if all of the hypotheses are terrible and have a large risk. Because the hypothesis class imposes no structure, we overfit the training points.]

If H induces Π dichotomies, $\Pr(\text{at least one dichotomy has } |\hat{R} - R| > \epsilon) \leq \delta$, where $\delta = 2\Pi\, e^{-2\epsilon^2 n}$.
[Let's fix a value of δ and solve for ϵ.] Hence with prob. $\geq 1 - \delta$, for every $h \in H$,

$$|\hat{R}(h) - R(h)| \leq \epsilon = \sqrt{\frac{1}{2n} \ln \frac{2\Pi}{\delta}}.$$

[This tells us that the smaller we make Π, the number of possible dichotomies, and the larger we make n, the number of training points, the more accurately the training error will approximate how well the classifier performs on test data.]

smaller Π or larger $n \Rightarrow$ training error probably closer to true risk (& test error).

[Smaller Π means we're less likely to overfit. We have less variance, but more bias. This doesn't necessarily mean the risk will be small. If our hypothesis class H doesn't fit the data well, both the training error and the test error will be large. In an ideal world, we want a hypothesis class that fits the data well, yet doesn't produce many dichotomies.]

Let $h^* \in H$ minimize $R(h^*)$; "best" classifier.
[Remember we picked the classifier $\hat{h}$ that minimizes the empirical risk. We really want the classifier h^* that minimizes the actual risk, but we can't know what h^* is. But if Π is small and n is large, the hypothesis $\hat{h}$ we have chosen is probably nearly as good as h^*.]

With prob. $\geq 1 - \delta$, our chosen $\hat{h}$ has nearly optimal risk:

$$R(\hat{h}) \leq \hat{R}(\hat{h}) + \epsilon \leq \hat{R}(h^*) + \epsilon \leq R(h^*) + 2\epsilon, \qquad \epsilon = \sqrt{\frac{1}{2n} \ln \frac{2\Pi}{\delta}}.$$

[This is excellent news! It means that with enough training data and a limit on the number of dichotomies, empirical risk minimization usually chooses a classifier close to the best one in the hypothesis class.]

Choose a δ and an ϵ.
The sample complexity is the # of training pts needed to achieve this ϵ with prob. $\geq 1 - \delta$:

$$n \geq \frac{1}{2\epsilon^2} \ln \frac{2\Pi}{\delta}.$$

[If Π is small, we won't need too many training points to choose a good classifier. Unfortunately, if $\Pi = 2^n$ we lose, because this inequality says that n has to be bigger than n. So the power set classifier can't learn much or generalize at all. We need to severely reduce Π, the number of possible dichotomies. One way to do that is to use a linear classifier.]

The Shatter Function & Linear Classifiers

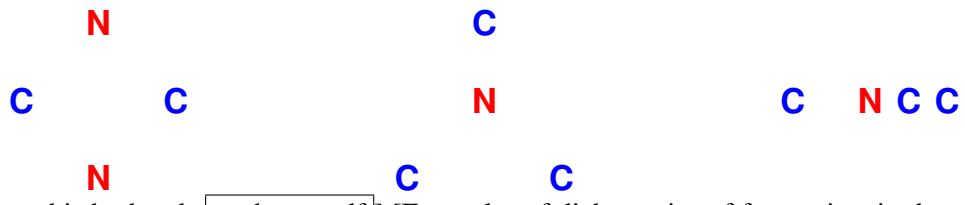
[How many ways can you divide n points into two classes with a hyperplane?]

# of dichotomies: $\Pi_H(X)$	$=$	$\lvert\{X \cap h : h \in H\}\rvert$	$\in [1, 2^n]$ where $n = \lvert X \rvert$
shatter function: $\Pi_H(n)$	$=$	$\max_{\lvert X\rvert = n, X \subseteq P} \Pi_H(X)$	[The most dichotomies of any point set of size n]

Example: Linear classifiers in plane. H = set of all halfplanes. $\Pi_H(3) = 8$:

[Draw this by hand. shatter.pdf] [Linear classifiers can induce all eight dichotomies of these three points. The other four dichotomies are the complements of these four.]

$\Pi_H(4) = 14$:
[Instead of showing you all 14 dichotomies, let me show you dichotomies that halfplanes *cannot* learn, which illustrate why no four points have 16 dichotomies.]



[Draw this by hand. unshatter.pdf] [Examples of dichotomies of four points in the plane that no linear classifier can induce.]

[This isn't a proof that 14 is the maximum, because we have to show that 15 is not possible for *any* four points in the plane. The standard proof uses a famous result called Radon's Theorem.]

Fact: for all range spaces, either $\Pi_H(n)$ is polynomial in n, or $\Pi_H(n) = 2^n \ \ \forall n \geq 0$.

[This is a surprising fact with deep implications. Imagine that you have n points, some of them training points and some of them test points. Either a range space permits every possible dichotomy of the points, and the training points don't help you classify the test points at all; or the range space permits only a polynomial subset of the 2^n possible dichotomies, so once you have labeled the training points, you have usually cut down the number of ways you can classify the test points dramatically. No shatter function ever occupies the no-man's-land between polynomial and 2^n.]

[For linear classifiers, we know exactly how many dichotomies there can be.]

Cover's Theorem [1965]: linear classifiers in $\mathbb{R}^d$ allow up to $\Pi_H(n) = 2\sum_{i=0}^{d}\binom{n-1}{i}$ dichotomies of n pts.

For $n \leq d+1$, $\Pi_H(n) = 2^n$.
For $n \geq d+1$, $\Pi_H(n) \leq 2\left(\frac{e(n-1)}{d}\right)^d$ [Observe that this is polynomial in n! With exponent d.]
and the sample complexity needed to achieve $R(\hat{h}) \leq \hat{R}(\hat{h}) + \epsilon \leq R(h^*) + 2\epsilon$ with prob. $\geq 1-\delta$ is

$$n \geq \frac{1}{2\epsilon^2}\left(d \ln \frac{n-1}{d} + d + \ln \frac{4}{\delta}\right).$$ [Observe that the logarithm turned the exponent d into a factor!]

Corollary: linear classifiers need only $n \in \Theta(d)$ training pts
for training error to accurately predict risk or test error.

[In a d-dimensional feature space, we need more than d training points to train an accurate linear classifier. But it's reassuring to know that the number we need is linear in d. By contrast, if we have a classifier that permits all 2^n possible dichotomies however large n is, then no amount of training data will guarantee that the training error of the hypothesis we choose approximates the true risk.]

[The constant hidden in that big-Θ notation can be quite large. For example, if you choose $\epsilon = 0.1$ and $\delta = 0.1$, then setting $n = 550\,d$ will always suffice. (For very large d, $n = 342\,d$ will do.) If you want a lot of confidence that you've chosen one of the best hypotheses, you have to pay for it with a large sample size.]

[This sample complexity applies even if you add polynomial features or other features, but you have to count the extra features in d. So the number of training points you need increases with the number of polynomial terms.]

VC Dimension

The Vapnik–Chervonenkis dimension of (P, H) is

$$\mathrm{VC}(H) = \max\{n : \Pi_H(n) = 2^n\}. \qquad \Leftarrow \text{Can be } \infty.$$

Say that H shatters a set X of n pts if $\Pi_H(X) = 2^n$.
$\mathrm{VC}(H)$ is size of largest X that H can shatter.
[This means that X is a point set for which all 2^n dichotomies are possible.]

[I told you earlier that if the shatter function isn't 2^n for all n, then it's a polynomial in n. The VC dimension is motivated by an observation that sometimes makes it easy to bound that polynomial.]

Theorem: $\Pi_H(n) \leq \sum_{i=0}^{\mathrm{VC}(H)} \binom{n}{i}$. Hence for $n \geq \mathrm{VC}(H)$, $\Pi_H(n) \leq \left(\frac{en}{\mathrm{VC}(H)}\right)^{\mathrm{VC}(H)}$.

[So the VC dimension is an upper bound on the exponent of the polynomial. This theorem is useful because often we can find an easy upper bound on the VC dimension. You just need to show that for some number n, no set of n points can have all 2^n dichotomies.]

Corollary: $O(\mathrm{VC}(H))$ training pts suffice for accuracy. [Again, the hidden constant is big.]

[If the VC dimension is finite, it tells us how the sample complexity grows with the number of features. If the VC dimension is infinite, no amount of training data will make the classifier generalize well.]

Example: Linear classifiers in plane.
Recall $\Pi_H(3) = 8$: there exist 3 pts shattered by halfplanes.
But $\Pi_H(4) = 14$: no 4 pts are shattered.
Hence:

- $\mathrm{VC}(H) = 3$ [The VC dimension of halfplanes is 3.]
- $\Pi_H(n) \leq \frac{e^3}{27} n^3$ [The shatter function is polynomial.]
- $O(1)$ sample complexity.

[The VC dimension doesn't always give us the tightest bound. In this example, the VC dimension promises that the number of ways halfplanes can classify the points is at worst cubic in n; but Cover's Theorem says it's quadratic in n. In general, linear classifiers in d dimensions have VC dimension $d + 1$, which is one dimension looser than the exponent Thomas Cover proved. That's not a big deal, though, as the sample complexity and the accuracy bound are both based on the *logarithm* of the shatter function. So if we get the exponent wrong, it only changes a constant in the sample complexity.]

[The important thing is simply to show that there is some polynomial bound on the shatter function at all. VC dimension is not the only way to do that, but often it's the easiest.]

[The main point you should take from this lecture is that if you want to have generalization, you need to limit the expressiveness of your hypothesis class so that you limit the number of possible dichotomies of a point set. This may or may not increase the bias, but if you don't limit the number of dichotomies at all, the overfitting could be very bad. If you limit the hypothesis class, your artificial child will only need to look at $O(d)$ cows to learn the concept of cows. If you don't, your artificial child will need to look at every cow in the world, and every non-cow too.]

B Bonus Lecture: The Kernel Trick

KERNELS

Recall featurizing map $\Phi : \mathbb{R}^d \to \mathbb{R}^D$. d input features; D features after featurization (Φ).
Degree-p polynomials blow up to $D \in \Theta(d^p)$ features.
[When d and p are not small, 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 polynomial decision functions, then each featurized feature vector has a length of roughly 4 million.]
Today, magically, we use those features without computing them!

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

Suppose $w = X^\top a = \sum_{i=1}^{n} a_i X_i$ for some $a \in \mathbb{R}^n$.
Substitute this identity into alg. and optimize n dual weights a (aka dual parameters)
instead of D primal weights w.

Kernel Ridge Regression

Center X and y so their means are zero: $X_i \leftarrow X_i - \mu_X$, $y_i \leftarrow y_i - \mu_y$, $X_{i,d+1} = 1$ [don't center the 1's!]
This lets us replace I' with I in normal equations:

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

[To kernelize ridge regression, we need the weights to be a linear combination of the training points. Unfortunately, that only happens if we penalize the bias term $w_{d+1} = \alpha$, as these normal equations do. Fortunately, when we center X and y, the "expected" value of the bias term is zero. The actual bias won't usually be exactly zero, but it will often be close enough that we won't do much harm by penalizing the bias term.]

Suppose a is a solution to

$$(XX^\top + \lambda I)a = y. \qquad \text{[Always has a solution if } \lambda > 0.\text{]}$$

Then $X^\top y = X^\top X X^\top a + \lambda X^\top a = (X^\top X + \lambda I)X^\top a$.
Therefore, $w = X^\top a$ is a solution to the normal equations, **and** w is a linear combo of training points!

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

Find a that minimizes $\|XX^\top a - y\|^2 + \lambda \|X^\top a\|^2$.

[We obtain this dual form by substituting $w = X^\top a$ into the original ridge regression cost function.]

Training: Solve $(XX^\top + \lambda I)a = y$ for a.
Testing: 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 $k(x, z) = x^T z$ 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 $K = XX^T$ be $n \times n$ kernel matrix. Note $K_{ij} = k(X_i, X_j)$.

K may be singular. If so, probably no solution if $\lambda = 0$. [Then we must choose a positive λ. But that's okay.] Always singular if $n > d + 1$. [But don't worry about the case $n > d + 1$, because you would only want to use the dual form when $d > n$, i.e., for polynomial features. But K could still be singular when $d > n$.]

Dual/kernel ridge regr. alg:

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

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

[**Important**: dual ridge regression produces the same predictions as primal ridge regression (with a penalized bias term)! The difference is the running time; the dual algorithm is faster if $d > n$, because the primal algorithm solves a $d \times d$ linear system, whereas the dual algorithm solves an $n \times n$ linear system.]

The Kernel Trick (aka Kernelization)

[Here's the magic part. We can compute a polynomial kernel without actually computing the features.]

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

Theorem: $(x^T z + 1)^p = \Phi(x)^T \Phi(z)$ for some $\Phi(x)$ containing every monomial in x of degree $0 \ldots p$.

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

$$\begin{aligned} (x^T z + 1)^2 &= x_1^2 z_1^2 + x_2^2 z_2^2 + 2x_1 z_1 x_2 z_2 + 2x_1 z_1 + 2x_2 z_2 + 1 \\ &= [x_1^2 \quad x_2^2 \quad \sqrt{2}x_1 x_2 \quad \sqrt{2}x_1 \quad \sqrt{2}x_2 \quad 1]\, [z_1^2 \quad z_2^2 \quad \sqrt{2}z_1 z_2 \quad \sqrt{2}z_1 \quad \sqrt{2}z_2 \quad 1]^T \\ &= \Phi(x)^T \Phi(z) \qquad \text{[This is how we're defining } \Phi\text{.]} \end{aligned}$$

[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 that appear in $\Phi(x)$, but they don't matter much, because the primal weights w will scale themselves to compensate. Even though we don't directly compute the primal weights, they implicitly exist in the form $w = X^T a$.]

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

Kernel ridge regr. replaces X_i with $\Phi(X_i)$: let $k(x, z) = \Phi(x)^T \Phi(z)$,
but doesn't compute $\Phi(x)$ or $\Phi(z)$; it computes $k(x, z) = (x^T z + 1)^p$.

Running times for 3 ridge algs:

	primal	dual (no kernel trick)	kernel
train	$O(D^3 + D^2 n)$	$O(n^3 + n^2 D)$	$O(n^3 + n^2 d)$
test (per test pt)	$O(D)$	$O(nD)$	$O(nd)$

[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 write out the final polynomial. The running time can be asymptotically smaller than D, the number of terms in the polynomial.]

Kernel Logistic Regression

Let $\Phi(X)$ be $n \times D$ matrix with rows $\Phi(X_i)^\top$. [$\Phi(X)$ is the design matrix of the featurized training points.]

Featurized logi. regr. with batch grad. descent:

$w \leftarrow 0$ [starting point is arbitrary]
repeat until convergence
  $w \leftarrow w + \epsilon\, \Phi(X)^\top (y - s(\Phi(X)\, w))$ apply s component-wise to vector $\Phi(X)\, w$
for each test pt z
  $h(z) \leftarrow s(w^\top \Phi(z))$

Dualize with $w = \Phi(X)^\top a$.
Then the code "$a \leftarrow a + \epsilon\,(y - s(\Phi(X)\, w))$" has same effect as "$w \leftarrow w + \epsilon\, \Phi(X)^\top (y - s(\Phi(X)\, w))$".
Let $K = \Phi(X)\, \Phi(X)^\top$. [The $n \times n$ kernel matrix; but we don't compute $\Phi(X)$—we use the kernel trick.]
Note that $Ka = \Phi(X)\, \Phi(X)^\top a = \Phi(X)\, w$. [And $\Phi(X)\, w$ appears in the algorithm above.]

Dual/kernel logistic regression:

$a \leftarrow 0$ [starting point is arbitrary]
$\forall i, j, \quad K_{ij} \leftarrow k(X_i, X_j)$ $\Leftarrow O(n^2 d)$ time (kernel trick)
repeat until convergence
  $a \leftarrow a + \epsilon\,(y - s(Ka))$ $\Leftarrow O(n^2)$ time/iteration [apply s component-wise]
for each test pt z
  $h(z) \leftarrow s\left(\sum_{i=1}^{n} a_i\, k(X_i, z)\right)$ $\Leftarrow O(nd)$ time/test pt [kernel trick]

[For classification, you can skip the logistic function $s(\cdot)$ and just compute the sign of the summation.]

[Kernel logistic regression computes the same answer as the primal algorithm, but the running time changes.]
Important: running times depend on original dimension d, not on length D of $\Phi(\cdot)$! Training for j iterations:
Primal: $O(nDj)$ time Dual (no kernel trick): $O(n^2 D + n^2 j)$ time Kernel: $O(n^2 d + n^2 j)$ time

Alternative training: stochastic gradient descent (SGD). Primal logistic SGD step is
$w \leftarrow w + \epsilon\, \left(y_i - s(\Phi(X_i)^\top w)\right)\, \Phi(X_i)$.

Dual logistic SGD maintains a vector $q = Ka \in \mathbb{R}^n$. Note that $q_i = (\Phi(X)\, w)_i = \Phi(X_i)^\top w$.
Let K_{*i} denote column i of K.

$a \leftarrow 0$; $q \leftarrow 0$; $\forall i, j, K_{ij} \leftarrow k(X_i, X_j)$ [If you choose a different starting point, set $q \leftarrow Ka$.]
repeat until convergence
  choose random $i \in [1, n]$
  $a_i \leftarrow a_i + \epsilon\,(y_i - s(q_i))$ $\Leftarrow O(1)$ time
  $q \leftarrow q + \epsilon\,(y_i - s(q_i))\, K_{*i}$ $\Leftarrow$ computes $q = Ka$ in $O(n)$ time, *not* $O(n^2)$ time

[SGD updates only one dual weight a_i per iteration; that's a nice benefit of the dual formulation. We cleverly update $q = Ka$ in linear time instead of performing a quadratic-time matrix-vector multiplication.]

Primal: $O(Dj)$ time Dual (no kernel trick): $O(n^2 D + nj)$ time Kernel: $O(n^2 d + nj)$ time

Alternative testing: If # of training points and test points both exceed D/d, classifying with primal weights w may be faster. [This applies to ridge regression as well.]

$w = \Phi(X)^\top a$ $\Leftarrow O(nD)$ time (once only)
for each test pt z
  $h(z) \leftarrow s\,(w^\top \Phi(z))$ $\Leftarrow O(D)$ time/test pt

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 dimensions, why don't we go all the way and generate feature vectors in an infinite-dimensional space?]

Gaussian kernel, aka radial basis fn kernel: there exists a $\Phi : \mathbb{R}^d \to \mathbb{R}^\infty$ such that

$$k(x,z) = \exp\left(-\frac{\|x-z\|^2}{2\sigma^2}\right) \qquad \text{[This kernel takes } O(d) \text{ time to compute.]}$$

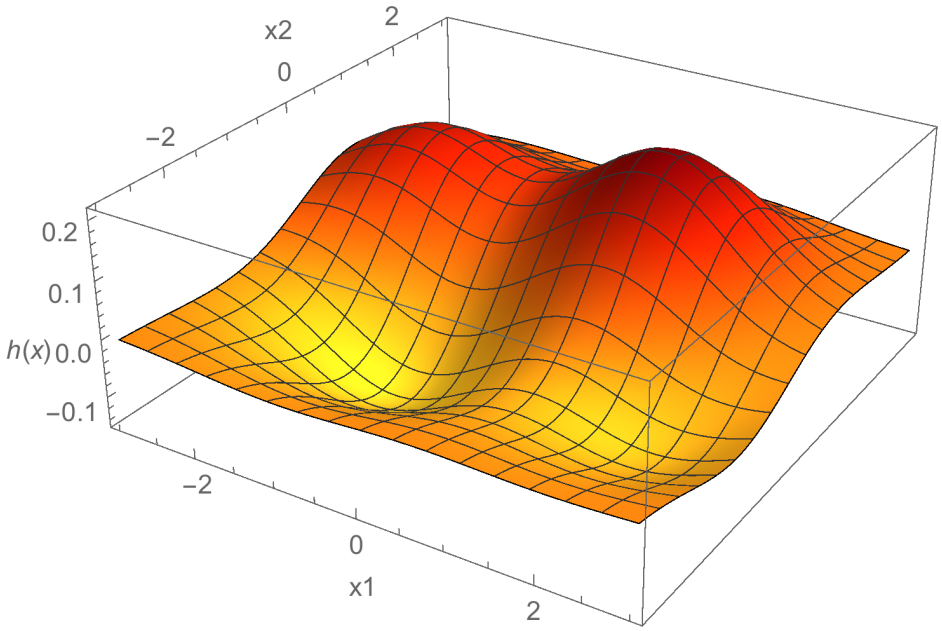
[In case you're curious, here's the feature vector that gives you this kernel, for the case where you have only one input feature per sample point.]

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 $\Phi(x)\cdot\Phi(z)$ is a series that converges to $k(x,z)$. 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. It's no longer a useful intuition. 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 training pts.
[The dual weights are the coefficients of the linear combination.]
[The Gaussians are a basis for the hypothesis.]



gausskernel.pdf [A hypothesis h that is a linear combination of Gaussians centered at four training points, two with positive weights and two with negative weights. If you use ridge regression with a Gaussian kernel, your "linear" regression will look something like this.]

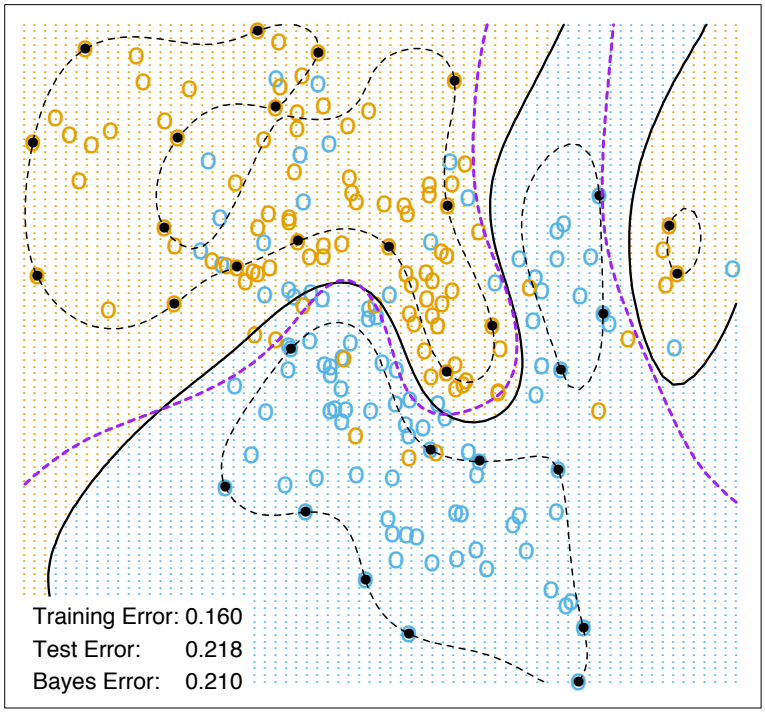
Very popular in practice! Why?

- Gives very smooth h. [In fact, h is infinitely differentiable; it's C^∞-continuous.]
- Behaves somewhat like k-nearest neighbors, but smoother.
- Oscillates less than polynomials (depending on σ).
- $k(x, z)$ interpreted as a similarity measure. Maximum when $z = x$; goes to 0 as distance increases.
- Training points "vote" for value at z, but closer points get weightier vote.

[The "standard" kernel $k(x, z) = x \cdot z$ assigns more weight to training point vectors that point in roughly the same direction as z. By contrast, the Gaussian kernel assigns more weight to training points near z.]

Choose σ by validation.
σ trades off bias vs. variance:

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



gausskernelsvm.pdf (ESL, Figure 12.3) [The decision boundary (solid black) of a soft-margin SVM with a Gaussian kernel. Observe that in this example, it comes reasonably close to the Bayes optimal decision boundary (dashed purple). The dashed black curves are the boundaries of the margin. The small black disks are the support vectors that lie on the margin boundary.]

[By the way, there are many other kernels that, like the Gaussian kernel, 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. There is an elaborate theory about how to construct valid kernel functions. However, you probably won't need it. The polynomial and Gaussian kernels are the two most popular by far.]

[As a final word, be aware that not every featurization Φ leads to a kernel function that can be computed faster than $\Theta(D)$ time. In fact, the vast majority cannot. Featurizations that can are rare and special.]

C Bonus Lecture: Spectral Graph Clustering

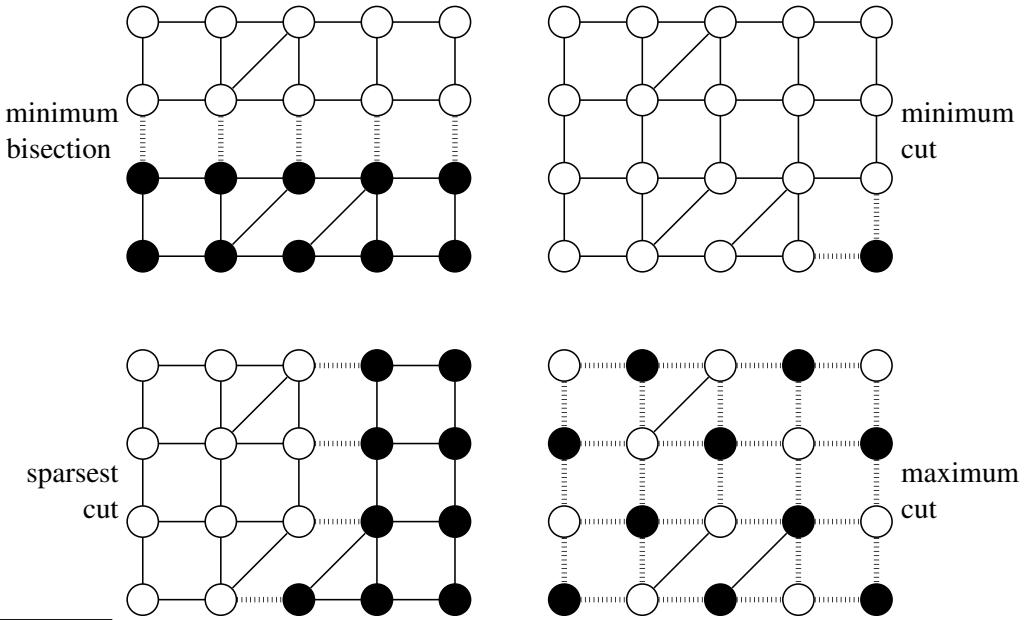
SPECTRAL GRAPH CLUSTERING

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.
[Today 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:

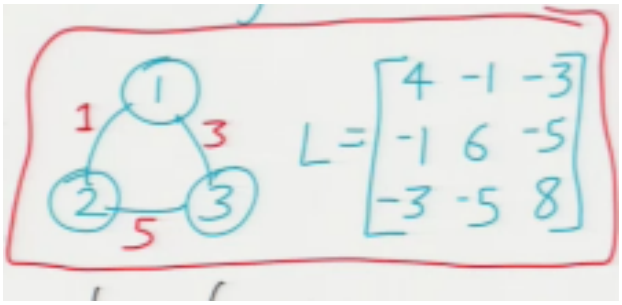
$$y_i = \begin{cases} 1 & \text{vertex } i \in G_1, \\ -1 & \text{vertex } i \in G_2. \end{cases}$$

Then $w_{ij}\dfrac{(y_i - y_j)^2}{4} = \begin{cases} w_{ij} & (i,j) \text{ is cut}, \\ 0 & (i,j) \text{ is not cut}. \end{cases}$

$$\begin{aligned} \text{Cut}(G_1, G_2) &= \sum_{(i,j)\in E} w_{ij}\frac{(y_i - y_j)^2}{4} && \text{[This is quadratic, so let's try to write it with a matrix.]} \\ &= \frac{1}{4}\sum_{(i,j)\in E}\left(w_{ij}y_i^2 - 2w_{ij}y_i y_j + w_{ij}y_j^2\right) \\ &= \frac{1}{4}\left(\underbrace{\sum_{(i,j)\in E} -2w_{ij}y_i y_j}_{\text{off-diagonal terms}} + \underbrace{\sum_{i=1}^{n} y_i^2 \sum_{k\neq i} w_{ik}}_{\text{diagonal terms}}\right) \\ &= \frac{y^\top L y}{4}, \end{aligned}$$

where $L_{ij} = \begin{cases} -w_{ij}, & i \neq j, \\ \sum_{k\neq i} w_{ik}, & i = j. \end{cases}$

L is symmetric, $n \times n$ Laplacian matrix for G.



[Draw this by hand graphexample.png]

[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.]

Define $\mathbf{1} = [1 \quad 1 \quad \ldots \quad 1]^\top$; then $L\mathbf{1} = \mathbf{0}$, so [It's easy to check that each row of L sums to zero.]
$\mathbf{1}$ is an eigenvector of L with eigenvalue 0.

[If G is a connected graph 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.]

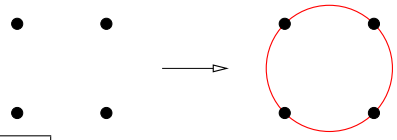
Bisection: exactly $n/2$ vertices in G_1, $n/2$ in G_2. Write $\mathbf{1}^T y = 0$.
[So we have reduced graph bisection to this constrained optimization problem.]
Minimum bisection:

Find y that minimizes $y^T L y$	
subject to $\forall i$, $y_i = 1$ or $y_i = -1$	← binary constraint
and $\mathbf{1}^T y = 0$	← 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 the continuous problem, 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 hypersphere of radius $\sqrt{n}$.

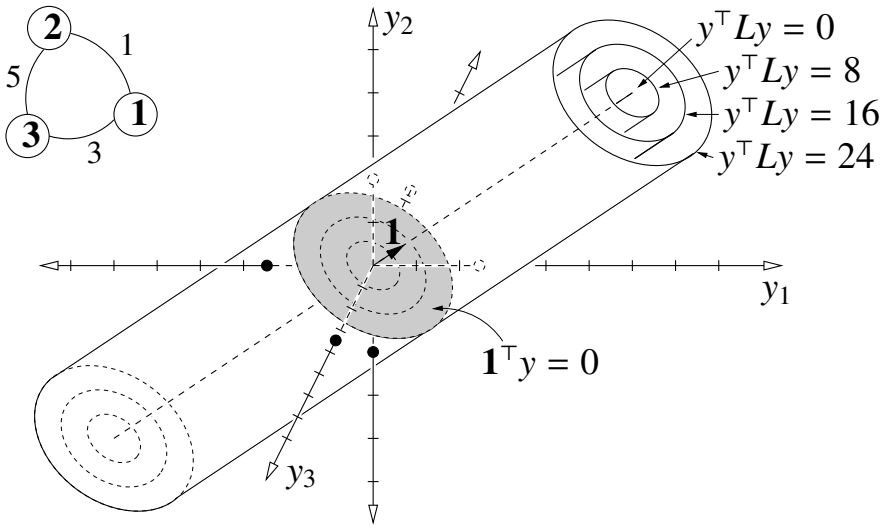


[Draw this by hand. circle.pdf] [Instead of constraining y to lie at a vertex of the hypercube, we constrain y to lie on the hypersphere through those vertices.]

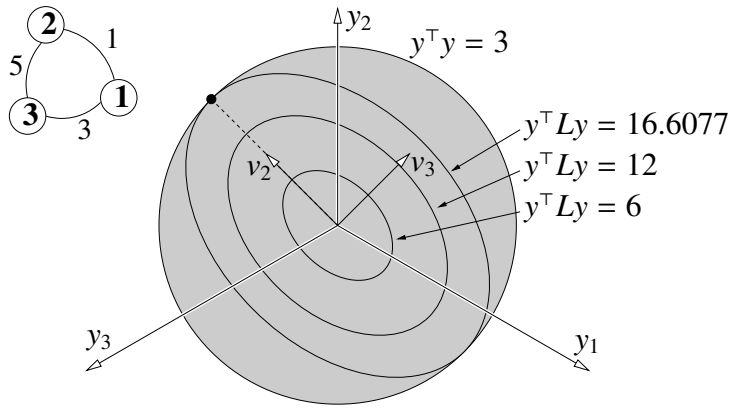
Relaxed problem:

Minimize $y^T L y$	
subject to $y^T y = n$	} = Minimize $\frac{y^T L y}{y^T y}$ = Rayleigh quotient of L & y
and $\mathbf{1}^T y = 0$	

(subject to same two constraints)



cylinder.pdf [The isosurfaces of $y^T L y$ are elliptical cylinders. The gray cross-section is the hyperplane $\mathbf{1}^T y = 0$. We seek the point that minimizes $y^T L y$, subject to the constraints that it lies on the gray cross-section and that it lies on a sphere centered at the origin.]

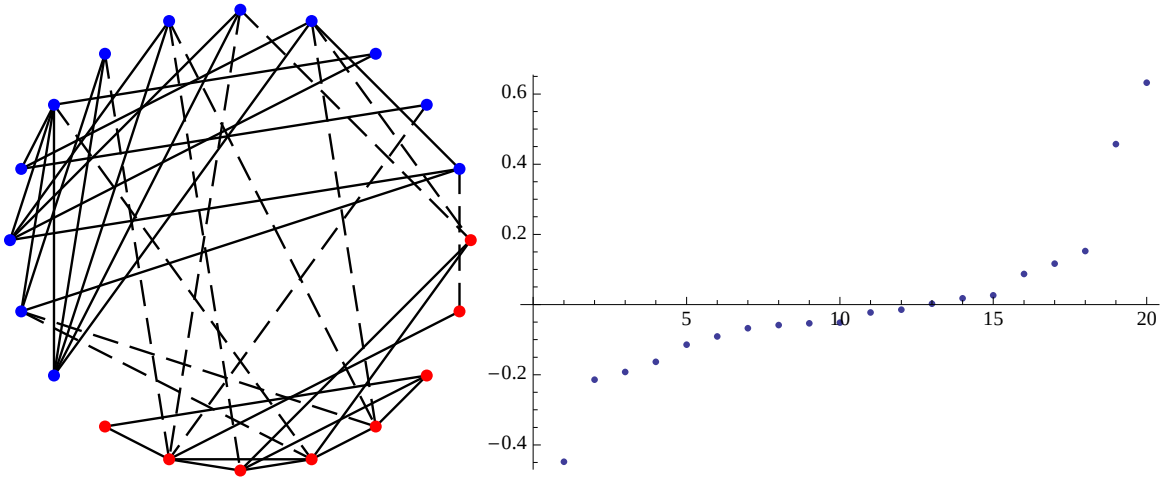


endview.pdf [The same isosurfaces restricted to the hyperplane $\mathbf{1}^\top y = 0$. The solution is constrained to lie on the outer circle.]

[You should remember this Rayleigh quotient from the lecture on PCA. As I said then, when you see a Rayleigh quotient, you should smell eigenvectors 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.]

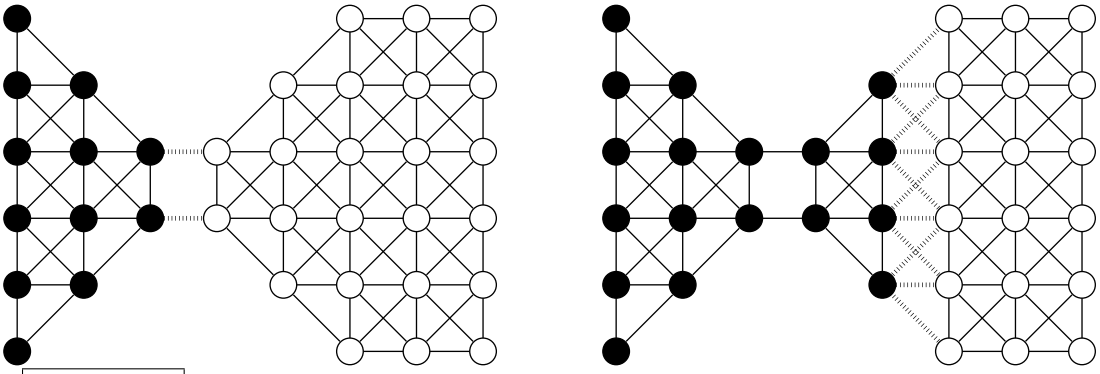
Let λ_2 = second-smallest eigenvalue of L.
Eigenvector v_2 is the Fiedler vector. v_2 solves the relaxed problem.
[It would be wonderful if every component of the Fiedler vector was 1 or -1, but that happens more or less never. So we round v_2. The simplest way is to round all positive entries to 1 and 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 updating the sparsity, 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 unbalanced cut if it means we cut fewer edges. Even though our discrete problem was the minimum bisection problem, our relaxed, continuous problem will be an approximation of the sparsest cut problem. This is a bit counterintuitive.]

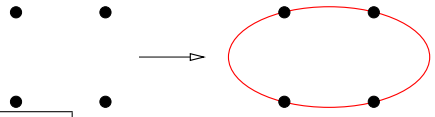


lopsided.pdf [A graph for which an unbalanced cut (left) is sparser than a balanced one (right).]

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.]

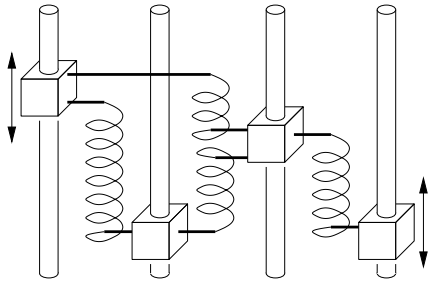
Now solution is Fiedler vector of generalized eigensystem $Lv = \lambda Mv$.
[Most algorithms for computing eigenvectors and eigenvalues of symmetric matrices can easily be adapted to compute eigenvectors and eigenvalues of symmetric *generalized* eigensystems too.]

[For the grad students, here's the most important theorem in spectral graph partitioning.]

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 $\geq \lambda_2/2$.

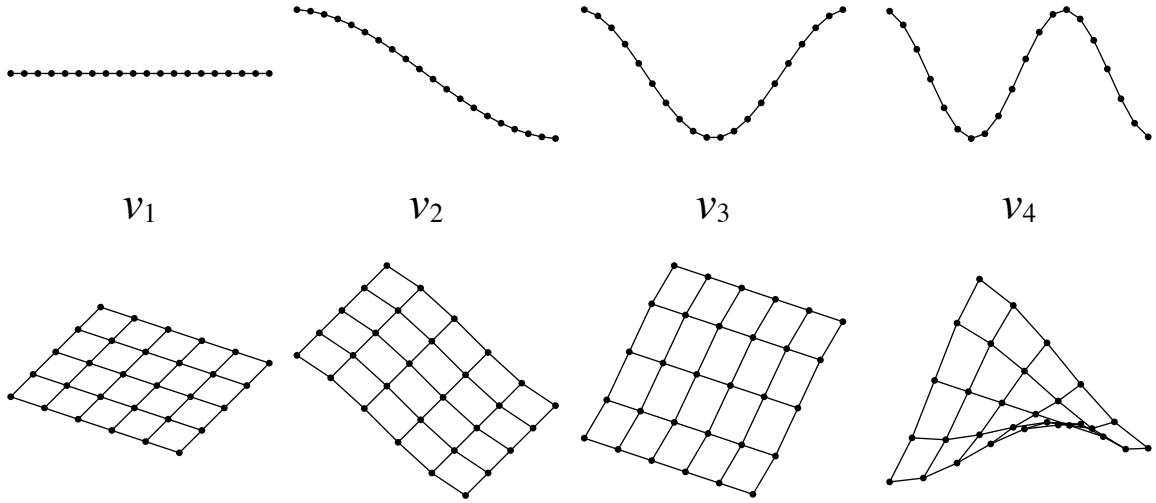
[So the spectral partitioning algorithm is an approximation algorithm, albeit not one with a constant factor of approximation. Cheeger's inequality is a very famous result in spectral graph theory, because it's one of the most important cases where you can relax a combinatorial optimization problem to 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

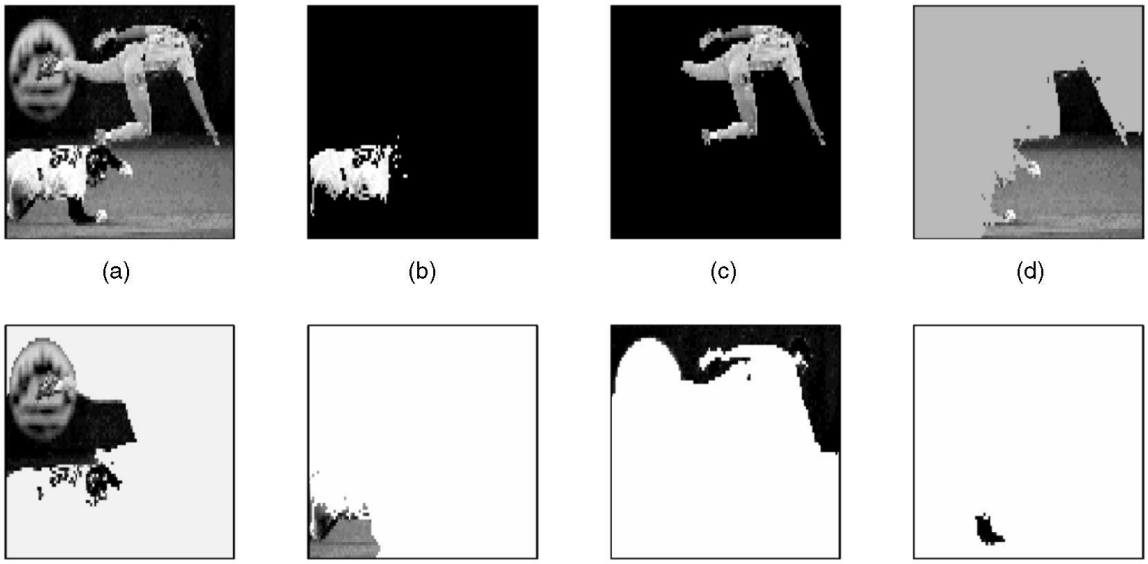
Partition G into 2 subgraphs; recursively partition them.
[The sparsity is a good criterion for graph clustering. Use G's sparsest cut to divide it into two subgraphs, then recursively cut them. You can stop when you have the right number of clusters. Alternatively, you can make a finer tree and then prune it back.]

The Normalized Cut

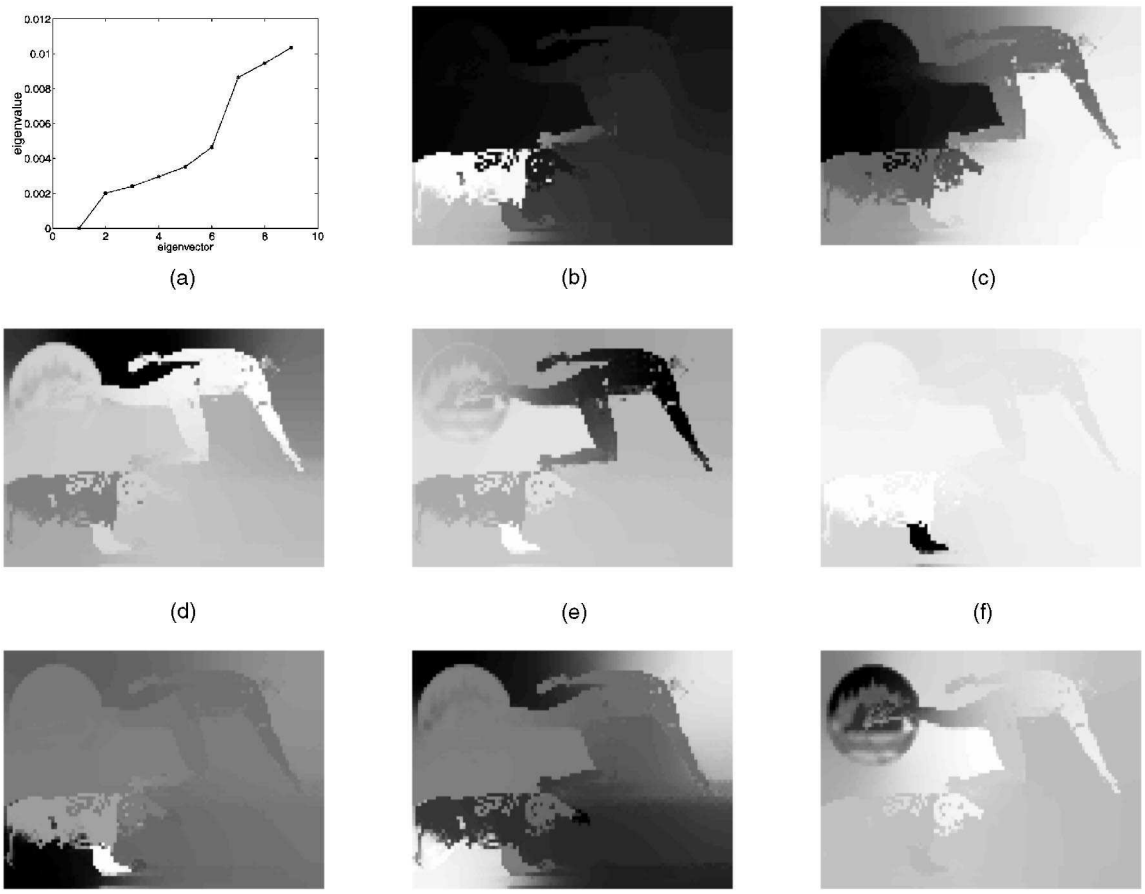
Set vertex i's mass $M_{ii} = L_{ii}$. [Sum of edge weights adjoining vertex i.]
[That is how we define a normalized cut, which turns out to be a good choice for many different applications.]
Popular for image segmentation.
[Image segmentation is the problem of looking at a photograph and separating it into different objects. To do that, we define a graph on the pixels.]
For pixels with coordinate p_i, brightness b_i, use graph weights

$$w_{ij} = \exp\left(-\frac{\|p_i - p_j\|^2}{\alpha} - \frac{|b_i - b_j|^2}{\beta}\right) \qquad \text{or zero if } \|p_i - p_j\| \text{ large.}$$

[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.]

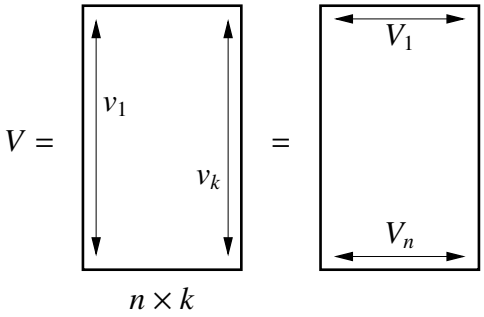
Invented by [our own] Prof. Jitendra Malik and his student Jianbo Shi.

D Bonus Lecture: Multiple Eigenvectors; Latent Factor Analysis

Clustering w/Multiple Eigenvectors

[When we use the Fiedler vector for spectral graph clustering, it tells us how to divide a graph into two graphs. If we want more than two clusters, we can use divisive clustering: we repeatedly cut the subgraphs into smaller subgraphs by computing their Fiedler vectors. However, there are several other methods to subdivide a graph into k clusters in one shot that use multiple eigenvectors rather than just the Fiedler vector v_2. These methods sometimes give better results. They use k eigenvectors in a natural way to cluster a graph into k subgraphs.]

For k clusters, compute first k eigenvectors $v_1 = \mathbf{1}, v_2, \ldots, v_k$ of generalized eigensystem $Lv = \lambda Mv$.
Scale them so that $v_i^\top M v_i = 1$. E.g., $v_1 = \frac{1}{\sqrt{\sum M_{ii}}}\mathbf{1}$. Now $V^\top M V = I$. [The eigenvectors are M-orthogonal.]



[V's columns are the eigenvectors with the k smallest eigenvalues.]
[Yes, we do include the all-1's vector v_1 as one of the columns of V.]

[Draw this by hand. eigenvectors.pdf]

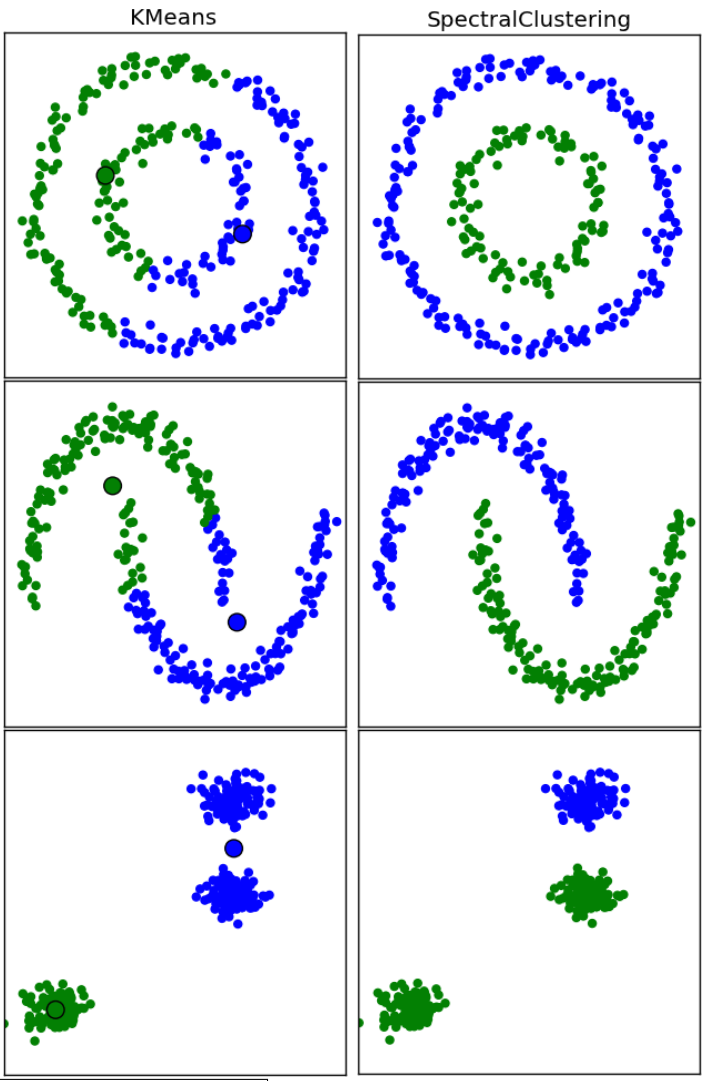
Row V_i is spectral vector [my name] for vertex i. [The rows are vectors in a k-dimensional space I'll call the "spectral space." When we were using just one eigenvector, it made sense to cluster vertices together if their components were close together. When we use more than one eigenvector, it turns 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] [A 2D example showing two clusters on a circle. If the 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, Andrew Ng [when he was still a student at Berkeley], Yair Weiss.

[This wasn't the first algorithm to use 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. To do that, you need to know how frequently each word occurs in general. I'll omit the details, but this is the common practice.]

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 large δ_i, u_i and v_i represent a cluster of documents & terms.

- Large components in u_i mark docus using similar/related terms, i.e., a genre.
- " " " v_i mark frequent terms in that genre.
- 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 similar vocabulary 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.]

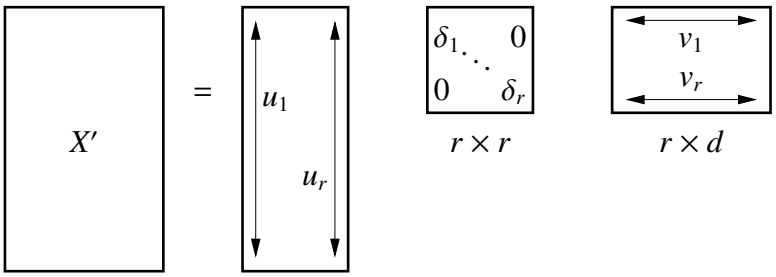
Like clustering, but clusters overlap: if u_1 picks out romances &
u_2 picks out histories, they both pick out historical romances.

[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, suburban 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. This motivates approximating a matrix by using only some of its singular vectors.]

Truncated SVD $X' = \sum_{i=1}^{r} \delta_i u_i v_i^\top$ is a low-rank approximation of X, of rank r. [Assuming $\delta_r > 0$.]

[We choose the singular vectors with the largest singular values, because they carry the most 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 that probably has low rank. If that assumption is partly true, then the reduced-rank matrix X' might be better than the input X.]
- Matrix compression. [As you can see above, if we use a low-rank approximation with a small rank r, we can express the approximate matrix as an SVD that takes up much less space than the original matrix. Often this low-rank approximation supports faster matrix computations.]
- 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. We want to fill in the missing values. 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.]

PREDICTING PERSONALITY FROM FACES

www.nature.com/scientificreports

SCIENTIFIC REPORTS

OPEN

Signatures of personality on dense 3D facial images

Sile Hu[1], Jieyi Xiong[1,2], Pengcheng Fu[3], Lu Qiao[1], Jingze Tan[4], Li Jin[4] & Kun Tang[1]

Received: 27 July 2016
Accepted: 27 January 2017
Published online: 06 March 2017

It has long been speculated that cues on the human face exist that allow observers to make reliable judgments of others' personality traits. However, direct evidence of association between facial shapes and personality is missing from the current literature. This study assessed the personality attributes of 834 Han Chinese volunteers (405 males and 429 females), utilising the five-factor personality model ('Big Five'), and collected their neutral 3D facial images. Dense anatomical correspondence was established across the 3D facial images in order to allow high-dimensional quantitative analyses of

hu.pdf

Hu et. al (2017).

Big Five (BF) model of personality:

- O: openness
- C: conscientiousness
- E: extraversion
- A: agreeableness
- N: neuroticism

[Researchers have found that these five personality factors are approximately orthogonal to each other. They are highly heritable and highly stable during adulthood.]

Can we predict these traits from 3D faces?

[Studies have shown that people looking at photographs of static faces with neutral expressions can identify the traits better than chance, especially for conscientiousness, extraversion, and agreeableness. This experiment asks whether machine learning can do the same with 3D reconstructions of faces. The subjects were 834 Han Chinese volunteers in Shanghai, China. We don't know whether any of these results might generalize to people who are not Han Chinese.]

[The faces were scanned in high-resolution 3D and a non-rigid face registration system was used to fit a grid of 32,251 vertices to each face in a manner that maps each vertex to an appropriate landmark on the face. (They call this "anatomical homology.") So the design matrix X was $834 \times 100{,}053$, representing 834 subjects with 32,251 3D features each.]

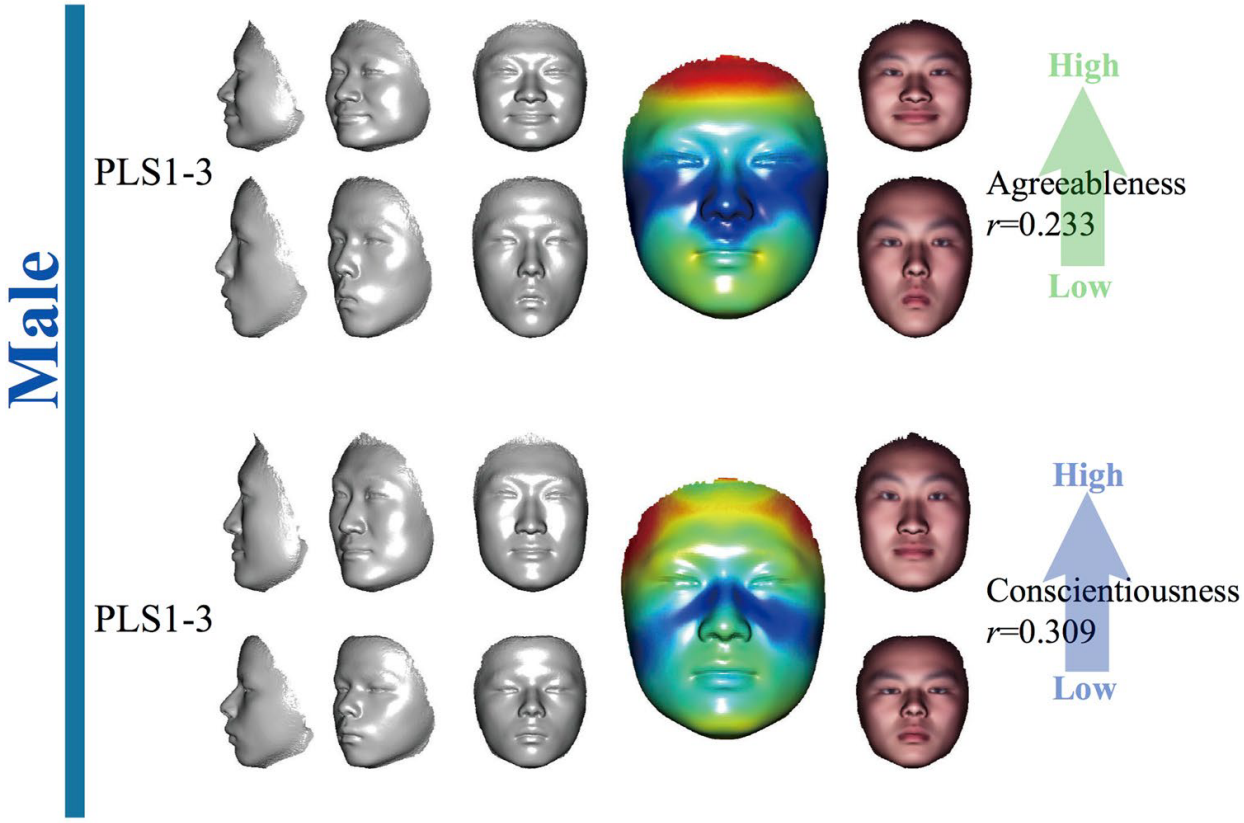
[Subject personalities were evaluated with a self-questionnaire, namely our own Berkeley Personality Lab's Big Five Inventory, translated into Chinese. The authors treated men and women separately.]

Uses partial least squares (PLS) to find associations between personality & faces.

[Everything from here to the end is spoken, not written.]

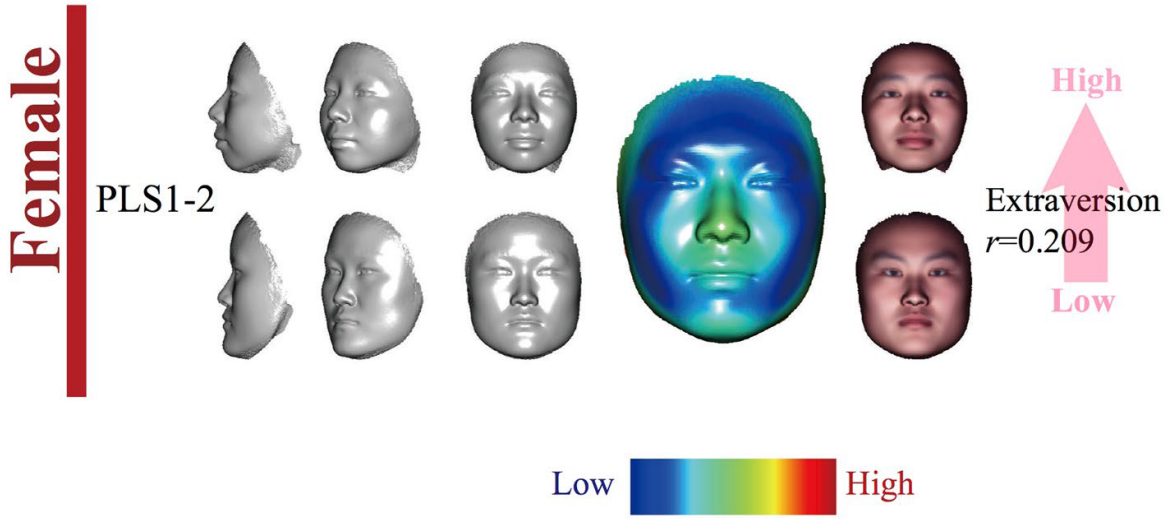
Partial least squares (PLS) is like a supervised version of PCA. It takes in two matrices X and Y with the same number of rows. In our example, X is the face data and Y is the personality data for the 834 subjects. Like PCA, PLS finds a set of vectors in face space that we think of as the most important components. But whereas PCA looks for the directions of maximum variation in X, PLS looks for the directions in X that maximize the correlation with the personality traits in matrix Y.

The researchers found the top 20 or so PLS components and used cross-validation to decide which components have predictive power for each personality trait. They found that the top two components for extraversion in women were predictive, but no components for the other four traits in women were predictive. Men are easier to analyze: they found two or three components were predictive for each of extraversion, agreeableness, conscientiousness, and neuroticism in men. However, the correlations were statistically significant only for agreeableness and conscientiousness.



male.pdf [The relationship between male faces, agreeableness, and conscientiousness. The large, colored faces are the mean faces. Colors indicate the values in the most predictive PLS component vector.]

More agreeable men correlate with much wider mouths that look a bit smiley even when neutral; stronger, forward jaws; wider noses; and shorter faces, especially shorter in the forehead, compared to less agreeable men. More conscientious men tend to have higher, wider eyebrows; wider, opened eyes; a withdrawn upper lip with more mouth tension; and taller faces with more pronounced brow ridges (the bone protuberance above the eyes). The authors note that men with low A and C scores look both more relaxed and more indifferent.



female.pdf [The relationship between female faces and extraversion. The large, colored face is the mean face. Colors illustrate the most predictive PLS component vector.]

More extraverted women correlate with rounder faces, especially in profile, with a more protruding nose and lips but a recessed chin, whereas the introverts have more flat, square-shaped faces. To my eyes, the extraverts also have more expressive mouths.

It's interesting is that physiognomy, the art of judging character from facial shape, used to be considered a pseudoscience, but it's been making a comeback in recent years with the help of machine learning. One reason it fell into disrepute is because, historically, it was sometimes applied across races in fallacious and insulting ways. But if you want to train classifiers that guess people's personalities with some accuracy, you probably need a different classifier for each race. This is a classifier trained exclusively for one race, Han Chinese, which is probably part of why it works as well as it does. If you tried to train one classifier to work on many different races, I suspect its performance would be much worse.

Another thing that's notable is that the authors were able to find statistically significant correlations for some personality traits, but the majority of traits defeated them. So while physiognomy has some predictive power, it's only weakly predictive. It's an open question whether machine learning will ever be able to predict personality from visual information substantially better than this or not. Adding a time dimension and incorporating people's movements and dynamic facial expressions seems like a promising way to improve personality predictions.

Tools like this raise some ethical issues. The one that concerns me the most is that, if tools like this are emerging now, many governments probably already had similar tools ten years ago, and have probably been using them to profile us.

One student asked whether these methods might be used by employers to screen prospective employees. I think that tools like this are inferior to simply giving an interviewee a personality test. Such tests are legal in the USA, so long as their questions are not found to violate an employee's right to privacy and the results are not used to discriminate against legally protected groups. The most troubling part of using physiognomy to screen employees would not be that personality testing is unlawful. (It isn't, and quite a few companies do it.) It would be that physiognomy isn't nearly accurate enough. An employer who uses a poorly designed or unvalidated personality test to make personnel decisions might run a higher risk that a court might rule that the test could have a discriminatory effect, violating Title VII of the Civil Rights Act of 1964. Also, they probably won't make good decisions. But perhaps in the future, better measurements, better statistical procedures, and better algorithms *might* overcome these problems.

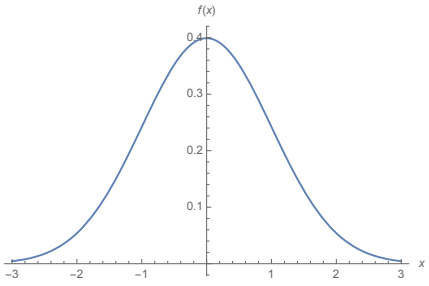
E Bonus Lecture: High Dimensions; Random Projection

THE GEOMETRY OF HIGH-DIMENSIONAL SPACES

[High-dimensional geometry sometimes acts in ways that are completely counterintuitive, defying our intuitions from low-dimensional geometry.]

Consider a random point $p \sim \mathcal{N}(0, I) \in \mathbb{R}^d$.
What is the distribution of its length?

[Looking at the one-dimensional normal distribution, you would expect it to be very common that the length is close to zero, a bit less common that the length is close to 1 or −1, and not rare for the length to be close to 2 or −2. But in high dimensions, that intuition is completely wrong.]



normal.pdf [A one-dimensional normal distribution.]

[If the dimension is very high, the vast majority of the random points are at approximately the same distance from the mean. So they lie in a thin shell. Why? To answer that, let's study the square of the distance. By Pythagoras' Theorem, the squared distance from p to the mean is]

$$\|p\|^2 = p_1^2 + p_2^2 + \ldots + p_d^2$$

[Each component p_i is sampled independently from a univariate normal distribution with mean zero and variance one. The square of a component, p_i^2, is said to come from a chi-squared distribution. So is $\|p\|^2$.]

$$p_i \sim \mathcal{N}(0,1), \quad p_i^2 \sim \chi^2(1), \quad E[p_i^2] = 1, \quad \mathrm{Var}(p_i^2) = 2, \quad \|p\|^2 \sim \chi^2(d)$$

[Recall that when you add d independent, identically distributed random numbers, you scale their mean and variance by d, and the standard deviation is the square root of the variance.]

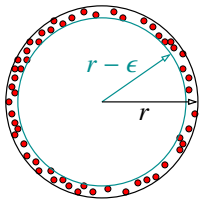
$$\begin{aligned} E[\|p\|^2] &= d\,E[p_1^2] = d \\ \mathrm{Var}(\|p\|^2) &= d\,\mathrm{Var}(p_1^2) = 2d \\ \mathrm{SD}(\|p\|^2) &= \sqrt{2d} \end{aligned}$$

For large d, $\|p\|$ is concentrated in a thin shell around radius $\sqrt{d}$ with a thickness proportional to $\sqrt[4]{2d}$.
[The mean value of $\|p\|$ isn't exactly $\sqrt{d}$, but it is close, because the mean of $\|p\|^2$ is d and the standard deviation is much, much smaller. Likewise, the standard deviation of $\|p\|$ isn't exactly $\sqrt[4]{2d}$, but it's close.]

[So if d is about a million, imagine a million-dimensional egg whose radius is about 1,000, and the thickness of the shell is about 67, which is about 10 times the standard deviation. The vast majority of random points are in the eggshell. Not inside the egg; actually in the shell itself. It is counterintuitive that random vectors sampled from a high-dimensional normal distribution almost all have almost the same length.]

[There is a statistical principle hiding here. Suppose you want to estimate the mean of a distribution—in this case, the distribution $\chi^2(1)$. The standard way to do that is to sample very many numbers from the distribution and take their mean. The more numbers you sample, the more accurate your estimate is—that is, the smaller the standard deviation of your sample mean is. When we sample a vector from a million-dimensional normal distribution and compute its length, that's exactly what we're doing!]

What about a uniform distribution? Consider concentric spheres of radii r & $r - \epsilon$.



[Draw this by hand concentric.pdf] [Concentric balls. In high dimensions, almost every point chosen uniformly at random in the outer ball lies outside the inner ball.]

Volume of outer ball $\propto r^d$
Volume of inner ball $\propto (r-\epsilon)^d$
Ratio of inner ball volume to outer =

$$\frac{(r-\epsilon)^d}{r^d} = \left(1 - \frac{\epsilon}{r}\right)^d \approx \exp\left(-\frac{\epsilon d}{r}\right) \qquad \text{which is small for large } d.$$

E.g., if $\frac{\epsilon}{r} = 0.1$ & $d = 100$, inner ball has $0.9^{100} = 0.0027\%$ of volume.

Random points from uniform distribution in ball: nearly all are in thin outer shell.
" " " Gaussian " : nearly all are in some thin shell.

Consequences:
- In high dimensions, sometimes the nearest neighbor and 1,000th-nearest neighbor don't differ much!
- k-means clustering and nearest neighbor classifiers are less effective for large d.

Angles between Random Vectors

What is the angle θ between a random $p \sim \mathcal{N}(0, I) \in \mathbb{R}^d$ and an arbitrary $q \in \mathbb{R}^d$?

Without loss of generality, set $q = [1 \quad 0 \quad 0 \ldots 0]^\top$.
[The value of q doesn't matter, because the direction that p points in is uniformly distributed over all possible directions. By a formula we learned early this semester, the angle between p and q is θ, where . . .]

$$\cos\theta = \frac{p \cdot q}{\|p\| \, \|q\|} = \frac{p_1}{\|p\|}$$

$$E[\cos\theta] = 0; \qquad \mathrm{SD}(\cos\theta) \approx \frac{\mathrm{SD}(p_1)}{\sqrt{d}} = \frac{1}{\sqrt{d}}$$

If d is large, $\cos\theta$ is almost always very close to zero; θ is almost always very close to 90°!

[In high-dimensional spaces, two random vectors are almost always very close to orthogonal. To put it another way, an arbitrary vector is almost orthogonal to the vast majority of all the other vectors!]

[A former CS 189/289A head TA, Marc Khoury, has a nice short essay entitled "Counterintuitive Properties of High Dimensional Space", which you can read at https://people.eecs.berkeley.edu/~jrs/highd]

RANDOM PROJECTION

An alternative to PCA as preprocess for clustering, classification, regression.
Approximately preserves distances between points!

[We project onto a random subspace instead of the PCA subspace, but sometimes it preserves distances better than PCA. Because it roughly preserves the distances, algorithms like k-means clustering and nearest neighbor classifiers will give similar results to what they would give in high dimensions, but they run much faster. It works best when you project a very high-dimensional space to a medium-dimensional space.]

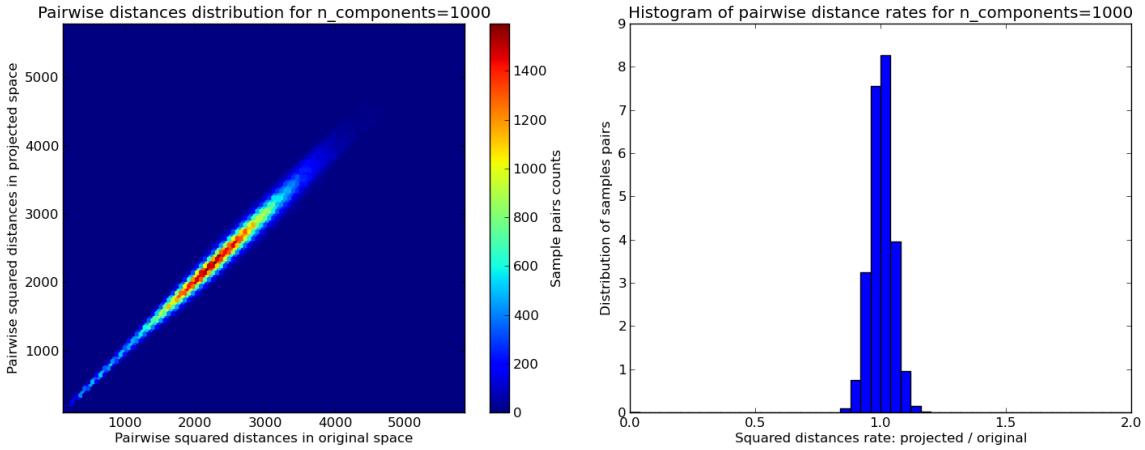
Pick a small ϵ, a small δ, and a random subspace $S \subset \mathbb{R}^d$ of dimension k, where $k = \left\lceil \frac{2\ln(1/\delta)}{\epsilon^2/2 - \epsilon^3/3} \right\rceil$.

For any pt q, let $\hat{q}$ be orthogonal projection of q onto S, multiplied by $\sqrt{\frac{d}{k}}$.

[The multiplication by $\sqrt{d/k}$ helps preserve the distances between points after you project.]

Johnson–Lindenstrauss Lemma (modified):
For any two pts $q, w \in \mathbb{R}^d$, $(1-\epsilon)\|q-w\|^2 \leq \|\hat{q}-\hat{w}\|^2 \leq (1+\epsilon)\|q-w\|^2$ with probability $\geq 1-2\delta$.
Typical values: $\epsilon \in [0.02, 0.5]$, $\delta \in [1/n^3, 0.05]$. [You choose ϵ and δ according to your needs.]

[The squared distance between two points after projecting changes by less than 2%, or less than 50%, as you wish. In practice, experiment with k to find the best speed-accuracy tradeoff. If you want all inter-sample-point distances to be accurate, you should set δ smaller than $1/n^2$, so you need a subspace of dimension $\Theta(\log n)$. Reducing δ doesn't cost much (because of the logarithm), but reducing ϵ costs more. You can bring 1,000,000 sample points down to a 10,000-dimensional space with at most a 6% error in the distances.]
[What is remarkable about this result is that the dimension d of the input points doesn't matter!]



100000to1000.pdf [Comparison of inter-point distances before and after projecting points in 100,000-dimensional space down to 1,000 dimensions.]

[Why does this work? A random projection of $q - w$ is like taking a random vector and selecting k components. The mean of the squares of those k components approximates the mean for the whole population.]

[How do you get a uniformly distributed random projection direction? You can choose each component from a univariate Gaussian distribution, then normalize the vector to unit length. How do you get a random subspace? You can choose k random directions, then use Gram–Schmidt orthogonalization to make them mutually orthonormal. Interestingly, Indyk and Motwani show that if you skip the expensive normalization and Gram–Schmidt steps, random projection still works almost as well, because random vectors in a high-dimensional space are nearly equal in length and nearly orthogonal to each other with high probability.]