Machine Learning Cheatsheet

Janet Matsen's Machine Learning (ML) notes from CSE 446, Winter 2016. `http://courses.cs.washington.edu/courses/cse446/16wi/`
Used LaTeX template from an existing Statistics cheat sheet: `https://github.com/wzchen/probability_cheatsheet`, by William Chen (`http://wzchen.com`) and Joe Blitzstein.

Last Updated March 16, 2016

Essential ML ideas

- Never ever ever touch the test set
- You know you are overfitting when there is a big test between train and test results. E.g. metric of percent wrong.
- Need to be comfortable taking a hit on fitting accuracy if you can get a benefit on the result.
- Bias vs variance trade-off. High bias when the model is too simple & doesn't fit the data well. High variance is when small changes to the data set lead to large solution changes.
- If features are non discriminative in the beginning, they don't work for any classifier.
- Your feature vector often has a smaller dimension that the feature space. If you have too long of a feature vector, you may get overfitting.
- You need to prevent the optimizer from getting an easy way out.
- Whenever we are building a discriminative classifier, we should not expect it to reason about things it has never seen before You can't classify on something that has only seen + points. The classifier needs to see some + and some - or it will fail miserably.
- We should always avoid making hard decisions early. E.g. don't put a lot of trust in classifiers early on in bagging.

You can do l_2 normalization for a feature vector to get a unit vector: Convert x to $\hat{x}$ so that if you form $||\hat{x}||_2^2 = 1$ Can also do l_1

Math/Stat Review

Random Variable X belongs to set Ω

Conditional Probability *is* Probability $P(A|B)$ is a probability function for any fixed B. Any theorem that holds for probability also holds for conditional probability. $P(A|B) = P(A \cap B)/P(B)$

Bayes' Rule - Bayes' Rule unites marginal, joint, and conditional probabilities. We use this as the definition of conditional probability.

$$P(\mathbf{A}|\mathbf{B}) = \frac{P(\mathbf{A} \cap \mathbf{B})}{P(\mathbf{B})} = \frac{P(\mathbf{B}|\mathbf{A})P(\mathbf{A})}{P(\mathbf{B})}$$

$$P(A = a \mid B) = \frac{P(A = a)P(B \mid A = a)}{\sum_{a'} P(A = a)P(B \mid A = a)}$$

Law of Total Probability : $\sum_x P(X = x) = 1$

Product Rule : $P(A, B) = P(A \mid B) \cdot P(B)$

Sum Rule : $P(A) = \sum_{x \in \Omega} P(A, B = b)$

i.i.d : $D = \{x_i | i = 1 \dots n\}, P(D|\theta) = \prod_i P(x_i \mid \theta)$

Vocab:

- **likelihood function** $L(\theta|O)$ is called as the likelihood function. θ = unknown parameters, O is the observed outcomes. The likelihood function is conditioned on the observed O and that it is a function of the unknown parameters θ. Not a probability density function.
- **"likelihood" vs "probability"**: if discrete, $L(\theta|O) = P(O|\theta)$. If continuous, $P(O|\theta) = 0$ so instead we estimate θ given O by maximizing $L(\theta|O) = f(O|\theta)$ where f is the pdf associated with the outcomes O.
- **hypothesis space**

Law of Total Probability (LOTP)

Let $B_1, B_2, B_3, ...B_n$ be a *partition* of the sample space (i.e., they are disjoint and their union is the entire sample space).

$$P(A) = P(A|B_1)P(B_1) + P(A|B_2)P(B_2) + \cdots + P(A|B_n)P(B_n)$$
$$P(A) = P(A \cap B_1) + P(A \cap B_2) + \cdots + P(A \cap B_n)$$

For **LOTP with extra conditioning**, just add in another event C!

$$P(A|C) = P(A|B_1, C)P(B_1|C) + \cdots + P(A|B_n, C)P(B_n|C)$$
$$P(A|C) = P(A \cap B_1|C) + P(A \cap B_2|C) + \cdots + P(A \cap B_n|C)$$

Special case of LOTP with B and B^c as partition:

$$P(A) = P(A|B)P(B) + P(A|B^c)P(B^c)$$
$$P(A) = P(A \cap B) + P(A \cap B^c)$$

Bayes' Rule

Bayes' Rule, and with extra conditioning (just add in C!)

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

$$P(A|B, C) = \frac{P(B|A, C)P(A|C)}{P(B|C)}$$

We can also write

$$P(A|B, C) = \frac{P(A, B, C)}{P(B, C)} = \frac{P(B, C|A)P(A)}{P(B, C)}$$

Odds Form of Bayes' Rule

$$\frac{P(A|B)}{P(A^c|B)} = \frac{P(B|A)}{P(B|A^c)} \frac{P(A)}{P(A^c)}$$

The *posterior odds* of A are the *likelihood ratio* times the *prior odds*.

Practice: What is $P(disease \mid +test)$ if P(disease) = 0.01, P(+ | disease) = 0.99, P(+ | no disease) = 0.01?

Expectation

f(X) probability distribution function of X

X ~ P : X is distributed according to P.

Expected value of f under P : $E_P[f(x)] = \sum_x p(x)f(x)$

E.g. unbiased coin. x = 1, 2, 3, 4, 5, 6. p(X=x) = 1/6 for all x.
$E(X) = \sum_x p(x) \cdot x = (1/6) \cdot [1 + 2 + 3 + 4 + 5 + 6] = 3.5$

Entropy

Always greater than or equal to 0. Zero when outcome is certain. 1 for uniform distribution.
Entropy is based on a pdf, not a list of labels. E.g.
H[1,1,0] → H[2/2, 1/3].
$X \sim P$, $x \in \Omega$

First define **Surprise**: $S(x) = -\log_2 p(x)$
$S(X = \text{heads}) = -\log_2(1/2) = 1$.

Axiom 1 : S(1) = 0. (If an event with probability 1 occurs, it is not surprising at all.)

Axiom 2 : S(q) > S(p) if q < p. (When more unlikely outcomes occur, it is more surprising.)

Axiom 3 : S(p) is a continuous function of p. (If an outcomes probability changes by a tiny amount, the corresponding surprise should not change by a big amount.)

Axiom 4 : S(pq) = S(p) + S(q). (Surprise is additive for independent outcomes.)

Surprise of 7 = pretty surprised. Probability of $1/2^7$ of happening (Shannon) **Entropy**:

$$\begin{aligned} H[X] &= -\sum_x p(x) \cdot \log_2 p(x) \\ &= -\sum_x p(x) S(x) \\ &= E[S(x)] \end{aligned}$$

The entropy is the expectation of the surprise. Throw out x for $p(x) = 0$ because log(0) is ∞.

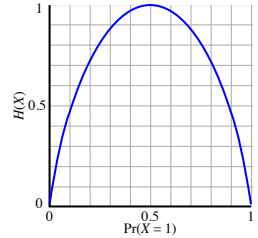
Binary Entropy Function: $p(X = 1) = \theta$ and $p(X = 0) = 1 - \theta$

$$\begin{aligned} H(X) &= -[p(X = 1) \log_2 p(X = 1) + p(X = 0) \log_2 p(X = 0)] \\ &= -[\theta \log_2 \theta + (1 - \theta) \log_2 (1 - \theta)] \end{aligned}$$

Entropy of an unbiased coin flip:
X is a coin flip. $P(X = \text{heads}) = 1/2$, $P(X = \text{tails}) = 1/2$
Note: $\log_2(1/2) = -1$, $-\log_2(1/2) = \log_2(2) = 1$
$H[X] = -[1/2 \log_2(1/2) + 1/2 \log_2(1/2)] = 1$

Entropy of a coin that always flips to heads:
$P(X = \text{heads}) = 1$, $P(X = \text{tails}) = 0$
Note: $\log_x(0) = 0$
$H[X] = -[1 \log_2(1) + 0] = 0$
No surprise: you are sure what you are going to get.

Binary entropy plot.



Canonical example:

X	Y
0	1
1	0
1	1

If you want to estimate entropy of X, you can use P(X=0).

$$H[X] = -[\frac{1}{3}\log_2\frac{1}{3} + \frac{2}{3}\log_2\frac{2}{3}]$$
$$= \frac{1}{3}\log_2 3 + \frac{2}{3}\log_2 3 - \frac{2}{3}\log_2 2$$
$$= \log_2 3 - \frac{2}{3} \approx 0.91$$

This time H[X] = H[Y] because of symmetry.
The discrete distribution with maximum entropy is the uniform distribution. For K values of X, $H(X) = \log_2 K$
Conversely, the distribution with minimum entropy (which is zero) is any delta-function that puts all its mass on one state. Such a distribution has no uncertainty.

Conditional Entropy

If you don't know x: (this is kind of an average).
$H[Y \mid X = x] = -\sum_y P(Y = y \mid X = x) \cdot \log_2 P(y \mid X = x)$
$H[Y \mid X = x] = E[S(Y \mid X = x)]$
Note that we are summing over y because we are specifying x.

For a particular value of X:
$H[Y \mid X] = \sum_x p(x) H[Y \mid X = x]$

Back to table above:

$$H[Y \mid X = 0] = ?$$
look only at X=0 in table.
$$= -[0 + 1\log_2]$$

Now that you know X=0, entropy goes to 0.

$H[Y \mid X = 1] = 1$: You know *less* if you know X=1.

Now use $H[Y \mid X] = \frac{1}{3}(0) + \frac{2}{3}(1) = 2/3$
Given X, you know more. Average our the more certain case and the less certain case.

Note: $H[Y \mid X] \leq H[Y]$: knowing something can't make you know less.

Entropy and Information Gain

Information Gain - $IG(X) = H(Y) - H(Y \mid X)$
Y is the node on top. X are the nodes below. He might have used lower case.
Class Example: If X_1 is a node for a split, and you want to know the information gain for that node, you:

- calculate entropy of the split. Find Entropy of each branch of the split, and the fraction of points that were channeled to each split. E.g.

$$\{T,T,T,T,T,F\} \rightarrow \{T,T,T,T\} \text{ (for } X_1 = T), \{T,F\} \text{ (for } X_1 = F)$$
$$\rightarrow P(X_1 = T) = 4/6, P(X_1 = F) = 2/6$$
$$\rightarrow H(X_1 = T) = (1 * \log_2 1 + 0 * \log_2 0) = 0$$
$$\rightarrow H(X_1 = F) = \frac{2}{6}(\frac{1}{2}\log_2\frac{1}{2} + \frac{1}{2}\log_2\frac{1}{2})$$
$$= 1 \text{ (uniform distribution)}$$
$$H(Y|X_1) = -\frac{4}{6}(1 * \log_2 1 + 0 * \log_2 0) - \frac{2}{6}(\frac{1}{2}\log_2\frac{1}{2} + \frac{1}{2}\log_2\frac{1}{2})$$
$$= 2/6$$

- find the entropy of the unsplit data:
  $\{T,T,T,T,T,F\} \rightarrow -(5/6)\log_2(5/6) - (1/6)\log_2(1/6) = 0.65$
- subtract the weighted average of the split entropies from the original: $IG(X_1) = H(Y) - H(Y|X_1) = 0.65 - 0.33$

- Low uncertainty ↔ Low entropy.
- Lowering entropy ↔ More information gain.

Bits

If you use log base 2 for entropy, the resulting units are called bits (short for binary digits).
How many things can you encode in 15 bits? 2^{25}.

Common notation

semicolon versus | in probabilities:
E.g. $P(X;\theta)$ vs $P(X|\theta)$
| is for random variables and ; is for parameters.
Andrew Ng verbalizes the semicolon as "parameterized by." So $f(x;\theta)$ would be spoken as "f of x parameterized by theta"

Decision Trees

Summary:

- One of the most popular ML tools. Easy to understand, implement, and use. Computationally cheap (to solve heurisRcally).
- Uses informaton gain to select attributes (ID3, C4.5,)
- Presented for classification, but can be used for regression and density estimation too
- Decision trees will overfit!!!
- Must use tricks to find simple trees, e.g., (a) Fixed depth/Early stopping, (b) Pruning, (c) Hypothesis testing
- Tree-based methods partition the feature space into a set of rectangles.
- Interpretability is a key advantage of the recursive binary tree.

Pros:

- easy to explain to people
- more closely mirror human decision-making than do the regression and classification approaches
- can be displayed graphically, and are easily interpreted even by a non-expert
- can easily handle qualitative predictors without the need to create dummy variables

Cons:

- trees generally do not have the same level of predictive accuracy as some of the other regression and classification approaches
- can be very non-robust. A small change in the data can cause a large change in the final estimated tree

Vocab:

- **classification tree** - used to predict a qualitative response rather than a quantitative one
- **regression tree** - predicts a quantitative (continuous) variable.
- **depth of tree** -the maximum number of queries that can happen before a leaf is reached and a result obtained
- **split** -
- **node** - synonymous with split. A place where you split the data.
- **node purity** -
- **univariate split** - A split is called univariate if it uses only a single variable, otherwise multivariate.
- **multivariate decision tree** - can split on things like A + B or Petal.Width / Petal.Length ¡ 1. If the multivariate split is a conjunction of univariate splits (e.g. A and B), you probably want to put that in the tree structure instead.
- **univariate decision tree** - a tree with all univariate splits/nodes. E.g. only split on one attribute at a time.
- **binary decision tree**
- **argmax** - the input that leads to the maximum output
- **greedy** - at each step of the tree-building process, the best split is made at that particular step, rather than looking ahead and picking a split that will lead to a better tree in some future step.
- **threshold splits** -
- **random forest**: an ensemble of decision trees which will output a prediction value. Each decision tree is constructed by using a random subset of the training data.

Protocol:

1. Start from empty decision tree
2. Split on next best attribute (feature).
   - Use something like information gain to select next attribute. $\arg\max_i IG(X_i) = \arg\max_i H(Y) - H(Y|X_i)$
3. Recurse

When do we stop decision trees?

- Dont split a node if all matching records have the same output value
- Only split if all of your bins will have data in them. His words: "none of the attributes can create multiple nonempty children." He also said "no attributes can distinguish", and showed that for the remaining training data, each category only had data for one label. And third, "If all records have exactly the same set of input attributes then dont recurse"

He noted that you might not want to stop splitting just because all of your information nodes have zero information gain. You would miss out on things like XOR.
Decision trees will overfit. If your labels have no noise, the training set error is always zero. To prevent overfitting, we must introduce some bias towards simpler trees. Methods available:

- Many strategies for picking simpler trees
- Fixed depth
- Fixed number of leaves
- Or something smarter

One definition of overfitting: If your data is generated from a distribution $D(X,Y)$ and you have a hypothesis space H:
Define errors for hypothesis $h \in H$: training error = $error_{train}(h)$, Data (true) error = $error_D(h)$. The hypothesis h overfits the training data if there exists an h' such that $error_{train}(h) < error_{train}(h')$ and $error_D(h) > error_{train}(D)$. In plain english, if there is an alternative hypothesis that gives you more error on the training data but less error in the test data then you have overfit your data.

How to Build Small Trees
Two reasonable approaches:

- Optimize on the held-out (development) set. If growing the tree larger hurts performance, then stop growing. But this requires a larger amount of data
- Use statistical significance testing. Test if the improvement for any split it likely due to noise. If so, dont do the split. Chi Square test w/ MaxPchance = something like 0.05.

Pruning Trees
Start at the bottom, not the top. The top is most likely to have your best splits. In this way, you only cut high branches if all the branches below were cut.
Don't use the validation set for pruning. **Your code should never use the validation set.** The validation set is for **you** to learn from; the code will always learn from the training set.
Classification vs. Regression Trees
In class we mostly discussed nodes with categorical attributes. You can have continuous attributes (see HW1). You can also have either discrete or continuous output. When output is discrete, you can chose your splits based on entropy. If it is continuous, you need to do something more like least squares. For regression trees, see pg 306 from ISL or pg 307 of ESLII.
For discrete data:
"For discrete data, you can't split twice on the same feature. Once you've moved down a branch, you know that all data in that branch has the same value for the splitting feature."
For continuous data:
More computationally expensive than discrete data. Often can try to change continuos data to categorical. Might lose some smoothness for real numbers, but might be worth it
K-fold validation versus using a held-out data set:
If you have enough data to pull out a held-out set, that is preferable to K-fold validation.

Maximum Likelihood & Maximum a Posteriori

(Also see paragraph at the end of this PDF near vocab for a MLE/MAP comparison.)
Vocab

- **likelihood**: the probability of the data given a parameter. E.g. $P(D|\theta)$ (for discrete like Binomial). Need not a pdf; need not be normalized.
- **log-likelihood**: lower-case: $l(\theta|x) = \log L(\theta|x)$
- **maximum likelihood** (ML):
- **MLE**: Maximum Likelihood Estimation.
- **PAC**: Probability Approximately Correct.
- **Posterior**: the likelihood times the prior, normalized

MLE: Maximum Likelihood Estimation
Choose θ to maximize probability of D.
Set derivative of __ to zero and solve. If function is multivariate, set each partial derivative to zero and solve.
$\hat{\theta} = \arg\max_\theta P(D|\theta) = \arg\max_\theta \ln P(D|\theta)$
Note we are using ln, not $\log_2$ as we did for entropy above. Want it to cancel exponents now.

Binomial Distribution
Assumes i.i.d: $D = \{x_i | i = 1 \dots n\}, P(D|\theta) = \prod_i P(x_i \mid \theta)$.
Likelihood function: $P(D|\theta) = \theta^{\alpha H}(1-\theta)^{\alpha T}$
P(heads) = θ, P(tails) = $1-\theta$

$$\begin{aligned}\hat{\theta} &= \arg\max_\theta \ln P(D|\theta) \\ &= \arg\max_\theta \ln \theta * \alpha H (1-\theta)^{\alpha T}\end{aligned}$$

Find optimal theta by setting the derivative to zero:

$$\begin{aligned}\frac{d}{d\theta}\ln P(D|\theta) &= \frac{d}{d\theta}\ln \theta * \alpha H(1-\theta)^{\alpha T} \\ &= \arg\max_\theta \ln \theta * \alpha H(1-\theta)^{\alpha T} \\ &= \dots = \frac{\alpha_H}{\alpha_H + \alpha_T}\end{aligned}$$

For Binomial, there is exponential decay in uncertainty with # of observations. You can also find the probability that you are approximately correct (see notes).
$P(|\widehat{\theta} = \theta * | \geq \epsilon) \leq 2e^{-2N\epsilon^2}$. Can calculate N (# of flips) to have error less than ϵ with probability of being incorrect δ. Your sensitivity depends on your problem; error on stock market data might cost billions.
What if you had prior beliefs? Use MAP instead of MLE.

Bayesian Learning

Inferring the probability of the parameters themselves, not the probability of the data. Whenever you see $P(\theta|D)$ you know that is some posterior distribution. That is a tidy way of representing your knowledge about θ and your uncertainty about that knowledge. (The uncertainty is held in the PDF; narrow = certain and flat = uncertain).

Rather than estimating a single θ, we obtain a distribution over possible values of θ.
For small sample size, prior is important!
Use Bayes' Rule: $P(\theta|D) = \frac{P(D|\theta)P(\theta)}{P(D)}$

- **Posterior**: $P(\theta|D)$. Note $P(\theta|D) \propto P(D|\theta)P(\theta)$
- **Data Likelihood**: $P(D|\theta)$
- **Prior**: $P(\theta)$
- **normalization**: $P(D)$. Just a constant so it doesn't matter. Hard to calculate anyway.

Or equivalently, $P(\theta|D) \propto P(D|\theta)P(\theta)$. **Always use this form**, not the one with $P(D)$ in the denominator.

Note: you are multiplying two PDFs here. When you plug in particular data, your two terms become numbers.

As you get more and more data, $P(\theta|D)$ grows more and more narrow. Like with more cannon ball holes, you are more certain about your angle θ.
Murphy 2012 pg 70: In general, when we have enough data, the posterior $p(h|D)$ becomes peaked on a single concept, namely the MAP estimate, i.e., $p(h|D) \to \delta_{\hat{h}MAP}(h)$ where $\delta_{\hat{h}MAP} = \arg\max_h p(h|D)$ is the posterior mode, and wehre δ is the Dirac measure defined by $\delta_x(A) = 1$ if $x \in A$ and $= 0$ if $x \notin A$

About the $P(D)$. It is the "marginal probability", which is basically the probability of D when you integrate out θ.

For uniform priors, MAP reduces to MLE objective. $P(\theta) \propto 1$ leads to $P(\theta|D) \propto P(D|theta)$

If you have a uniform prior, you just do MLE.
$P(\theta) \propto 1 \to P(\theta|D) \propto P(D|\theta)$

Note: if you have D first it is Likelihood, and if you have θ first it is the Posterior. ($P(D|\theta)$ $P(\theta|D)$).
Vocab

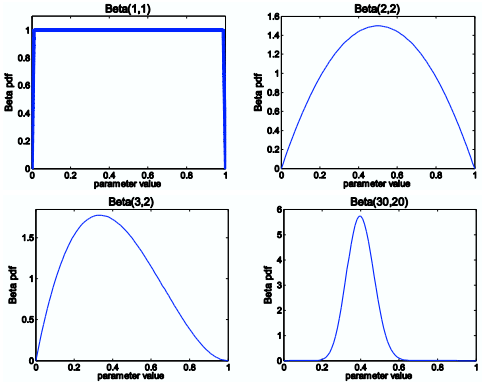
- **prior**:
- **prior distribution**: (same as "prior")
- **posterior**:
- **posterior distribution**: (same as "posterior")
- **Maximum likelihood**: Find the parameter that makes the probability highest. E.g. θ for coin toss. (A famous "point estimator")
- **MAP**: Maximum a posteriori (estimation). Maximize the posterior instead of the likelihood. Take the value that causes the highest point in the posterior distribution.
  Just take the peak of your posterior. Forget about the uncertainty. Pretty much like MLE, but you also have some influence of a prior.
- **conjugate**: If our posterior is a distribution that is of the same family as our prior, then we have conjugacy. We say that the prior is conjugate to the likelihood.
- **conjugate model**: great because we know the exact distribution of the posterior so we can easily simulate or derive quantities of interest analytically. In practice, we rarely have conjugacy.

Likelihood	Prior	Posterior
Binomial	Beta	Beta
Negative Binomial	Beta	Beta
Poisson	Gamma	Gamma
Geometric	Beta	Beta
Exponential	Gamma	Gamma
Normal (mean unknown)	Normal	Normal
Normal (variance unknown)	Inverse Gamma	Inverse Gamma
Normal (mean and variance unknown)	Normal/Gamma	Normal/Gamma
Multinomial	Dirichlet	Dirichlet

Thumbtack Problem, Bayesian style (MAP)
Start as usual with Bayes' without $P(D)$: $P(\theta|D) \propto P(D|\theta)P(\theta)$.
Define parameters: θ is the probability of one side up. α_H and α_T are the number of heads and tails tossed. β_H and β_T are the parameters of the prior. These "beta prior parameters" can be thought of as "fake counts" in the case of the beta distribution.

- use Binomial as the likelihood: $P(D|\theta) = \theta^{\alpha_H}(1-\theta)^{\alpha_T}$.
  Note: we are estimating θ after seeing α_H heads and α_T tails.
- the prior is $P(\theta) = \frac{\theta^{\beta_H}(1-\theta)^{\beta_T - 1}}{B(\beta_H, \beta_T)} \sim \text{Beta}(\beta_H, \beta_T)$.
  Now β_H and β_T are the number of heads and tails you expected to see. The B in the denominator is for the beta function (not same as beta distribution).
- To get a simple posterior form, use a conjugate prior.
  Conjugate prior of Binomial is the Beta Distribution. See slides for math: $P(\theta|D) \sim \text{Beta}(\beta_H + \alpha_H, \beta_T, \alpha_T)$
- note that there are similar terms in the prior and likelihood functions. Some will cancel out when you multiply them.
- $P(\theta|D) = \frac{\theta^{\beta_H + \alpha_H - 1} \cdot (1-\theta)^{\beta_T + \alpha_T - 1}}{B(\beta_H + \alpha_H, \beta_T + \alpha_T)} \sim$
  $\text{Beta}(\beta_H + \alpha_H, \beta_T + \alpha_T)$.
  So your probability is shaped by the number of heads/tails you expected to see (β_H, β_T) and the number of heads/tails you actually saw (α_H, α_T).
- Apply MAP: $\widehat{\theta} = \arg\max_\theta P(\theta|D) = \frac{\alpha_H + \beta_H - 1}{\alpha_H + \beta_H + \alpha_T + \beta_T - 2}$.
  Effectively, our prior is just adding $\beta_H - 1$ heads flips and $\beta_T - 1$ tail flips to the dataset.
- The Beta prior is equivalent to extra thumbtack flips. As $N \to \infty$, the prior is forgotten. But for small sample size, prior is important.

Beta Distribution



MAP (point) estimation:

1. Chose a distribution to fit the data to. Your choice determines the form of the likelihood ($P(\theta|D)$).
2. Chose a prior (distribution). Can use a table that shows conjugate priors for various distributions. Prior is over the parameters you are guessing.
3. Now you have a posterior (multiply prior by likelihood).
4. Plug in your particular data values under many values of θ to get the likelihood ($P(D|\theta)$). Recall the likelihood need not be a PDF (need not be normalized).
5. Pick the value that causes the highest point on the peak.

MAP estimation
Closely related to Fisher's method of maximum likelihood (ML), but employs an augmented optimization objective which incorporates a prior distribution over the quantity one wants to estimate. You get to pick the distribution to represent the prior. MAP estimation can therefore be seen as a regularization of ML estimation. (Another famous "point estimator")
Chosing between MLE and MAP:
Chose ML if you don't know enough about the domain to impose a new prior.
If you are measuring a continuous variable, Gaussians are your friend.

Gaussians

Properties of Gaussians:

- Affine transformation (multiplying by a scalar and adding a constant) are Gaussian. If X $\sim$ N(μ,σ^2) and Y = aX + b, then Y $\sim$ N($a\mu + b, a^2\sigma^2$)
- Sum of Gaussians is Gaussian. If X $\sim$ N(μ_X, σ_X^2), Y $\sim$ N(μ_Y, σ_Y^2), and Z = X+Y, then Z $\sim$ N($\mu_X + \mu_Y, \sigma_X^2 + \sigma_Y^2$)
- Easy to differentiate.

Learn a Gaussian: $P(x|\mu,\sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{-(x-\mu)^2}{2\sigma^2}}$.
MLE for Gaussian: Prob of i.i.d. samples D = $\{x_1, \ldots, x_N\}$:
$P(D|\mu,\sigma) = (\frac{1}{\sigma\sqrt{2\pi}})^N \prod_{i=1}^{N} e^{\frac{-(x_i-\mu)^2}{2\sigma^2}}$.
Note: it is not $P(\mu,\sigma|D)$, like I thought in class.
Find μ_{MLE}, $\sigma_{MLE} = \arg\max_{\mu,\sigma} P(D|\mu,\sigma)$.
Log-likelihood:

$$\ln P(D|\mu,\sigma) = \ln[\text{thing above}] = -N \ln \sigma\sqrt{2\pi} - \sum_{i=1}^{N} \frac{(x_i-\mu)^2}{2\sigma^2}.$$

Differentiate w.r.t. μ and set = 0. End up with $\widehat{\mu} = \frac{1}{N}\sum_{i=1}^{N} x_i$.

Differentiate w.r.t. σ and set = 0. End up with
$\widehat{\sigma}^2_{MLE} = \frac{1}{N}\sum_{i=1}^{N}(x_i - \widehat{\mu})^2$.
But actually, that leads to a biased estimate, so people actually use
$\widehat{\sigma}^2_{unbiased} = \frac{1}{N-1}\sum_{i=1}^{N}(x_i - \widehat{\mu})^2$
The conjugate priors: mean: use Gaussian prior:
$P(\mu|\nu,\lambda) = \frac{1}{\lambda\sqrt{2\pi}} e^{\frac{-(\mu-\nu)^2}{2\sigma^2}}$. (Instead of σ, use λ and replace the $(x-\mu)^2$ with $(\mu-\nu)^2$).
For variance: use Wishard Distribution:

Linear Regression

The loss is $||y - Xw||^2 + \lambda||w||^2$
(this corresponds to the sum of squared error loss + L2 regularization)

Ordinary Least Squares
Notation:

- x_i: an input data point. __ rows by __ columns.
- y_i: a predicted output
- $\widehat{y_i}$: a predicted output
- $\widehat{y}$:
- w_k: weight k
- $\boldsymbol{w}^*$: the vector of weights found in regression.
- $f_k(x_i)$
- t: what we want to regress against
- t_j: the output variable that you either have data for or are predicting.
- $t(\boldsymbol{x})$: Data. "Mapping from x to t(x)"
- H: $H = \{h_1, \ldots, h_K\}$. Basis functions. In the simplest case, they can just be the value of an input variable/feature or a constant (for bias).
- L_2: The L_2. Can appear as a loss function to describe the deviation from data or as a penalty.
- $||\widehat{w}||_1$: "L_1" penalty. The "Manhattan distance". Like traveling a, b in a pythagorean triangle. $\sum |x_i|$
- $||\widehat{w}||_2$. "L_2" penalty. Euclidean length of a vector. Like c in a pythagorean triangle. $\sqrt{\sum |x_i|^2}$

Vocab:

- **bias-variance tradeoff** - the problem of simultaneously minimizing two sources of error that prevent supervised learning algorithms from generalizing beyond their training set.
  - The bias is error from erroneous assumptions in the learning algorithm. High bias can cause an algorithm to miss the relevant relations between features and target outputs (underfitting).
  - The variance is error from sensitivity to small fluctuations in the training set. High variance can cause overfitting: modeling the random noise in the training data, rather than the intended outputs.
- **basis function**
- **bias** (parameter): like the intercept in a linear equation. The part that doesn't depend on the features.
- **bias** (learning bias): (?? "inductive bias" ??) -
- **hyperplane** - a plane, usually with more than 2 dimensions.
- **input variable** - a.k.a. feature. E.g. a column like CEO salary for rows of data corresponding to different companies.
- **response variable** - synonyms: "dependent variable", "regressand", "predicted variable", "measured variable", "explained variable", "experimental variable", "responding variable", "outcome variable", and "output variable". E.g. a predicted stock price.
- **regularization** - introducing additional information in order to solve an ill-posed problem or to prevent overfitting. E.g. applying a penalty for large parameters in the model.
- **ridge regression** -
- **vector norm**: put in a vector and get out a number like length or size. Real valued function of some sort of vector or matrix quantity.
- **hyperparameters**:
  λ, not w. Parameters that control your actual parameters.

  In Bayesian analysis, the parameters that don't touch the data. Like the parameters for the prior on the prior. Called the ridge regression λ a hyperparameter, though this is a stretch in the terminology.
  Class version:
- **feature selection**: explicitly select features that can go into your model instead of throwing all features in.
- **loss function**: A function that maps an event or values of one or more variables onto a real number intuitively representing some "cost" associated with the event. An optimization problem seeks to minimize a loss function.
  $\sum_j (t(x_j) - \sum w_i h(x_i))^2$. (least-squares error ($L_2$)) (??? = training error??)
- **training set error**: *doesn't include the regularization penalty!*. A.k.a. "training error". Sum of squares error divided by the number of points. See formula later.

Ordinary Least Squares

$$\text{total error} = \sum_i (y_i - \hat{y_i})^2 = \sum_i (y_i - \sum_k w_k f_k(x_i))^2$$

i is for each data point, k is for each of the k basis functions.

Under the additional assumption that the errors be normally distributed, OLS is the maximum likelihood estimator.
?? Use words to describe what subset of regression in general this is. What is ordinary? What are we limiting?

The regression problem:
Given basis functions $\{h_1, \ldots, h_K\}$ with $h_i(\mathbf{x}) \in \mathbb{R}$,
find coefficients $\boldsymbol{w} = \{w_1, \ldots, w_k\}$.
$t(\boldsymbol{x}) \approx \widehat{f}(\boldsymbol{x}) = \sum_i w_i h_i(\boldsymbol{x})$

This is called linear regression b/c it is linear in the parameters. We can still fit to nonlinear functions by using nonlinear basis functions; $f_k \to h_i$ Minimize the **residual squared error**:
$\boldsymbol{w}* = \arg\min_{\boldsymbol{w}} \sum_j (t(\boldsymbol{x}_j) - \sum_i w_i h_i(\boldsymbol{x}_j))^2$
j for each data point, i for the number of weights, which is the number of basis functions.

For fitting a line in 2D space, your basis functions are $\{h_1(x) = x, h_2(x) = 1\}$. $h_2(x) = 1$ is the (constant) bias basis function. The size of w_2 controls the effect in prediction.

To fit a parabola, your basis functions could be
$\{h_1(x) = x^2, h_2(x) = x, h_3(x) = 1\}$.
Want a 2D parabola? Use
$\{h_1(x) = x_1^2, h_2(x) = x_2^2, h_3(x) = x_1 x_2, \dots\}$.
Can define any basis functions $h_i(\boldsymbol{x})$ for n-dimensional input
$\boldsymbol{x} =< x_1, \dots, x_n >$

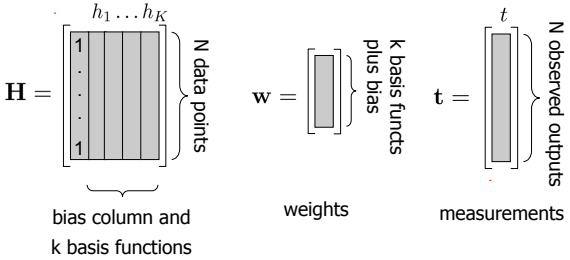
Linear in feature space vs linear in the parameter space:
When we want to fit a parabola, we are still using a function that is linear in parameter space. The weight matrix is still all constants. When we use nonlinear basis functions, we are nonlinear in feature space. The basis functions may be transformations of the input parameters.

Regression: matrix notation

$$\boldsymbol{w}* = \arg\min_w \sum_j (t(\boldsymbol{x}_j - \sum_i w_i h_i(\boldsymbol{x}_j))^2$$

$$\boldsymbol{w}* = \arg\min_w (\boldsymbol{Hw} - \boldsymbol{t})^T (\boldsymbol{Hw} - \boldsymbol{t})$$

$(\boldsymbol{Hw} - \boldsymbol{t})^T (\boldsymbol{Hw} - \boldsymbol{t})$ is the residual error.



Regression: closed form solution

$$\boldsymbol{w}^* = \arg\min_w (\boldsymbol{Hw} - \boldsymbol{t})^T (\boldsymbol{Hw} - \boldsymbol{t})$$
$$\boldsymbol{F}(\boldsymbol{w}) = \arg\min_w (\boldsymbol{Hw} - \boldsymbol{t})^T (\boldsymbol{Hw} - \boldsymbol{t})$$
$$\nabla_{\boldsymbol{w}} \boldsymbol{F}(\boldsymbol{w}) = 0$$
$$2\boldsymbol{H}^T(\boldsymbol{Hw} - \boldsymbol{t}) = 0$$
$$(\boldsymbol{H}^T\boldsymbol{Hw}) - \boldsymbol{H}^T\boldsymbol{t} = 0$$
$$\boldsymbol{w}^* = (\boldsymbol{H}^T\boldsymbol{H})^{-1}\boldsymbol{H}^T\boldsymbol{t}$$

Regression solution: simple matrix math

$$\mathbf{w}^* = \arg\min_{\mathbf{w}} \underbrace{(\mathbf{Hw} - \mathbf{t})^T(\mathbf{Hw} - \mathbf{t})}_{\text{residual error}}$$

$$\text{solution: } \mathbf{w}^* = \underbrace{(\mathbf{H}^T\mathbf{H})^{-1}}_{\mathbf{A}^{-1}}\underbrace{\mathbf{H}^T\mathbf{t}}_{\mathbf{b}} = \mathbf{A}^{-1}\mathbf{b}$$

$$\text{where } \mathbf{A} = \mathbf{H}^T\mathbf{H} = \underbrace{\quad}_{\text{k×k matrix for k basis functions}} \quad \mathbf{b} = \mathbf{H}^T\mathbf{t} = \underbrace{\quad}_{\text{k×1 vector}}$$

Dimensions:

- $\boldsymbol{t}$: N-dimensional (N = # of input data points)
- $\boldsymbol{w}$:
- $\boldsymbol{H}$: k + 1 by N. N is # of rows.

Linear regression prediction is a linear function plus Gaussian noise

Casual explanation:
If we assume that our data y_i is drawn from a linear function with some zero-mean Gaussian noise, i.e.

$$y_i = \sum_j w_j X_i j + \epsilon_i$$

(where ϵ_i is zero-mean gaussian noise drawn from Normal(0, σ)

Then we can show that the MLE estimates of w_j are exactly the optimal weights obtained by minimizing the SSE: $\sum_i (y_i - w^T X_i)^2$

(note that the variance σ actually doesn't matter for the derivation, you can just assume it's some positive number).

More formal explanation:
We can model as linear combination of basis functions + noise ϵ. It's safe to assume epsilon comes from Gaussian distribution.
$t(\boldsymbol{x}) = \sum_i w_i h_i(\boldsymbol{x}) + \epsilon$
Note: no μ because we set it to zero.

We can learn $\mathbf{w}$ using MLE: $P(t|x, w, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{-[t - \sum_i w_i h_i(x)]^2}{2\sigma^2}}$
Take the log and maximize with respect to w: (maximizing log-likelihood with respect to w)

$$\ln P(D|\boldsymbol{w}, \sigma) = \ln(\frac{1}{\sigma\sqrt{2\pi}})^N \prod_{j=1}^N e^{\frac{-[t_j - \sum_i w_i h_i(x_j)]^2}{2\sigma^2}}$$

Now find the w that maximizes this:
$\arg\max_w \ln(\frac{1}{\sigma\sqrt{2\pi}})^N + \sum_{j=1}^N \frac{-[t_j - \sum_i w_i h_i(x_j)]^2}{2\sigma^2}$
the first term isn't impacted by w so
$= \arg\max_w \sum_{j=1}^N \frac{-[t_j - \sum_i w_i h_i(x_j)]^2}{2\sigma^2}$
switch to $\arg\min_w$ when we divide by -1. The numerator is constant.:
$= \arg\min_w [t_j - \sum_i w_i h_i(x_j)]^2$
Least-squares Linear Regression is MLE for Gaussians!!!

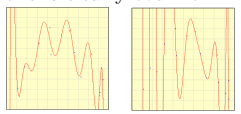
If you have a polynomial you are fitting, how many basis functions are there (??)

OLS Protocol (incomplete)

1. Chose your basis functions h_i. (Requires expertise). Can be nonlinear, e.g. $x_1^2, \sin(x)$, etc.
   - The number of parameters is len(H) + 1 (bias). E.g. for fitting a parabola (formula $y = ax^2 + bx + c$), you have 3 parameters: weights for basis functions x, x^2, and bias. Typically the # of basis functions is < the # of features.
2. Chose a regularization method so your weights don't get too big. (see below)
3. Plug them in to regression to get the weights (w_is). (Form the sum (residual squared error + regularization) that you want to minimize, then minimize.)
4. Make sure your weights aren't too big.

Regularization in Linear Regression

You need to regularize to prevent parameters from growing too large. Both of these were built from the same set of basis functions; the right one is clearly over-fit.



Ridge Regression

Ridge Regression is the most famous form of linear regression Here is our old "ordinary" least squares objective function:

$$\widehat{w} = \arg\min_w \sum_{j=1}^N [t(x_j) - (w_0 + \sum_{i=1}^k w_i h_i(x_j))]^2$$

It is the same as the previous ones but $i = 0$ is pulled out.
Now for ridge regression, we use that same notation.
And we add a penalty term that isn't applied to the bias feature:

$$\widehat{w}_{ridge} = \arg\min_w \sum_{j=1}^N [t(x_j) - (w_0 + \sum_{i=1}^k w_i h_i(x_j))]^2 + \lambda \sum_{i=1}^k w_i^2$$
$$= \arg\min_w (\boldsymbol{Hw} - \boldsymbol{t})^T (\boldsymbol{Hw} - \boldsymbol{t}) + \lambda \boldsymbol{w}^T I_{0+k} \boldsymbol{w}$$

$$I_{0+k} = \begin{bmatrix} 0 & 0 & 0 & \dots \\ 0 & 1 & 0 & \dots \\ 0 & 0 & 1 & \dots \\ \dots & \dots & \dots & \dots \end{bmatrix}$$ k+1 (k+1 columns)

k+1 x k+1 identity matrix, but with 0 in upper left

That I_{0+k} matrix is this:
Allows you to multiply the whole weight array without getting the bias term in there.
Note: $W^T W$ is w_i^2 or $||w_i||^2$
A similar derivation leads to a closed form solution:
$w^*_{ridge} = (\boldsymbol{H}^T\boldsymbol{H} + \lambda I_{0+k})^{-1}\boldsymbol{H}^T\boldsymbol{t}$
(Recall that un-regularized regression was $w^* = (\boldsymbol{H}^T\boldsymbol{H})^{-1}\boldsymbol{H}^T\boldsymbol{t}$).

How do you chose how large λ is?
* As $\lambda \to 0$, becomes same as MLE: unregularized. Large magnitudes of coefficients.
* As $\lambda \to \infty$, all weights become 0.

Experiment cycle

1. select a hypothesis f to best match the training set. (??) Is this the same as choosing your basis functions (?)
2. isolate a held-out data set if you have enough data, or do K-fold cross-validation if not enough data.
   - tune hyperparameters (λ) on the held-out set or via cross-validation. (Try many values of λ and chose the best one.)
   - You can use the same held-out data set each time if that set is big.
   - If doing K-fold, divide the data into k subsets. Repeatedly train on k-1 and test on the remaining one. Average the results.
   - find the w that minimizes the error. (Do so by taking the derivative and setting = 0); see ridge regression notes.
3. Select basis functions

Regularization options: Ridge & Lasso

Ridge:

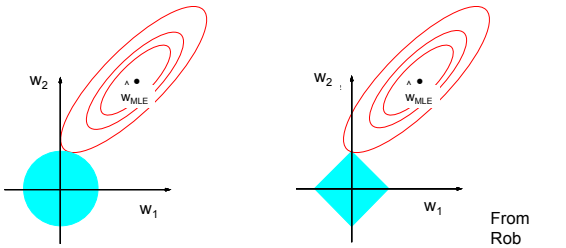
- $\widehat{w}_{ridge} = \arg\min_w \sum_{j=1}^{N} [t(x_j) - (w_0 + \sum_{i=1}^{k} w_i h_i(x_j))]^2 + \lambda \sum_{i=1}^{k} w_i^2$
- L_2 penalty. ("L_2 norm of $\boldsymbol{w}$). Large distances get penalized more. $Y = x^2$: don't want errors to cancel each other; differentiable.

Lasso:

- $\widehat{w}_{ridge} = \arg\min_w \sum_{j=1}^{N} [t(x_j) - (w_0 + \sum_{i=1}^{k} w_i h_i(x_j))]^2 + \lambda \sum_{i=1}^{k} |w_i|$
- L1 penalty: linear penalty pushes more weights to zero. Allows for a type of feature selection. But it is not differentiable and there is no closed form solution.
- L1 is absolute value: don't need to square it.
- Lasso may be more useful when you have too many features for the amount of data you get. Example: 100k parameters about companies to predict stock prices but only 100 data points. Could tune lambda until you have about 100 nonzero weights.

Your chose of penalty has huge effects on the algorithm!
Sometimes L_1 will do better than L_2 and vice versa, but usually L_2 is more powerful than L_1.
Would be better to use a combination of the penalties than to first reduce the number of features with L_2 before applying L_1.
You can generate a lot of basis functions and use L_1 to chose the good ones.

Geometric Intuition



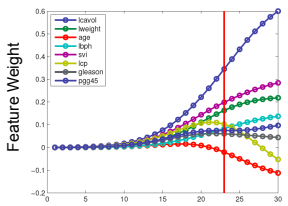
Ridge Regression Lasso

From Rob Tibshirani slides

This figure shows:

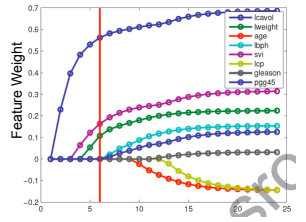
- The contour lines represent the maximum likelihood of the vector of weights. All points on the contour have equal likelihood.
- The two axes represent different parameters for two of the weights. (regression coefficients)
- Circles are characteristic of ridge regression (with L2 penalty): Penalty = the magnitude of the vector.
- Shapes that are pointy on the axes are characteristic of Lasso (with L1 penalty): the vector components get added.
- Where the likelihood function touches in this w_1, w_2 space represents the coefficients of the weights. For Ridge Regression, we see that small but nonzero values of the coefficients can be obtained. For Lasso Regression, the curves are most likely to touch the diamond on the axes, resulting in coefficients that are truly zero.

Ridge Coefficent Path



Larger λ ← $||\hat{w}||_2$ → Smaller λ

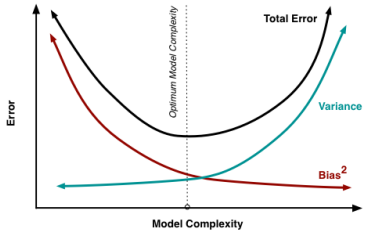
LASSO Coefficent Path



Larger λ ← $||\hat{w}||_1$ → Smaller λ

Don't compare coefficient magnitudes at given λs, but do note that for Ridge the gradually come away from the zero axis and in Lasso they are zero until they pop out.

Bias-Variance Tradeoff



Your choice of hypothesis class (e.g. degree of polynomial) introduces learning bias.
The more complex the model, the more the ____ set accuracy goes down
A more complex class → less bias and more variance.

From https://www.youtube.com/watch?v=Rm6s6gmLTdg: hfill
High bias = inability to represent the true function in the class of functions we are willing to tolerate. Pulls us to a particular function or class of functions regardless of the data. Can be from looking only at a specific type of function, or by having strong regularization
High variance = extremely dependent on the exact data they were trained on. Might fit the data well, but may do poorly on a different data self.
From Wikipedia:
Ideally, one wants to choose a model that both accurately captures the regularities in its training data, but also generalizes well to unseen data. Unfortunately, it is typically impossible to do both simultaneously. High-variance learning methods may be able to represent their training set well, but are at risk of overfitting to noisy or unrepresentative training data. In contrast, algorithms with high bias typically produce simpler models that don't tend to overfit, but may underfit their training data, failing to capture important regularities.

Models with low bias are usually more complex (e.g. higher-order regression polynomials), enabling them to represent the training set more accurately. In the process, however, they may also represent a large noise component in the training set, making their predictions less accurate - despite their added complexity. In contrast, models with higher bias tend to be relatively simple (low-order or even linear regression polynomials), but may produce lower variance predictions when applied beyond the training set.

Adding features (predictors) tends to decrease bias, at the expense of introducing additional variance.

Solutions:

- Dimensionality reduction and feature selection can decrease variance by simplifying models.
- Similarly, a larger training set tends to decrease variance.
- Use tunable modeling parameters such as applying more significant regularization.

If held-out is too small, we can end up over-fitting.

Error Definitions

True ("Prediction") Error:
Since the training set error can be a poor measure of the "quality" of the solution, we can use prediction error ("true error"). The error over all possibilities. Instead of sum, take expectation.

$$error_{true}(\boldsymbol{w}) = E_X[(t(\boldsymbol{x_j}) - \sum_i w_i h_i(\boldsymbol{x_j}))^2]$$

Gold Standard:

$$error_{true}(\boldsymbol{w}) = \int_x (t(\boldsymbol{x_j}) - \sum_i w_i h_i(\boldsymbol{x_j}))^2 p(\boldsymbol{x}) d\boldsymbol{x}$$

$p(x)$ is assumed to be the "true" distribution of the data, which we don't actually know.
How to get $p(\boldsymbol{x})$? Need to know the true distribution of the data (?). You almost never know how to compute $p(x)$. And, the integral is a very big sum.

To do this, you need to split your data into training and test set. If we have a dataset which is randomly sampled from $p(x)$, which we assume our data is, we can estimate the true error using the average of the samples. Since we train on our training set, the error might be a bad estimate of the true error (we're biased to decrease the error), so we use a separate hold-out testing set to approximate the true error in instead.
Sample a set of i.i.d. points $\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_M\}$ from $p(x)$.
Approximate the integral with the sample average:

$$error_{true}(\boldsymbol{w}) \approx \frac{1}{M} \sum_{j=1}^{M} \left(t(\boldsymbol{x}_j - \sum_i w_i h_i(\boldsymbol{x}_j) \right)$$

That is the sampling approximation of the predicted error:
That leads to a fair approximation.

The true prediction error is the expectation over **future** test cases you don't have. Since you don't have the x values, you go to probability. Pick a point from the distribution, and calculate the _____.

Don't use the training data to predict true error; you've already trained to that data! You would have too optimistic of a prediction for true error.
Prediction error is high when the model is too simple and too complex, unlike training set error which only penalizes too simple.

Training Set Error: **optimistically biased** (a.k.a. training error)

$$error_{train}(\boldsymbol{w}) = \frac{1}{N_{train}} \sum_{j=1}^{N_{train}} (t(\boldsymbol{x_j}) - \sum_i w_i h_i(\boldsymbol{x_j}))^2$$

Decreases exponentially with model complexity.
Training error is a poor prediction of prediction error!
You expect to see training error to decrease with complexity, but that doesn't mean you have a good solution!

Test Error: (our final measure)

Uses the same formula as prediction error, except that we have never observed the test data. See formula below. We expect the true error to be smile shaped if the x-axis is model-complexity.

Testing is for the user of your algorithm. The user doesn't care about the values in your model. They just care about how well it works. They dont' care about the value of your hyperparameters.

Effects of λ value on model

- high lambda → simple model → lots of zeros → high error.
- with lambda = 0 → converges ridge regression to regular regression.
- with enough traiing data and lambda = 0, we expect overfitting → small training error.

Choosing λ

How to find lambdas?

- try a bunch and find which does best on the held out data set. Can't touch test data, even w/ stick so use held-out set.
- Do k-fold cross validation to pick the lambda that gives minimum error.
  Average over the loss curves Loss($fold_i$, λ)
  Minimum error = lowest loss on the hold-out set (the curve should look U shaped, so you just find the bottom of the U).
  May not find the bottom-of-the-U shape lambda.
  But it is ok to be off a bit; the red U might be pretty flat at the bottoms.

This use of the held-out data doesn't count as training. Lambda is fixed each time, and we train separately. Training on training data, just watching the number on the held-out set.
The model's parameters are not determined by the held-out data

How do you chose the range of lambda? (practical solution)

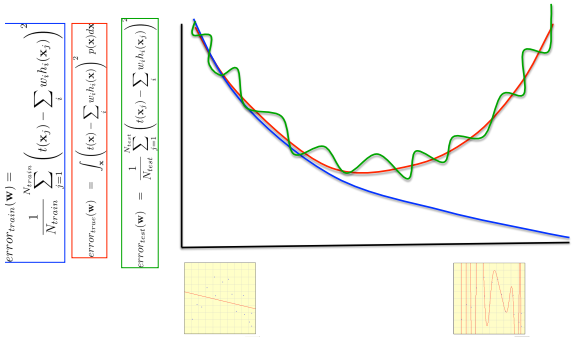
- You are limited to a grid search over values of lambda.
- You need a strategy for sweeping that space. Could do a "binary search", or something like a gradient search: take a step until we screw it up, and step back smaller. If it gets worse, you take a smaller step.
- From your loss, take the value of your loss (compute it). Loss = $(t(x) - \sum w_i h(x_i))^2$ **"You should be able to find loss given w** Can compute norm of W, too. If your loss is on the order of 1000 and your norm is on the order of 1, you can use these for defining search space. Firs try lamba = 1, 10, 100, 1000. Find the best from there then explore around there. If 100 was good, try 200, etc. You just want to know in which order the loss function is going to appear.

How to handle error calculations:

Given a dataset, randomly split it into two parts:
* training data: $\{\boldsymbol{x_1}, \ldots, \boldsymbol{x_{N_train}}\}$
* test data: $\{\boldsymbol{x_1}, \ldots, \boldsymbol{x_{N_test}}\}$
Use the training data to optimize parameters $\boldsymbol{w}$.
To calculate the test set error, you use the final solution $\boldsymbol{w^*}$ and calculate

$$error_{test}(\boldsymbol{w}) \approx \frac{1}{N_{test}} \sum_{j=1}^{N_{test}} (t(\boldsymbol{x_j} - \sum_i w_i h_i(\boldsymbol{x_j}))^2$$

Test set error as a function of model complexity

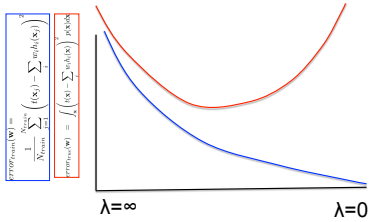


(blue is train, red is true, green is test)

overfitting

Assume:
* Data generated from distribution D(X,Y).
* A hypothesis space H
Define:
* Training error: $error_{train}(h)$
* Data (true) error: $error_{true}(h)$
We say h **overfits** the training data if there exists an $h' \in H$ such that:
* $error_{train}(h) < error_{train}(h)'$ and $error_{true}(h) > error_{true}(h)'$

Error as function of regularization parameter, fixed model complexity



Lambda (the regularization parameter) is fixed. Blue = training set error. Leads to overfitting/too-large-parameters when λ –¿ 0. Red = true error, which likes a happy medium λ.

Warning: your test set is only unbiased if you never ever ever do **any** learning on the data. This includes using the test data to select for the degree of the polynomial you fit.

(Recall, you can create a held-out/validation set from your training data or do k-fold validation.)

The height of the true error (red curve) is the "bias".

What you need to know

Regression:

- Basis functions = features
- Optimizing a sum of squared errors
- Relationship between regression and Gaussians

Regularization

- Ridge regression math
- LASSO formulation
- How to set lambda

Bias-Variance trade-off

Naive Bayes

Parameters are from data statistics; probabilistic interpretation.
Train on Y conditioned on x.
Bayes: $P(Y = X) \propto P(X|Y)P(Y)$. But that is (2^N) parameters, and you might not be able to make a model with that many (N) parameters if your training data has a lot of features. So you can add some assumptions that take you from 2^N parameters down to $2N$ of them.
Study the problem feature by feature. Assume chunks of features are conditionally independent. This is a false assumption, but often works. (Features are probably not independent, but it turns out to be reasonably safe to assume so.) MAP is the foundation for Naive Bayes classifiers.

Do we do optimization with NB? No. We start with the joint distribution: $P(x|y) * prior, \ldots,$ take the max. Not ML optimizing. Not required to optimize every time we learn something.

Vocabulary

- $h_{NB}(x)$ the function that returns the best class.
- Loss function: in Naive Bayes the loss function is the negative log likelihood of the data given the parameters: $-\ln(P(X, y|w))$.

Advantages:

- Fast to train (single scan/pass through data). Fast to classify
- Not sensitive to irrelevant features
- Handles real and discrete data
- Handles streaming data well
- conditional independence assumption → we don't need to see occurrences of joint assignments to estimate their probabilities.

Disadvantages:

- Assumes independence of features
- All observations have equal weight in prediction.

Vocab:

- **prior**: $P(Y)$, the probability of a label/class.
- **Likelihood**: $P(\boldsymbol{X}|Y)$.

Conditional independence

X is conditionally independent of Y given Z if the probability distribution for X is independent of the value of Y, given the value of Z:
$(\forall i, j, k)\ P(X = i|Y = j|Z = k) = P(X = i|Z = k)$.
E.g. P(Thunder | Rain, Lightening) = P(Thunder | Lightning)
Equivalent to P(X, Y | Z) = P(X | Z) P(Y | Z)

TODO: put in plain english.

HW 2 forum: Naive Bayes assumption (is): $P(D1, D2|H) = P(D1|H)P(D2|H)$, everything else follows from the rules of probability.

Naive Bayes Assumption

Features are independent given the class:

$$P(X_1, X_2|Y) = P(X_1|X_2, Y)P(X_2|Y)$$
$$= P(X_1|Y)P(X_2|Y)$$

more generally:

$$P(X_1, \ldots, X_n|Y) = \prod_i P(X_i|Y)$$

This reduces the number of parameters a lot! Say you had 5 features. Before this assumption, each of your 5 features could be dependent. Then you have to assign a probability to each state. If each is binary, then you can say 2^5. After this assumption, you would just have 5 parameters.

Homework clarifications

Homework 2 TA notes:
Naive Bayes gives us a way to compute the whole joint distribution P(Cold, Headache, Cough, SoreThroat). Once we have this, everything else follows from the sum/chain rules of probability:
From the definition of conditional probability (which is the chain rule in disguise): P(Cold — notHeadache, Cough, SoreThroat) = P(Cold, not Headache, Cough, SoreThroat)/P(not Headache, Cough, Sore Throat)
From the sum rule we see that the denominator is: P(not Headache, Cough, SoreThroat) = P(Cold, not Headache, Cough, SoreThroat) + P(not Cold, not Headache, Cough, SoreThroat)
Thus the entire conditional probability can be computed using the joint distribution (which can be computed from the factorization and the estimates from the dataset).
The reason we want to include the denominator is to make sure that P(not Headache, Cough, SoreThroat) is non-zero (as asked in another post). It's not zero here, but if it was that would mean we can't really ask about the prediction P(Cold — not Headache, Cough, SoreThroat), since this is only defined when P(not Headache, Cough, SoreThroat) is non-zero. In practice people just assume this isn't an issue, and directly maximize the joint probability.

The point of Naive Bayes isn't that $P(H|D1, D2)$ is proportional to $P(H, D1, D2)$, as this is true for any distribution (Here Di are the data, H is the hypothesis for some arbitrary problem). The point is that we assume the conditional independence $P(D1, D2|H)$ is $P(D1|H)P(D2|H)$. No the denominator is not used in practice, but that's a simplifying assumption that only works if none of your conditional probabilities are zero (which in turn means all joint probabilities are nonzero). Using the denominator doesn't defeat the point of Naive Bayes, it just ensures that $P(Cold|notHeadache, Cough, SoreThroat)$ is well defined as explained above.
You should be using the conditional independence assumption from Naive Bayes to compute the joint probabilities in the numerator/denominator: e.g.
$P(H, D1, D2) = P(H) * P(D1|H) * P(D2|H)$, where $P(H)$, $P(D1|H)$, $P(D2|H)$ are estimated from the data (with or without Laplacian smoothing). The conditional independence assumption means that we don't estimate $P(D1 = True, D2 = False|H)$ from the number of times we see $(D1 = True, D2 = False)$ together, we split $P(D1 = True, D2 = False|H)$ into $P(D1 = True|H) * P(D2 = False|H)$ and estimate each conditional independently. This is a huge simplification, since if we have N binary variables, we would need to see 2^N combinations to see all of the joint assignments (D1,...DN).

[Cold] is not independent of [Headache], [Cough], or [SoreThroat]. Consider: if [Cold] were in fact not related to these variables, we wouldn't be predicting [Cold] from them. Furthermore, [Cough] and [Headache] are not independent, since both are not independent of [Cold]. However, the Naive Bayes assumption is that once you account for that dependence, [Cough] and [Headache] are independent.
Here's a common-sense description of this. If you go around talking to people, you notice that some have headaches, and some have colds. At first, you are confused and wonder why this might be. Is it, perhaps, that coughing all the time is so annoying that it eventually causes people headaches? But then you realize that there is a common illness, the cold, that causes both headaches and coughs. Aha! you say. So you start asking people not only if they have a headache and a cough, but also whether or not they have a cold. You find that among people with colds, having a headache and having a cough seem like unrelated phenomena. And among people without colds, headaches and colds are again unrelated. This is the world that Naive Bayes assumes.
You ask "Would it be the case then that P(Cd, H, C, S) = P(H, C, S — Cd) P(Cd) ?" Yes. That fact is the definition of conditional probability, and it is a basic fact of probabilities. It is true for all variables, in all situations, no matter what assumptions you make. It is true whether Cd and H are independent or not, whether they represent colds or aliens or whatever. You can always use this fact about any variables.
————-
Naive Bayes is not about ignoring the denominator. This seems to have been a common misunderstanding.
Naive Bayes is an independence assumption; namely, that your input features are independent given the output feature.
This assumption allows you to estimate probabilities like P(X, Y, Z) in terms of other probabilities P(X, Y) and P(X, Z). Since it relates probabilities to each other, it makes it easier to estimate probabilities from samples. For example, if you are trying to detect if a message is spam, you want to compute P(spam — Hello, I, am, Prince, Albert, of, Nigeria, ...), and you may have never seen any points with text (Hello, I, am, Prince, Albert, of, Nigeria, ...) and so would not have any estimate of the probability that that message is spam. With Naive Bayes, you would rewrite this to in terms of the probabilities P(spam — Hello) and P(spam — I) and P(spam — am) and ..., all of which you can compute because you have seen many messages with the words "Hello", or "Prince", or "Nigeria", and know the associated probabilities of spam.
This question asks you to use the Naive Bayes assumption to estimate a probability value given some data, which is exactly what you use Naive Bayes for.

Naive Bayes Classifier

Given:

- a prior $P(Y)$
- n conditionally independent features $\boldsymbol{X}$ given the class Y
- calculated likelihood for each X_i of the form $P(X_i|Y)$

Your decision rule is: (note h_{NB} is Naive Bayes, not Neg Binom)

$$y^* = h_{NB}(x) = \arg\max_y P(y)P(x_1, \ldots, x_n|y)$$
$$= \arg\max_y P(y) \prod_i P(x_i|y)$$

Although the assumption that the predictor (independent) variables are independent is not always accurate, it does simplify the classification task dramatically, since it allows the class conditional densities $p(x_k|C_j)$ to be calculated separately for each variable, i.e., it reduces a multidimensional task to a number of one-dimensional ones. In effect, Naive Bayes reduces a high-dimensional density estimation task to a one-dimensional kernel density estimation. Furthermore, the assumption does not seem to greatly affect the posterior probabilities, especially in regions near decision boundaries, thus, leaving the classification task unaffected.
Nave Bayes is NOT sensitive to irrelevant features. However, this assumes that we have good enough estimates of the probabilities, so the more data the better.

Digit classification example

Simplify images of digits to pixels, and assign them True or False for whether they are "on".
Each input maps to a feature in a vector. E.g. pixel in the 0th for and 0th column is $F_{0,0}$.
The Naive Bayes model is:
$P(Y|F_{0,0}, \ldots, F_{15,15}) \propto P(Y) \prod_{i,j} P(F_{i,j}|Y)$. We assume the features are independent given the class. We need to learn the distribution of pixels on at each pixel given each number.
How to calculate the prior, P(Y):
$P(Y) = \frac{count(Y = y)}{\sum_{y'} Count(Y = y'}$ (denominator is summing over all y values)
How to calculate the likelihood:
$P(X_i = x|Y = y) = \frac{Count(X_i = x, Y = y)}{\sum_{x'} Count(X_i = x', Y = y)}$

For binary features, use the Beta prior and MAP.

Just like likeihood of binomial previously!
$P(\theta|D) = \frac{\theta^{\beta_H+\alpha_H-1}(1-theta)^{\beta_T+\alpha_T-1}}{B(\beta_H+\alpha_H, \beta_T+\alpha_T)} \approx Beta(\beta_H+\alpha_H, \beta_T+\alpha_T)$
Chose θ using MAP:
$\widehat{\theta} = \arg\max_\theta P(\theta|D) = \frac{\alpha_H+\beta_H-1}{\alpha_H+\beta_H+\alpha_T+\beta_T-2}$.
Once again, the Beta prior is equivalent to adding extra observations for each feature.
If you don't have a lot of observations, the prior is important.
And as the number of observations goes to ∞, the prior is "forgotten".

Multinomials: Laplace Smoothing

Laplace's estimate:
Pretend you saw every outcome k extra times:
$P_{LAP,k}(x) = \frac{c(x)+k}{N+k|X|}$
N = number of observations.
$|X|$ = the number of categories you are counting.
k is the strength of the prior; how much of the prior information you are going to enforce

Example: $P_{LAP,0}(X) = \langle \frac{2}{3}, \frac{1}{3} \rangle$.

Set $k = 1$. $|X|$ is 2. N is 3. $P_{LAP,1}(X) = \langle \frac{3}{5}, \frac{2}{5} \rangle$

$P_{LAP,100}(X) = \langle \frac{102}{203}, \frac{101}{203} \rangle$

Laplace for conditionals: Smooth each condition independently:
$P_{LAP,k}(x|y) = \frac{c(x,y)+k}{c(y)+k|X|}$

Subtleties of the NB classifier

(1) Usually the features are not conditionally independent:
$P(X_1, \ldots, X_n|Y) \neq \prod_i P(X_i|Y)$
The actual probabilities $P(Y|\boldsymbol{X})$ are often biased towards 0 or 1.
Nonetheless, NB is the single most used classifier out there. It performs well even when the independence assumption is violated.
(2) Overfitting
Conditional probabilities can easily be calculated as zero. Zero probabilities kill that class' chance at being called.
??? If the feature is binary, we can use MAP with a beta prior. ???
That's equivalent to adding extra observations for each feature.

NB for text classification

- Need a feature vector with a suitably small number of features. Bag of words model is commonly used.
- i is the i^{th} word
- NB assumption(___) helps a lot. $P(X_i = x_i|Y = y)$ is just the probability of observing word x_i in a document on topic y.
  $h_{NB}(x) = \arg\max_y P(y) \prod_{i=1}^{LengthDoc} P(x_i|y)$
- Additional assumption: bag of words model.
  Order of words ignored. Works really well.
  $P(X_i = x_i|Y = y) = P(X_k = x_i|Y = y)$ (k is the k^{th} word (?); all positions have the same distribution).
  $P(y) = \prod_{i=1}^{LengthDoc} P(x_i|y)$
- Prior, $P(Y)$, is the fraction of documents of each topic.
- Likelihood, $P(X_i|Y)$ is count for how many times you saw the word in documents of this topic. This distribution is shared across all positions i.
- Testing: Use Naive Bayes decision rule.
  $h_{NB}(x) = \arg\max_y P(y) \prod_{i=1}^{LengthDoc} P(x_i|y)$

NB for continuous X_i

- k is an index over all possible labels.
- i is the i^{th} feature. Here it is the pixel.
- j is the j^{th} training example.
- X_i^j is the i^{th} pixel in the j^{th} training sample.
- Y^j is the label corresponding to the j^{th} training example.
- y_k is the k^{th} label
- j is j^{th} training example.
- $\delta(x) = 1$ if x true, else 0.
- h: the function that returns the best class.

Example: character recognition where the darkness of each pixel is continuous.

Gausian Naive Bayes (GNB) for continuous features

Find parameter that makes all the data points most likely. What parameters explain our data best?

Naive Bayes continuous video:
`https://www.youtube.com/watch?v=r1in0YNetG8`

$$P(X_i = x|Y = y_k) = \frac{1}{\sigma_{ik}\sqrt{2\pi}} e^{\frac{-(x-\mu_{ik})^2}{2\sigma_{ik}^2}}$$

- μ_{ik} is the mean of the values for the i^{th} feature for the k^{th} class.
- σ_{ik} is the standard deviation of the values for the i^{th} feature for the k^{th} class.

Sometimes we assume one or both of these:

- variance is independent of Y (i.e. σ_i)
- variance is independent of X_i (i.e. σ_k)

If we assume both, we assume just one σ without subscripts.

Estimating parameters for discrete Y and continuous X_i:

- **mean**: $\widehat{\mu}_{ik} = \frac{1}{\sum_j \delta(Y^j = y_k)} \sum_j X_i^j \delta(Y^j = y_k)$
  - first term: divide by the number of training examples that are of class k.
  - second term: summing the continuous input of pixel i for all examples in the training set that match label k.
  - so this is just an average brightness for pixel i given class k using all the training data.
- **variance**: $\widehat{\sigma}_{ik}^2 = \frac{1}{\sum_j \delta(Y^j = y_k) - 1} \sum_j (x_i^j - \widehat{\mu}_{ik})^2 \delta(Y^j = y_k)$
  - first term: divide by the number of training points of class k minus 1.
  - second term: sum the squared difference in brightness of pixel i compared to the mean for that pixel and label.

We don't need to use a Gaussian for the prior, $P(y)$. We aren't optimizing over it, so it is safe to count.

When Bayes Classifier is Optimal:

In Bayes we are learning the function h that produces labels Y based on inputs $\boldsymbol{X}$. More formally:
$h : \boldsymbol{X} \mapsto Y$, or
we are learning "the function h that maps features $\boldsymbol{X}$ to labels Y".

If you know the true $P(Y|\boldsymbol{X})$, then
$h_{Bayes} = \arg\max_y P(Y = y|\boldsymbol{X} = x)$.

Note the subscript is Bayes, not Naive Bayes; no assumption of conditional independence. Also, the conditionality is back to likelihood instead of posterior.

Theorem: Bayes (not NB) classifier h_{Bayes} is optimal.
$error_{true}(h_{Bayes}) \leq error_{true}(h), \forall h$

This is theoretical result: we don't know $P(\boldsymbol{x})$. We can't calculate the true Bayes classifier b/c we don't know the distribution of all the data.) We also don't know $P(Y|\boldsymbol{X})$, the true class' highest probability. Usually that's hidden; if we knew it we would go home happy.
Plain english: the predictions you get from Bayes are better than any other function/prediction available.

Proof:

$$P_h(error) = \int_x P_h(error|\boldsymbol{X})P(\boldsymbol{X})$$

def. of error: $P_h(error|\boldsymbol{x}) = \int_y \delta(h(\boldsymbol{X}), Y)P(Y|\boldsymbol{X})$

(note, usually we'd sum over the classes, Y)

$$= \int_x \int_y \delta(h(\boldsymbol{X}), Y)P(Y|\boldsymbol{X})P(\boldsymbol{X})$$

(the double integral is zero when $P(Y|\boldsymbol{X})$ is largest, which is when the correct classification was selected.)

We are averaging over novel data sets that are generated under the same conditions.
Different notation: delta has a comma in the parentheses and not an equality.

- $P_h(error)$ is probability of error across all classifications.
- $\delta(h(\boldsymbol{X}, Y)$ is 1 if your X is classified right and 0 if not.
- $P_h(error|\boldsymbol{x})$ is the probability that your classification is wrong.
- $\int_x P_h(error|\boldsymbol{X})P(\boldsymbol{X})$ is the expectation of the errors.
- $\delta(h(\boldsymbol{X}), Y)$ is

Proof in words: ???
Aside: note that for one classification y is not a vector. It is a point.

Logistic Regression

- Another probabilistic approach to classification (categorical predictions).
- directly optimizing the conditional log likelihood
- Discriminative: learn $P(Y|X)$ directly then discriminate between classes. We solved similar problems using Bayes before (a generative approach). But if we don't want to bother with modeling all those joint or conditional probabilities, we can do this discriminative approach instead.
- Parameters from gradient ascent
- Linear, uses a probabilistic model, and is discriminative.
- produces weights, w^* of dim n + 1 (n = number of features in each training point). These parameters are tied together, unlike in Naive Bayes where there is independence for parameters for different classes.
- Use the dot product of those weights against the feature vectors as an input to the sigmoid function.

Logistic Regression is the discriminative counterpart to a Naive Bayes generative classifier over Boolean features. The difference between logistic and Naive Bayes is just one word: "conditional". We are maximizing conditional log likelihood, that is conditional on X. We are not going to spend our time encoding the distribution over X.
Parameters are tied together, unlike in Naive Bayes.
Can use discrete or continuous outputs.
It's a linear classifier; the decision rule is a hyperplane.
You optimize it (find weights) by gradient ascent, which works because it is concave.
You can use "maximum conditional a posteriori" for regularization.

Note: $\frac{e^x}{1+e^x} = \frac{1}{1+e^{-x}}$

Summary from non-class sources:
We are still using linear regression in the inputs, but putting the result into a sigmoid function.
Recall $w_0 + w_1x_1 + w_2x_2 + w_3x_3 = w^T x$ and $x = (1, x_1, x_2, x_3)$.
$P(death|x) = \sigma(w^T x)$ where σ, the sigmoid function, converts your regression output into a sigmoid curve.
$\sigma(a) = \frac{1}{1+e^{-a}} = \frac{1}{1+e^{-(w_0+w_1x_1+w_2x_2+w_3x_3)}} = \frac{1}{1+e^{-(w^T x)}}$
We can convert this to a linear relationship by "taking the logit".
The logit (log odds) is the inverse of the logistic.
$F(x) = \sigma(a)$ above. It is the probability that the dependent variable equals a case, given some linear combination of the predictors. It can range from $-\infty$ to ∞ The logit is $\ln \frac{F(x)}{1-F(x)}$, or equivalently, after exponentiating both sides:
$\frac{F(x)}{1-F(x)} = e^{w^T x}$
The logit (i.e., log-odds or natural logarithm of the odds) is equivalent to the linear regression expression.
Note used odds ratio: $\frac{p}{1-p}$

$$\text{logit}(\frac{1}{1+e^{-(w^T x)}}) = \log(\frac{\frac{1}{1+e^{-(w^T x)}}}{1-\frac{1}{1+e^{-(w^T x)}}})$$

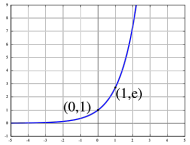
We can now proceed with linear regression.
Note that our predictions are now on the log scale; this impacts interpretation of the coefficients.

Lecture's presentation:
Notation:

- x_i^j: the i^{th} attribute of data point j
- y^j: the j^{th} class
- x^j: the j^{th} training example

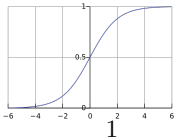
Once again we don't want to try to estimate $P(X, Y)$; that is challenging due to the size of the distribution.
We could make the Naive Bayes assumption and only need to calculate $P(X_i|Y)$, but if we want $P(Y|X)$, why not learn that directly? You can use logistic regression.

Exponential:



$$y = e^x = exp(x)$$

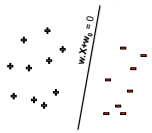
Reuse ideas from regression, but let the y-intercept define the probability.
$P(Y = 1|\boldsymbol{X}, \boldsymbol{w}) \propto exp(w_0 + \sum_i w_i X_i)$
With normalization constants:
$P(Y = 0|\boldsymbol{X}, \boldsymbol{w}) \frac{1}{1 + exp(w_0 + \sum_i w_i X_i)}$
$P(Y = 1|\boldsymbol{X}, \boldsymbol{w}) \frac{exp(w_0 + \sum_i w_i X_i)}{1 + exp(w_0 + \sum_i w_i X_i)}$



Logistic function:

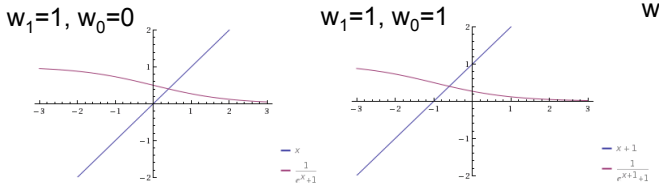
$$y = \frac{1}{1 + exp(-x)}$$

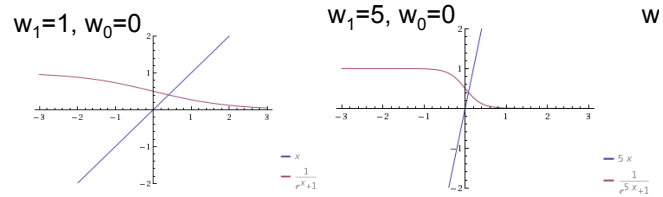
Making a decision boundary out of logistic equations:
Output the Y with the highest $P(Y|X)$.
If binary Y, output Y=1 if $1 < \frac{P(Y = 1|X)}{P(Y = 0|X)}$
That simplifies to just $1 < exp(w_0 + \sum_i w_i X_i)$ or $0 < w_0 + \sum_i w_i X_i$

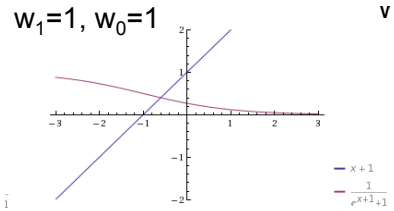


The decision boundary is a line (or hyperplane), hence we have a linear classifier!

For $P(Y = 0|\boldsymbol{X}, \boldsymbol{w}) = \frac{1}{1 + exp(w_o + w_1 x_1)}$:







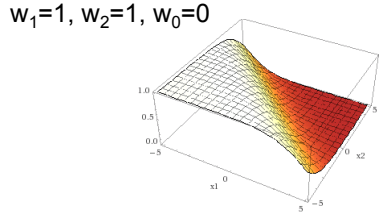
(See notes for more w_0, w_1 values plotted.)
In these plots, Y is the probability that the class is 1.
The red curve is the sigmoid. The blue line is the decision boundary.
The decision boundary is from the equation $0 = w_1 X + w_0$.

Larger weights result in a sharper curve. The bias w_0 shifts there the middle of the curve is.
The red sigmoid defines a probability distribution over Y in {0,1} for every possible input X.

The decision boundary leads to $P(Y = 0|X, w) = 0.5$ when you are at the $y = 0$ point on the line.
(E/J words: when the blue line crosses the x axis, that's when the sigmoid curve is above 1/2, which corresponds to classifying it as a no/0.)

If $w_0 = 0$, $P(Y = 0|X, w) = 0.5$ when at the $y = 0$ point on the line: the exponent just becomes 0 and the whole equation evaluates to 0.5. If $w_0 \neq 0$, this statement is no longer true. And similar argument for the 2-d case. The slope of the line defines how quickly the probabilities go to 0 or 1 around the decision boundary.

2D inputs:
For $P(Y = 0|\boldsymbol{X}, \boldsymbol{w}) = \frac{1}{1 + exp(w_o + w_1 x_1 + w_2 x_2)}$:



$P(Y = 0|X, w)$ decreases as $w_0 + \sum_i w_i x_i$ increases. Again, if you set the stuff inside the exponential to zero, you get the decision boundary hyperplane.

Finding the w coefficients: Loss Function

Simple Intro:
You can use the chain rule in the opposite direction to decompose the loss function into the log likelihood of X as well as the conditional log likelihood of the labels y given X.
Class Version:
Generative (Naive Bayes) loss function: Now j is a data point with observations indexed over i.

$$\ln P(D|\boldsymbol{w}) = \sum_{j=1}^{N} \ln P(x^j, y^j|\boldsymbol{w}) \quad \text{(the full log-likelihood)}$$

use Bayes' rule to rewrite conditionally

$$= \sum_{j=1}^{N} \ln[P(y^j|x^j, \boldsymbol{w}) P(x^j|\boldsymbol{w})]$$

$$= \sum_{j=1}^{N} \ln P(y^j|x^j, \boldsymbol{w}) + \sum_{j=1}^{N} \ln P(x^j|\boldsymbol{w})$$

We decide to ignore the 2nd term because it won't help you get better predictions for that data anyway. Or, "From a machine learning perspective, "God gave us the data" and we don't care about the 2nd sum."
Professor Farhadi is calling this first time a discriminative (logistic regression) loss function:
It is helping you discriminate between different classes. It's not going to help you model the data. This is unlike regression; we don't care about the value it puts out. We only care about what the resulting class is.
This is the difference between statistics and machine learning. We only care about getting the best $\boldsymbol{w}$ for discriminating between classes.
Conditional Data Likelihood: "Conditional" because you are conditioning on what $\boldsymbol{X}$ is.

$$\ln P(D_Y|D_{\boldsymbol{X}}, \boldsymbol{w}) = \sum_{j=1}^{N} \ln P(y^j|\boldsymbol{x}^j, \boldsymbol{w})$$

D_Y = ???
$D_{\boldsymbol{X}}$ = ???
Doesn't waste effort learning $P(X)$. Focuses on $P(Y|\mathbf{X})$, which is all that matters for classification.
Discriminative models cann't compute $P(\boldsymbol{x}^j|\boldsymbol{w})$! ???

Conditional Log Likelihood

Just need to figure out how to go up and find the maximum likelihood. (the binary case only).
$P(Y = 0|\boldsymbol{X}, \boldsymbol{w}) = \frac{1}{1+\exp(w_0+\sum_i w_i X_i)}$
$P(Y = 1|\boldsymbol{X}, \boldsymbol{w}) = \frac{\exp(w_0+\sum_i w_i X_i)}{1+\exp(w_0+\sum_i w_i X_i)}$
($l(\boldsymbol{w})$ is conditional data log-likelihood.)

$$l(\boldsymbol{w}) \equiv \sum_j \ln P(y^j|x^j, \boldsymbol{w})$$

Since y^j is in {0, 1}, sum over the two cases:

(the y^j and $(1 - y^j)$ act like delta functions)

$$l(\boldsymbol{w}) = \sum_j y^j \ln P(y^j = 1|x^j, \boldsymbol{w}) + (1 - y^j) \ln P(y^j = 0|x^j, \boldsymbol{w})$$

plug in the definition of the likelihoods and do algebra to get:

$$= \sum_j y^j (w_0 + \sum_i^n w_i x_i^j) - \ln(1 + \exp(w_0 + \sum_i^n w_i x_i^j))$$

While we can't find a closed-form solution to optimize $l(\boldsymbol{w})$, $l(\boldsymbol{w})$ is concave so we can to gradient ascent. Jsut need to figure out how to go up and find the maximum likelihood.

Gradient ascent to optimize w

To maximize, we can't take derivative w.r.t $\boldsymbol{w}$ and optimize b/c no closed form solution. Instead we form the gradient vector and move along the gradient. Conditional likelihood for Logistic Regression is convex. (see above)
In this case it doesn't matter b/c we have a concave function. Iterate to find $\boldsymbol{w}$: start somewhere, find gradient direction, make step in that direction, repeat, repeat.
Gradient: gives ascent or descent direction.

$$\nabla_w l(\boldsymbol{w}) = [\frac{\partial l(\boldsymbol{w})}{\partial w_0}, \ldots, \frac{\partial l(\boldsymbol{w})}{\partial w_n}]'$$

(The $'$ at the end is for transpose b/c usually a column vector.)
Update Rule:

$$\Delta \boldsymbol{w} = \eta \nabla_{\boldsymbol{w}} l(\boldsymbol{w})$$

η is the learning rate. $\eta > 0$. It can't be too big or you can get lost. It can't be too small or you take forever to converge.
Your next weights $(t+1)$ become: $w_i^{(t+1)} \leftarrow w_i^{(t)} + \eta \frac{\partial l(\boldsymbol{w})}{\partial w_i}$

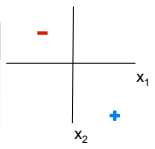
Gradient ascent is the simplest of optimization approaches. Note that conjugate gradient ascent is much better (see reading).(?)
See slides for derivation of:
$\frac{\partial l(w)}{\partial w_i} = \sum_j x_i^j (y^j - P(Y^j = 1 | x^j, w))$

Example of gradient ascent to maximize conditional log likelihood

(M(C)LE; C for conditional) Learning an approximation of the step function.
Use the equations:

- **eq 1:** $w_i^{(t+1)} \leftarrow w_i^{(t)} + \eta \frac{\partial l(\boldsymbol{w})}{\partial w_i}$
- **eq 2:** $\frac{\partial l(w)}{\partial w_i} = \sum_j x_i^j (y^j - P(Y^j = 1 | x^j, w))$ The loss function is conditional on log likelihood.
- **eq 3:** $P(Y = 1 | X, W) = \frac{\exp(w_0 + \sum_i w_i X_i)}{1 + \exp(w_0 + \sum_i w_i X_i)}$
- *note:* superscript is for the j^{th} data point and subscripts are for the i^{th} value of the input (x) array. Don't forget that there is an $i = 0$ $(x_0 = 1)$ bias term.

x_1	x_2	y
3	-3	1
-2	2	0



- begin with t=0 and $w = [w_0 + w_1 + w_2] = [0, 0, 0]$
- calculate **eq 3** for each Y.
  - For j = 0:

$$P(Y^0 = 1 | x^0, w) = \frac{\exp(0 + 0*3 + 0*(-3))}{(1 + \exp(0 + 0*3 + 0*(-3)))}$$
$$= 1/(1+1) = 1/2$$

  - For j = 1:

$$P(Y^1 = 1 | x^1, w) = \frac{\exp(0 + 0*(-2) + 0^(2))}{(1 + \exp(0 + 0*(-2) + 0^(2)))}$$
$$= 1/(1+1) = 1/2$$

- calculate the terms that go into **eq 2** by looping over the 3 values of i (bias and two other coefficients) and 2 values of j (2 data points).
  - for $i = 0, j = 0$: (j =0th (1st) training example, bias term)
    $x_0^0(y^0 - P(Y^0 = 1 | x^0, w)) = 1(1 - 0.5) = 0.5$
  - for $i = 0, j = 1$: (j =1st (2nd) training example, bias term)
    $x_0^0(y^0 - P(Y^0 = 1 | x^0, w)) = 1(0 - 0.5) = -0.5$
  - for $1 = 1, j = 0$: (j =0th (1st) training example, x_1 term)
    $x_1^0(y^0 - P(Y^0 = 1 | x^0, w)) = -2(1 - 0.5) = 1.5$
  - for $i = 1, j = 1$: (j =1st (2nd) training example, x_1 term)
    $x_1^1(y^1 - P(Y^1 = 1 | x^1, w)) = -2(0 - 0.5) = 1.0$
  - for $1 = 2, j = 0$: (j =0th (1st) training example, x_2 term)
    $x_2^0(y^0 - P(Y^0 = 1 | x^0, w)) = -3(1 - 0.5) = -1.5$
  - for $i = 2, j = 1$: (j =1st (2nd) training example, x_2 term)
    $x_2^1(y^1 - P(Y^1 = 1 | x^1, w)) = 2(0 - 0.5) = -1.0$
- Now we can compute the gradient (**eq 2**):
  - $\nabla_w l(\boldsymbol{w}) = [\frac{\partial l(w)}{\partial w_1}, \frac{\partial l(w)}{\partial w_2}, \frac{\partial l(w)}{\partial w_3}]$
    $= [0.5 - 0.5, 1.5 + 1.0, -1.5 - 1] = [0, 2.5, -2.5]$
- Use $\eta = 0.1$ to scale the gradient: (**eq 1**)
  $w = [0, 0, 0] + 0.1 * [0, 2.5, 2.5] = [0, 0.25, -0.25]$
- Start over with **eq 3** and updated w.
  $P(Y^0=1|x^0,w) \propto \exp(0+0.25*3-0.25*-3) = 0.82$
  $P(Y^1=1|x^1,w) \propto \exp(0+0.25*-2-0.25*2) = 0.27$
  i=0, j=0: $x_0^0(y^0-P(Y^0=1|x^0,w)) = 1(1-0.82) = 0.18$
  i=0, j=1: $x_0^1(y^1-P(Y^1=1|x^1,w)) = 1(0-0.27) = -0.27$
  i=1, j=0: $x_1^0(y^0-P(Y^0=1|x^0,w)) = 3(1-0.82) = 0.54$
  i=1, j=1: $x_1^1(y^1-P(Y^1=1|x^1,w)) = -2(0-0.27) = 0.54$
  i=2, j=0: $x_2^0(y^0-P(Y^0=1|x^0,w)) = -3(1-0.82) = -0.54$
  i=2, j=1: $x_2^1(y^1-P(Y^1=1|x^1,w)) = 2(0-0.27) = -0.54$
  grad = [0.13-0.27, 0.54+0.54, -0.54-0.54]
  = [-0.14,1.04,-1.04]

How to do more algorithmically:
You don't have to get all the info to update all the weights at once. Do them one at a time.
Gradient ascent algorithm: (learning rate $\eta > 0$)

do:

$$w_0^{(t+1)} \leftarrow w_0^{(t)} + \eta \sum_j [y^j - \hat{P}(Y^j = 1 \mid \mathrm{x}^j, \mathrm{w})]$$

For i=1…n: (iterate over weights)

$$w_i^{(t+1)} \leftarrow w_i^{(t)} + \eta \sum_j x_i^j [y^j - \hat{P}(Y^j = 1 \mid \mathrm{x}^j, \mathrm{w})]$$

Loop over training examples!

How do you decide when to stop?
(Not in notes!)
You need a criteria that tells you when to stop updating your gradient. Use a threshold for magnitude of the gradient. When the amount of change is small it is not changing a lot. If concave, you are close to global optima.

Why don't we wait until it gets to zero? The chances of us hitting zero is low, and we might spend a lot of time wasting CPU cycles on negligible changes.

When we are really far from the global optimum we can afford to take bigger steps. Think about the objective function. We can afford to make big steps if we are far. If we make big steps when close, we can get farther away .

You could use this policy: let ν be big at the beginning of a loop counter. Increase the value until you see a drop. "Line search" asks "what is maximum value of this parameter that gives you no drop?"

Overfitting in Logistic Regression

Intro #1:
Be weary of large parameters. Large parameters (large a in $\frac{1}{1+\exp(-a)}$) lead to a step function. That's ok if you truly want a step, but you should be concerned about overfitting the minute you see a large parameter. We could regularize again: apply a penalty for large parameters. Could do something like maximizing $w* = \arg\max_w l(w) - \lambda * (||w||_2^2)$ instead of $w* = \arg\max_w l(w)$. But that's not ideal. Might have a hard time finding a suitable value for λ. Instead, use a probabilistic way of regularization. We've done this before using a prior. Instead of MLE, we did MAP. The prior regularizes. $P(w|Y, X) \propto P(Y|X, w)P(w)$. What would be the form of our prior $(P(w))$ be? We want the posterior to be differentiable. The likelihood was a sigmoid shape, so the log likelihood had the form of was of the form $e^{something}$. We already know how to get the conditional likelihood in the form of e^{blah}. Well behaved summation of terms.
We pick a gaussian distribution for the prior for mathematical convenience. We wanted to form a function we can optimize. If it doesn't come from gaussian and does come from multinomial, what happens? We wouldn't be too wrong. And it is just a regularization. We are just trying to constrain $\boldsymbol{w}$ with something that is mathematically well behaved. Usually the prior is off. But it is only there to keep w from getting too big. It gives a low score when w is too big.
Q from student: why aren't we considering $P(X, W)$ as the prior? Why $P(Y|X, w)$? P(X,W) is discriminative. We aren't reformulating our problem. Just looking to estimate the parameters. We want to regularize the parameter not the data (?).
Intro #2:
Like in linear regression, the maximum likelihood solution prefers higher weights. Higher weights can give higher likelihood of properly classifying examples close to the decision boundary. This over-fitting causes features to have larger influences on the decision: **beware of overfitting!!!**. Again, you can use regularization to penalize high weights. This will be covered more later.

MAP for Logistic Regression

Above was M(Conditional)LE.
Recall that the MAP/MLE difference is _____.
Priors on $\boldsymbol{w}$ are commonly added for regularization.
Helps avoid very large weights and overfitting.
$P(\boldsymbol{w}|Y, \boldsymbol{X}) \propto P(Y|\boldsymbol{X}, \boldsymbol{w})P(\boldsymbol{w})$
Use a normal distribution with zero mean and "identity covariance".
(____):

$$P(\boldsymbol{w}) = \prod_i \frac{1}{\kappa\sqrt{2\pi}} e^{\frac{-w_i^2}{2\kappa^2}}$$

κ^2 is variance.

Now we form our MAP estimate:

$$w* = \arg\max_w \ln[P(\boldsymbol{w}) \prod_{j=1}^{N} P(y^j|x^j, \boldsymbol{w})]$$

$$= \arg\max_w \ln[\prod_i \frac{1}{\kappa\sqrt{2\pi}} e^{\frac{-w_i^2}{2\kappa^2}} \prod_{j=1}^{N} P(y^j|x^j, \boldsymbol{w})]$$

Instead of focusing on the stuff after the $\prod$ in the bracket we also have $P(w)$. Our prior, $P(w)$ is $\prod$ fo gaussians. The $\prod(y^j|x^,x_j)$ is called likelihood (???)
Add $\log P(\boldsymbol{w})$ to the objective:

$$\ln P(\boldsymbol{w}) \propto -\frac{\lambda}{2} \sum_i w_i^2$$

$$\frac{\partial \ln P(\boldsymbol{w})}{\partial w_i} = \lambda w_i$$

We now have a new way of optimizing w. This is a quadratic penalty (???) that drives the weights toward zero.
It adds a negative linear term to the gradients (???).
Also applicable in linear regression!

Review: MLE vs. MAP for Logistic Regression

Maximum conditional likelihood estimate:

$$w* = \arg\max_w \ln[\prod_{j=1}^{N} P(y^j|x^j, \boldsymbol{w})]$$

$$w_i^{(t+1)} \leftarrow w_i^{(t)} + \eta \sum_j x_i^j [y^j - \widehat{P}(Y^j = 1|x^j, \boldsymbol{w})]$$

Maximum conditional a posteriori estimate:

$$w* = \arg\max_w \ln[P(\boldsymbol{w}) \prod_{j=1}^{N} P(y^j|x^j, \boldsymbol{w})]$$

$$w_i^{(t+1)} \leftarrow w_i^{(t)} + \eta\{-\lambda w_i^{(t)} + \sum_j x_i^j [y^j - \widehat{P}(Y^j = 1|x^j, \boldsymbol{w})]$$

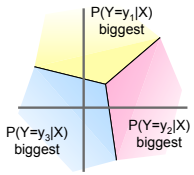
The λ term complains when the value is big. This $-\lambda$ if from the derivative of the gaussian. The whole point: if you add a prior and take a derivative, it is a form of regularization and should act as a penalty on big w.

Vocab

- **discriminative**: estimates joint probabilities. E.g. $p(Data, Zebra)$, $p(Data, NoZebra)$.
- **generative**: E.g. $p(Zebra|Data)$, $p(NoZebra|Data)$.

Multiclass Logistic Regression (discrete labels)

Define a weight vector w_i for each y_i where i ranges from 1 to $R-1$ for R classes. You don't need a weight vector for the R^{th} class b/c its probability is 1 - the sum of the rest.



Each category will have its own w values.
Don't need to train k w values. k-1. b/c they have to sum to 1.

$$P(Y = 1|X) \propto \exp(w_{10} + \sum_i w_{1i} X_i)$$

note: $\boldsymbol{w}$ is now a matrix, not an array.

$$P(Y = 2|X) \propto \exp(w_{20} + \sum_i w_{2i} X_i)$$

...

the last probability is defined relative to the rest.

$$P(Y = r|X) = 1 - \sum_{j=1}^{r-1} P(Y = j|X)$$

After normalizing, your probabilities become:

$$P(Y = y_k|X) = \frac{\exp(w_{k0} + \sum_{i=1}^{n} w_{ki} X_i)}{1 + \sum_{j=1}^{R-1} \exp(w_{j0} + \sum_{i=1}^{n} w_{ji} X_i)}$$

the last class is 1 - the sum of the other probabilities:

$$P(Y = y_R|X) = \frac{1}{1 + \sum_{j=1}^{R-1} \exp(w_{j0} + \sum_{i=1}^{n} w_{ji} X_i)}$$

Note: features can be discrete or discontinuous.

Gaussian Naive Bayes vs Logistic Regression.

Gaussian Naive Bayes with class-independent variances is representationally equivalent to Linear Regression. ("If you do have an infinite number of training examples, the GNB and LR produce identical classifiers.") The solutions different because of the objective (loss) function.

If you do have class-independent variances and the underlying model is gaussian, GNB does better than LR. You get an incorrect model for GNB if you don't have class-independent variances. LR is less biased because it does not assume conditional independence.
If you have enough categories, GNB w/ class independence will produce identical results to LR. Given enough points & the assumption features independence of class is held true, GNB will do better. If these assumptions are *not* true then LR will do better.

You could use either to learn $f : X \rightarrow Y$ for X that is a vector of real-valued features $(< X_1, \ldots, X_n >)$ and boolean Y.

The two approaches make different assumptions:

- NB assumes features are independent given the class. This is an assumption on $P(\boldsymbol{X}|Y)$.
- LR assumes functional form of $P(Y|\boldsymbol{X})$, and makes no assumption on $P(\boldsymbol{X}|Y)$.

Gaussian Naive Bayes classifier would assume:

- all X_i are conditionally independent given Y. If you fix what hump you are looking at, you don't need to worry about the rest of the parameters.
- model $P(X_i|Y = y_k)$ as Gaussian with $N(\mu_{ik}, \sigma_i)$. (Your inputs are each produced by a gaussian.)
- model $P(Y)$ as Bernoulli$(\theta, 1-\theta)$

This implies the form of $P(Y|X)$ is:
$P(Y = 1|X =< X_1, \ldots, X_n >) = \frac{1}{1 + \exp(w_0 + \sum_i w_i X_i)}$.

In the slides, he derives the form.
(??) You see that if you assume Gaussian in Logistic Regression, you get:
$w_0 = \ln \frac{1-\theta}{\theta} + \frac{\mu_{i0}^2 + \mu_{i1}^2}{2\sigma_i^2}$ and $w_i = \frac{\mu_{i0} + \mu_{i1}}{\sigma_i^2}$
His confusing summary:

Gaussian Naïve Bayes vs. Logistic Regression

Set of Gaussian Naïve Bayes parameters (feature variance independent of class label) ⟺ Set of Logistic Regression parameters

Can go both ways, we only did one way

- Representation equivalence
  - **But only in a special case!!!** (GNB with class-independent variances)
- But what's the difference???
- **LR makes no assumptions about P(X|Y) in learning!!!**
- **Loss function!!!**
  - Optimize different functions ! Obtain different solutions

Attempted translation:
Note: you need more parameters to train Naive Bayes.
$4n + 1$ parameters for GNB, $n + 1$ parameters for Logistic Regression.
The NB parameter estimates are uncoupled, so there are more of them. (The Logistic Regression parameter estimates are coupled.)
If you don't have infinite data (? "non-asymptotic analysis"):
For n attributes in X, NB needs $O(\log n)$ samples to converge and Logistic Regression needs $O(n)$. So GNB converges more quickly to its (perhaps less helpful) "asymptotic estimates."
You have to do all of the optimization at once, all together if you are doing logistic regression. ??? Easier in a certain sense to fit a Gaussian model. Typically we have a lot of data, and that data is not gaussian so Logistic Regression is probably better.

More notes from class

Logistic regression makes no assumption about $P(X|Y)$. Naive Bayes assumed independence of parameters, conditioned on label.
Optimizing different loss functions.

Which is better? Need to know how many parameters each classifier would need. # of parameters, # of features? How many parameters does NB need? ?? $2cC$ classes. C, d, n. C classes, d is # of training examples, n is number of features. $4n + 1$??. For logistic regression, the output was weights, w*. Dim was n + 1. NB parameters are independent (uncoupled). Logistic regression: the parameters are tied together.

Gen vs Disc. for NB/LR
If you have enough categories, GNB w/ class independence will produce identical results: Given enough points and the assumption features independence of class is held true If this is *not* true then LR will do better (when GNB is false). **With limiting data, use GNB.** GNB may converge faster but to something less true.

Do we do optimization with NB? No. We start w/ joint distribution, $P(x|y) * prior$, ..., take the max. We are not ML optimizing. Not required to optimize every time we learn something. **Both LR and NB are trying to do the same task, and one doesn't require training**.

NB requires less training examples.

Use logistic regression by default.

Is there a global optimum for LR with regularization? No. But still should be close. If you add a regularization that breaks the concavity of the function, you can't prove anything.

Logistic Regression Protocol

Steps:

- start with given data: x and labels.
- form conditional log likelihood function.
- either optimize that or add a prior
- do hill climbing until you stop seeing big changes in the weights
- then you claim victory

For binary classification:
The final output of trained regression is the weight vector, $w*$. It is d+1 dimensional for a d-dimensional feature vector. The +1 is for bias (that coefficient is 1).
How do you use it?
You found the form of decision boundary was
$P(Y = 1|X, w) = exp()/(1 + exp())$. Put in the x with the w.
Compute the probability of Y=0 and Y=1, pick the biggest result.
How do you know if your weights are good? Test on held-out data (NOT the test data).

Andrew Ng

* gives numbers between 0 and 1 (good for classification)
* called "logistic regression" but it is really for classification. (don't be confused by "regression")
* "sigmoid function" and "logistic function" are essentially synonymous.

Perceptrons

- Decision mechanism: sign of $w \cdot x$. If + say yes, if - say no.
- Linear classifier in feature space.
- Error driven, not probabilistic.
  - Mistake driven rather than model drive.
  - Parameters from reactions to mistakes
- does better with linearly separable data.
- Parameters are from a discriminative interpretation
- To train, you go through the data until the held-out accuracy maxes out.
- Just moving the plane to satisfy your labels. Then project new sample into this space and see which side it was on.
- Note you can scale your w (weight) vector(s) by any constant because all you care about is sign$(w \cdot x)$. This rescales your gamma by that constant too!
- Bias allows you to have decision boundaries that don't go through the origin. You could us any number, but 1 is used by convention.
- Go through the data point by point, not feature by feature. Feature by feature would assume independence of the features (bad).
- If linearly separable, it will converge. And we will know how fast it will converge.

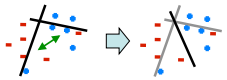
Properties of Perceptron

Separability: some parameters get the training set perfectly correct.
Convergence: if the training is separable, the perceptron will eventually converge.
Mistake Bound: the maximum number of mistakes (for the binary case) is related to the margin or degree of separability:
$mistakes \leq \frac{R^2}{\gamma^2}$.

Sort of inspired by what might happen in a human brain. Neurons send info. A module sums them up & produces a result.

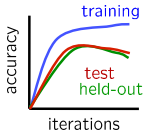
Problems with the Perceptron

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



Mediocre generalization: finds a "barely" separating solution

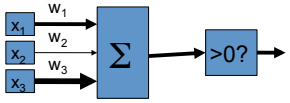
Overtraining: test / held-out accuracy usually rises, then falls
- Overtraining is a kind of overfitting



Lines might not be optimal: issues with generalizibility.
Leads into SVMs.

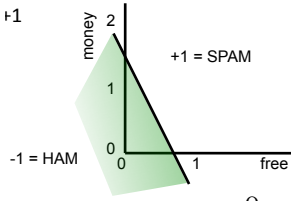
Linear Classifiers

Inputs are feature values.
Each feature has a weight.
Sum is the activation. activation$_w(x) = \sum_i w_i x_i = w \cdot x$
If the activation is positive, chose output class 1.
If the activation is negative, chose output class 2.



For a binary decision rule:
In the space of feature vectors:

- examples are points
- any weight vector is a hyperplane
- one side corresponds to y = +1
- the other side corresponds to y = -1
- ??? The $w \cdot x = 0$ is the solution to the line.



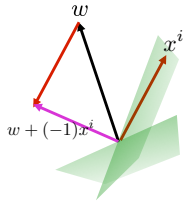
$$w \cdot x = 0$$

The black line is the decision boundary.
w is a vector normal to the decision boundary, and points towards the + label points.

Binary Perceptron Algorithm

- start with zero weights: $w = 0$
- for $t = 1 \ldots T$ (T passes over the data):
  - for $i = 1 \ldots n$ (each training example)
    - Classify with current weights:
      $y = sign(w \cdot x^i)$
      (sign(z) is +1 if $z > 0$, else -1)
    - if correct (i.e. $y = y^i$), don't change weights.
    - if it was wrong, update with $w = w + y^i x^i$

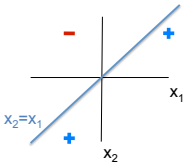
Figure showing "you got it wrong:"



The -1 is the y^i in the equation above.

- For t=1..T, i=1..n:
  - $y = sign(w \cdot x^i)$
  - if $y \neq y^i$
    $w = w + y^i x^i$

x_1	x_2	y
3	2	1
-2	2	-1
-2	-3	1



Initial:
- w = [0,0]

t=1,i=1
- [0,0]•[3,2] = 0, sign(0)=-1
- w = [0,0] + [3,2] = [3,2]

t=1,i=2
- [3,2]•[-2,2]=-2, sign(-2)=-1

t=1,i=3
- [3,2]•[-2,-3]=-12, sign(-12)=-1
- w = [3,2] + [-2,-3] = [1,-1]

t=2,i=1
- [1,-1]•[3,2]=1, sign(1)=1

t=2,i=2
- [1,-1]•[-2,2]=-4, sign(-4)=-1

t=2,i=3
- [1,-1]•[-2,-3]=1, sign(1)=1

Converged!!!
- $y=w_1x_1+w_2x_2 \rightarrow y=x_1+-x_2$
- So, at y=0 → $x_2=x_1$

$w \cdot x$ and the boundary between positive and negative answers
If a point has w * x = 1000, a small change in x might change w * x' to 999, or 1001, but it surely won't make w * x a negative value. On the other hand, if w * x = 0.0001, even tiny changes to x might make w * x' negative. And by extension, cases where w * x = 0 where even the tiniest change might make w * x' change sign. So the values where w * x = 0 are those that are on the boundary between positive and negative examples.[1]

So **the equation w * x = 0 defines the boundary between the positive and negative region**. Now let's unpack that statement. w is a fixed vector, while x is a variable point. So think of w = [w1, w2] as constants, and x = [x1, x2] as variables. The equation w * x = 0 is just another way of writing the equation w1 x1 + w2 x2 = 0. But this is a linear equation in two variables, so it defines a line. You can algebraically solve the equation for x2, and that gives you the "standard form" of the equation of a line, which you can then draw.

[1] The boundary of a region is defined as the set of points where even points very close by can be outside that region.

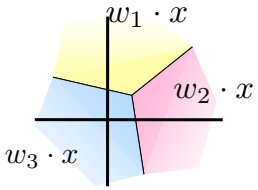
Multiclass Decision Rule

If we have more than two classes:

- we have a weight vector for **each** class: w_y
- we calculate an activation for each class:
  $\text{activation}_w(x, y) = w_x \cdot x$
- the highest activation wins:
  $y^* = \arg\max_y(activation_w(x, y))$
  "win the vote"

For each point, look at all 3 classes and see which is farthest from the hyperplane. You can treat the dot product (+ bias) as a confidence from the hyperplane you are.

Example: y is {1,2,3}
- We are fitting three planes: w_1, w_2, w_3
- Predict i when $w_i \bullet x$ is highest



Example

"win the vote"

x

BIAS	: 1
win	: 1
game	: 0
vote	: 1
the	: 1
...	

w_{SPORTS}

BIAS	: -2
win	: 4
game	: 4
vote	: 0
the	: 0
...	

$x \cdot w_{SPORTS} = 2$

$w_{POLITICS}$

BIAS	: 1
win	: 2
game	: 0
vote	: 4
the	: 0
...	

$x \cdot w_{POLITICS} = 7$

w_{TECH}

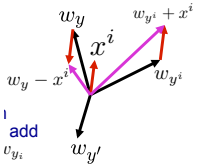
BIAS	: 2
win	: 0
game	: 2
vote	: 0
the	: 0
...	

$x \cdot w_{TECH} = 2$

POLITICS wins!!!

Binary Perceptron Algorithm

- start with zero weights: $w_y = 0$
- for $t = 1 \ldots T$ (T passes over the data):
  - for $i = 1 \ldots n$ (each training example)
    - Classify with current weights:
      (no more $y = sign(w_y \cdot x^i)$)
      instead: $y = \arg\max_y w_y \cdot x^i$
    - if correct (i.e. $y = y^i$), don't change weights.
    - if it was wrong, update two vectors:
      $w_{y^i} = w_{y^i} + x^i$
      If class 2 was the right label, add x_i to w_2.
      Also want to push down the wrong classification:
      (Didn't have to push one down for binary because adding to one pushed the other.) $w_y = w_y - x^i$

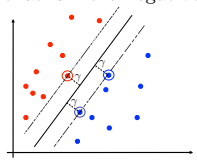


Linear Separability

binary case

Recall $||x||_2 = \sqrt{\sum_i x_i^2}$

The data is linearly separable with margin γ if:
$\exists . w \forall t . y^t (w \cdot x^t) \geq \gamma > 0$.
Plain english: the data is linearly separable if there exists a w that has a margin greater than zero for all points t.
Note: for $y^t = 1$, $w \cdot x^t \geq \gamma$ and for $y^t = -1$, $w \cdot x^t \leq -\gamma$.
Plain english: points having label = 1 have a dot product that is greater than γ, and points that have label = -1 have a dot product that is more negative than $-\gamma$.



maximum number of mistakes for training linearly separable binary data

Here, assume the data is separable with margin γ and the weight vector is a unit vector:
In math notation, this is: $\exists w^*$ such that $||w^*||_2 = 1$ and $\forall t . y^t (w^* \cdot x^t) \geq \gamma$.)
Recall that you can scale your w (weight) vector(s) by any constant because all you care about is $\text{sign}(w \cdot x)$, but that this scales your γ. You are just multiplying the equation above by a constant.
Also assume some maximum radius R for your data points:
$\forall t . ||x^t||_2 \leq R$
Then the number of mistakes (parameter updates) made by the perceptron is bounded by $mistakes \leq \frac{R^2}{\gamma^2}$.

For full inductive proof, see slides. Strategy: let w^k be the weights after the k-th update (mistake). Then $k^2\gamma^2 \leq ||w^k||_2^2 \leq kR^2$
Therefore $k \leq \frac{R^2}{\gamma^2}$.
If there is a linear separator, the Perceptron will find it!

The # of mistakes you need to make for a perceptron to converge is bounded by a ratio of R/γ^2.

What if gamma is small? Starting to violate linearly separability rule. Shouldn't trust perceptron under this.

Logistic Regression & Perceptron similarities

Logistic regression:
In vector notation, y is in the set $\{0, 1\}$.
$w = w + \nu \sum_j [y^j - P(y^j | x^j, w))] x^j$
Perceptron:
When y is in $\{0, 1\}$:
$w = w + [y^j - sign^0(w \cdot x^j)] x^j$
Note: $\text{sign}^0(z) = +1$ if $z > 0$ and 0 otherwise.

Differences:
1. Online vs. batch learning. Logistic regression is batch, Perceptron is on-line.
2. Logistic is probabilistic and Perceptron is error-driven.

Large-Margin Classifiers & Support Vector Machines

- perceptron was online, this is batch.
- desire for large margins led us away from perceptrons toward this.
- There are so many lines that can separate the points. We want good margins so it does better **on the test set**.
- We want to build the classifier that gives the biggest margins.
- Find weights w. It is a linear classifier, so form $w \cdot x$, and get sign or see what side of the line it appears.
- The dot product is projection onto the line. ??? (use better vocab).
  - y is the direction to go
  - $w \cdot x$ is projection
  - w_0 is bias
- label times prediction
- Line vs classifier: can run perceptron to get a line, then move it around to get the high margin one. But you don't need to do this (b/c convex optimization.) (HW 3 Q1 may be the by-hand version.)
- Need to normalize the weights w. There are an infinite ways to write the same line. As you increase the magnitude of the weights, the dot product goes up. You can always increase w and get a better objective function. **so you need to prevent the optimizer from getting an easy way out.** Maximizing gamma (minimizing $||w||_2$) takes care of this.
- maximizing the margin is minimizing the norm2 of the weight vector subject to the constraint that we don't like to make mistakes.
- There are very few points that play a role in the formation of the hyperplane. Those points are called support vectors. The points far from the line play no role.
- with SVM, if you assume linearly seperable, you come up w/ line that maximizes the margin.
- Can think of SVMS of minimizing loss + regularization. No closed form solution but convex, concave, can solve w/ quadratic programming. Standard solvers.
- Two ways of dealing with nonlinearity: soft margin and kernels
- Desire for nonlinear boundary → kernel.

what happens with very small w?

Small w → big margin → poor performance.
Vocab:
- **Large margin classifier**:
- **Support Vector**:
- **Support Vector Machine**:
- **hard SVM**: don't want anything inside your margin.
- **soft SVM**: softer: compute distance to hyperplane.

Notation:
- i or j: the i^{th} or j^{th} data (training) point.

- $\boldsymbol{w}$: the weights of the model. The black line is perpendicular to this vector.
- $\boldsymbol{w} \cdot \boldsymbol{x}$: the distance from x to the decision boundary.
- $\boldsymbol{w} \cdot \boldsymbol{x} + w_0$: to move the decision boundary off the origin, you need to shift it by a constant.
- $\boldsymbol{w_0}$: a constant that defines a line parallel to the decision boundary and is w_0 units away.

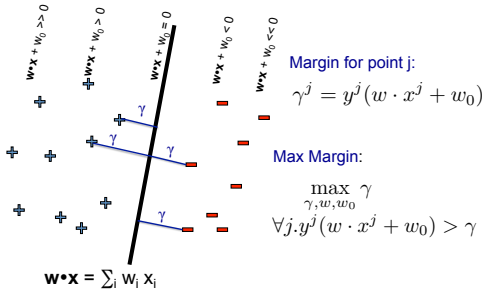
Intro

Like perceptron, but maximizes the margin.

Optimizing a small weight vector: $min_w \frac{1}{2}||w||^2$

and getting points right: $\forall$ points i, y $w_{y^*} \cdot x^i \geq w_y \cdot x^i + 1$

Summary:



Margin for point j:

$$\gamma^j = y^j(w \cdot x^j + w_0)$$

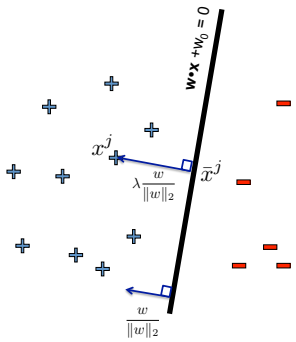
Max Margin:

$$\max_{\gamma,w,w_0} \gamma$$
$$\forall j.y^j(w \cdot x^j + w_0) > \gamma$$

There are many possible ways to write the same line:
If $\boldsymbol{w} \cdot \boldsymbol{x} + w_0 = 0$, then these also work: $2\boldsymbol{w} \cdot \boldsymbol{x} + 2w_0 = 0$, $1000\boldsymbol{w} \cdot \boldsymbol{x} + 1000w_0 = 0$, ...
Any constant scaling has the same intersection with the z = 0 plane, so you get the same dividing line.
This is why we <u>don't</u> want $\max_{\gamma,w,w_0}$.

Want a w so that when you multiply by x^j, the sign is similar to y^j but also gamma away. Nonzero gamma $\rightarrow$ margin. If you used 0 instead of γ, you would basically just be doing the perceptron.

The distance from x^j to the decision boundary is given by $\lambda \frac{w}{||w||_2}$:



$\bar{x}^j$ is the component of x^j that is normal to w.
So $x^j = \bar{x}^j + \lambda \frac{w}{||w||_2}$
(Recall $||w||_2 = \sqrt{\sum_i w_i^2}$)

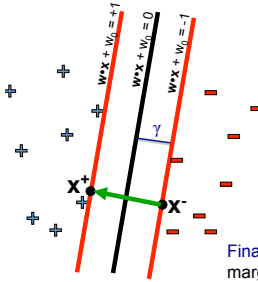
motivation for minimizing the norm of the weights

We can maximize the margin by minimizing $||w||_2$: $\gamma = \frac{||w||_2}{w \cdot w} = \frac{1}{||w||_2}$

Derivation:

$x^j = \bar{x}^j + \lambda \frac{w}{||w||_2}$ $\quad ||w||_2 = \sqrt{\sum_i w_i^2}$

Assume: x⁺ on positive line (y=1 intercept), x⁻ on negative (y=-1)



$$x^+ = x^- + 2\gamma \frac{w}{||w||^2}$$
$$w \cdot x^+ + w_0 = 1$$
$$w \cdot (x^- + 2\gamma \frac{w}{||w||_2}) + w_0 = 1$$
$$w \cdot x^- + w_0 + 2\gamma \frac{w \cdot w}{||w||_2} = 1$$
$$\gamma \frac{w \cdot w}{||w||_2} = 1 \qquad w \cdot w = \sum_i w_i^2 = ||w||_2^2$$
$$\gamma = \frac{||w||_2}{w \cdot w} = \frac{1}{||w||_2}$$

Final result: can maximize *constrained* margin by minimizing $||w||_2$!!!

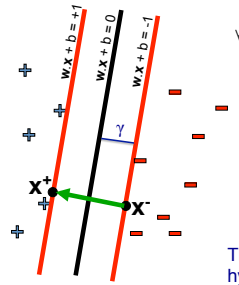
Intuitive explanation, after 2 key facts:

- Key fact #1: the bigger your $\boldsymbol{w}$ is, the <u>closer</u> the line $\boldsymbol{wx} = 1$ is to the line $\boldsymbol{wx} = 0$. Small distance between $\boldsymbol{wx} = 1$ and $\boldsymbol{wx} = 0$ translates to a small margin. Note that you could use any constant in place of 1.
- Key fact #2: The w_0 just shifts the black decision boundary line away from the origin.

Now we can explain why minimizing the norm of the weights leads to the largest margin.

- Imagine $w_0 = 0$, meaning the decision boundary goes through the origin.
- Now let $w = [2, 0]$ for simplicity. Where is the decision boundary? The decision boundary corresponds to $2x_0 + 0x_1 = 0$. That means the decision boundary is along $x_0 = 0$, which is a line through the x_1 (vertical) axis.
- But the point is that when you set $\boldsymbol{w} \cdot \boldsymbol{x} = c$ or some other constant. Key fact #1 says that the larger w is, the farther $\boldsymbol{w} \cdot \boldsymbol{x} = 0$ is from $\boldsymbol{w} \cdot \boldsymbol{x} = c$ In this case where $w_1 = 0$, you have $w_0 x_0 = c$, or $x_0 = c/w_0 = c/2$. If you want the = 1 line far from the decision boundary, you need to increase the distance between these lines. Since c is fixed, so the only way we can do this is to decrease w_0.
- The logic holds true for when w_1 is nonzero: you just want to minimize the size of the $\boldsymbol{w}$ vector. Minimizing the size of w is equivalent to minimizing $||w||_2$

Max margin using canonical hyperplanes



$$\max_{\gamma,w,w_0} \gamma$$
$$\forall j.y^j(w \cdot x^j + w_0) > \gamma$$
$$\gamma = \frac{1}{||w||_2}$$
$$\min_{w,w_0} \frac{1}{2}||w||_2^2$$
$$\forall j.y^j(w \cdot x^j + w_0) \geq 1$$

The assumption of canonical hyperplanes (at +1 and -1) changes the objective and the constraints!

The bottom line under the arrow keeps the w values from getting too small.

The 1/2 is just because we will take the derivative.
Previously we were just trying to maximizing margin. That was equal to minimizing the norm of w. Then we had a $\geq \gamma$ but now we have ≥ 1. Are we still maximizing the margin? Yes; we showed minimizing the norm of w leads to the max margin. **Part is pushing for bigger margin, part is keeping points from being inside the margin.**
The only points that have nonzero weights are the ones that are very close to the margin.
The line only changes if you remove points close to the line.

There are few points in the training data that are crucial for the line placement. Those points are called support vectors The rest of the points would not play a role. We call them support vectors b/c those are the points that can support the decision boundary.

SVM recipe

We want to minimize the norm subject to getting the predictions right:
$\min_{w,w_0} \frac{1}{2}||w||_2^2$ so that $\forall j.y^j(w \cdot x^j + w_0) \geq 1$
We do this with quadratic programming (QP). The decision boundary is defined by **support vectors**, which are data points on the canonical red lines. All the points that aren't on the red line are non-support vectors.

If your data is not linearly separable, you can add nonlinear features. These are called Kernels, and will be discussed later.

We were minimizing the $norm_2$ of w subject to constraint that $y_i(w \cdot x) > 1$

Balancing # of mistakes and $||w||_2^2$

We want to soften the constraint to allow for occasional mistakes. This doesn't happen by changing the 1 to 1/2. Reduce the number of points that have to be good. Introduce a new variable that says how wrong you are.

If your data isn't linearly separable, then you might need to allow your $||w||_2$ to be a little bigger and make a few mistakes. One option (not what we end up doing): $\min_{w,w_0} \frac{1}{2}||w||_2^2 + M$ (M = # of mistakes)
so that $\forall j.y^j(w \cdot x^j + w_0) \geq 1$

Hinge Loss

Zero loss for things you get correct, then penalty grows linearly if you make a mistake.

Want $1 - \xi = 0$ for when we are right. Use the margin: ξ is the perpendicular distance from point to hyperplane. Now our problem looks like:
min $||w||_2^2$ subject to constraint $y_i(w \cdot x_i) > 1 - \xi$ But we need to adjust it so we don't get a trivial solution. We need to adjust ξ.

One way to balance the number of mistakes and how big w is:
$\min_{w,w_0} \frac{1}{2}||w||_2^2 + C\sum_j \xi^j$
so that $\forall j.y^j(w \cdot x^j + w_0) \geq 1 - \xi^j$
C = strength of penalty.
ξ is size of error for each error. If the point is classified correctly, ξ is zero.

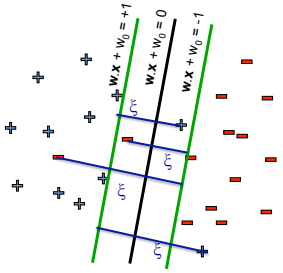
This is **"soft margin"** SVM.

We have a sum because we want to have small ξ Want both of them to be small. We want most of ξ values to be zero. At same time we also want norm of w to be small.

Note: we don't want to sum over ξ squared. We don't want extra big penalty if one is way off. We can use scaling to help.

That point on the hinge is sharp. Can replace it with ____ to make it softer. If you make no mistakes, then you get all zeros. That means your data is linearly separable.

When we want the dot product & shift $\geq 1 - \xi^j$, the 1 is for the green-line margin, and you are wrong by ξ amount.



Now we can tune our balance between asking for small weights $\boldsymbol{w}$ and asking for perfect classification.

- $C = \infty \rightarrow$ pressure to separate the data, even if the margin is skinny.
- $C = 0 \rightarrow$ fat margins over accuracy.

Use your training set to train C.

If there exists a + on the wrong side, penalize the distance to the margin or hyperplane? Typically use distance to the hyperplane. Just a difference of 1.

Note that if _____ ≥ 1, you don't care if the point is classified wrong. But if _____ < 1, you pay a linear penalty.
Hinge Loss:

$$\min_{w,w_0} \frac{1}{2}||w||_2^2 + C\sum_{j=1}^{N}[1 - y^j(w \cdot x^j + w_0)]$$

1^{st} term is regularization, 2^{nd} term is hinge loss.

Steps:

1. extract features
2. sweep parameters w/ k-fold validation.
3. solve SVM

Solve by differentiating and set equal to zero.
There is no closed form solution, but quadratic program is concave. (??)
Hinge loss is not differentiable (??), so gradient ascent is a little trickier.

This won't handle super messy tangled clouds, but hard SVM said don't want anything inside your margin. Softer: compute distance to hyperplane.

Logistic Regression to Minimize Loss

Logistic regression assumes $P(Y = 1|X = x) = \frac{exp(f(x))}{1+exp(f(x))}$
(For Logistic Regression, $f(x)$ was $w_0 + \sum_i w_i X_i$ and we had $Y = \{0, 1\}$)
Now we have $Y = \{-1, +1\}$.
To maximize data likelihood for $Y = \{-1, +1\}$:

$$P(y^i|x^i) = \frac{1}{1 + exp(-y^i f(x^i))}$$

$$\ln P(D_Y|D_{\boldsymbol{X}}, \boldsymbol{w}) = \sum_{j=1}^{N} \ln P(y^j|x^j, w)$$

plug in the P above

$$= -\sum_{i=1}^{N} \ln(1 + exp(-y^i f(x^i)))$$

Since $-\ln(z) = \ln(1/z)$ we get to minimize this (negative):

$$\sum_{i=1}^{N} \ln(1 + exp(-y^i f(x^i))) = \sum_{i=1}^{N} \ln(1 + exp(-y^i[w_0 + \sum w_i x_i]))$$

SVMs vs Regularized Logistic Regression

Logistic regression is a way to minimize loss. The probability of data point being class 1 is ... sigmoid function. Typically that function is a linear combination. We maximized the likelihood of the training data: maximize w. At the end you are minimizing the ln() below.

SVM objective: minimizing over w, for margin + hinge los. LR: loss is not hinge any more. Loss is exponential. But we are actually doing very similar shapes.

SVM Objective:

$$\arg\min_{\boldsymbol{w},w_0} \frac{1}{2}||w||_2^2 + C\sum_{j=1}^{N}[1 - y^j f(x^j)]_+$$

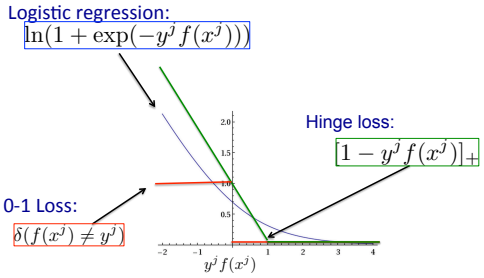
where $[x]_+ = max(x, 0)$
The $[x]_+$ is made for ease of notation. Just a mathematical way of writing that curve.

Logistic Regression Objective:

$$\arg\min_{\boldsymbol{w},w_0} \lambda||w||_2^2 + \sum_{j=1}^{N} ln(1 + \exp(-y^j f(x^j)))$$

Note that SVM and Logistic Regression have the same l_2 regularization term, but different error terms.

Graphing Loss vs Margin



We want to smoothly approximate 0/1 loss!

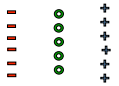
The red curve is true decision boundary. Green is hinge loss Blue is soft version. So the 2 are optimizing similar things. Trying to approximate the step function.

LR starts penalizing you a bit when you get close to the decision boundary. But once you hit the margin, SVM penalizes more. Then they flip back again after the lines extend past what's drawn on the left. SVM has hinge, LR has exponential or logistic function.

Multi-class SVMs

To do 3 classes, you need to learn 3 classifiers.

Can't just do $y = \arg\max_i w_i \cdot x$ for i classifiers. Wouldn't handle this:



Instead, we learn 3 classifiers for these 3 symbols:

1. + vs $\{O, -\}$, weights w_+
2. + vs $\{O, +\}$, weights w_-
3. + vs $\{+, -\}$, weights w_O

But to get it working for that set of 3 columns, we need additional constraints.
For each class:
for class y' that is not class y^j $(\forall y' \neq y^j)$:
And for all classes j $(\forall j)$:
$w^{y^j} \cdot x^j + w_0^{y^j} \geq w^{y'} \cdot x^j + w_o^{y'} + 1.$
$(\forall =$ "for all")
In plain english: ????.
??? Do I have the fact that j is for classes right? (Could j still be points?) ??

We can also introduce slack variables as before.

$$\min_{w,w_0} \sum_y ||w^y||_2^2 + C\sum_j \xi^j$$

$w^{y^j} \cdot x^j + w_0^{y^j} \geq w^{y'} \cdot x^j + w_o^{y'} + 1 - \xi^j.$
That's true for class y' that is not class y^j $(\forall y' \neq y^j)$, and all classes j $(\forall j)$ and for all $\xi^j > 0$

So you can do multiple classes in a one against all approach *or* a multiclass SVM approach.

Audio from week 7:

How can we change our SVM formulation to account for multiple classes? What do you do to your $y_i(w \cdot x) \geq 1 - \xi_i$? Want $\hat{y_i}$. If we get right w then all the blues should be on the left. so $w_{blue}x_{blue} > w_{anyotherx_i}$. For this green point, want the blue one to be responsible for the classification. Want the $w \cdot x$ for the right classifier to be better than any other by a margin of 1.

How is different than 1 vs 3 or 1 vs all? In 1 vs all, making independent predictions for categories. Here we are simultaneously learning them all at the same time. Training 3 ws at same time, together. For blue sample, w of blue times feature of that sample should be bigger than "1 + the margin" (??). Why is it + 1? Margin. Want all other points to be at least one away.

How many ξ values for n data point? for binary SVM, have n. # of samples. For multi-class, n times number of classifiers that we are training Each classifier has its own ξ. We could sum it up and consider 1 if we want. Both are options.
We can do multi-class with a soft margin as well Don't want any of the non-blue classifiers to claim it is w. Pushing the red, green lines away from blue points. Want w to not like any of the blues. At least not more than the blues by a margin of 1.

SVM info from other sources

http://axon.cs.byu.edu/Dan/478/misc/SVM.example.pd

The idea behind SVMs is to make use of a (nonlinear) mapping function that transforms data in input space to data in feature space in such a way as to render a problem linearly separable. The SVM then automatically discovers the optimal separating hyperplane (which, when mapped back into input space, can be a complex decision surface).

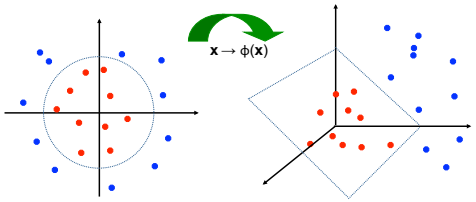
An Introduction to Statistical Learning

- shown to perform well in a variety of settings, and are often considered one of the best out of the box classifiers.
- The support vector machine is a generalization of a simple and intuitive classifier called the maximal margin classifier. Though the maximal margin classifier is elegant and simple, it cannot be applied to most data sets because it requires that the classes be separable by a linear boundary.
- People often loosely refer to the maximal margin classifier, the support vector classifier, and the support vector machine as support vector machines.
- Although the maximal margin classifier is often successful, it can also lead to overfitting when p is large.
- **support vector**: data points that support the maximal margin hyperplane in the sense that if these points were moved slightly then the maximal margin hyper- plane would move as well.
- In many cases no separating hyperplane exists, and so there is no maximal margin classifier.
- If you cant separate perfectly, you can settle for almost separating the classes using a so-called soft margin.
- Even if a separating hyperplane does exist, then there are instances in which a classifier based on a separating hyperplane might not be desirable.
- The fact that the maximal margin hyperplane is extremely sensitive to a change in a single observation suggests that it may have overfit the training data.
- The support vector classifier, sometimes called a **soft margin classifier**:
  Rather than seeking the largest possible margin so that every observation is not only on the correct side of the hyperplane but also on the correct side of the margin, we instead allow some observations to be on the incorrect side of the margin, or even the incorrect side of the hyperplane. (The margin is soft because it can be violated by some of the training observations.)
- An observation that lies strictly on the correct side of the margin does not affect the support vector classifier. Changing the position of that observation would not change the classifier at all, provided that its position remains on the correct side of the margin.
- The **support vector machine** (SVM) is an extension of the support vector classifier that results from enlarging the feature space in a specific way, using **kernels**.
- A **kernel** is a function that quantifies the similarity of two observations.
- What is the advantage of using a kernel rather than simply enlarging the feature space using functions of the original features? One advantage is computational, and it amounts to the fact that using kernels, one need only compute K(xi, xi?) for all (n choose 2) distinct pairs i, i?. This can be done without explicitly working in the enlarged feature space. This is important because in many applications of SVMs, the enlarged feature space is so large that computations are intractable.

Kernels

If the data is not linearly separable and/or you want a wiggly boundary, use kernels.
Can use for things you can't handle with slack variables.
Start w/ original feature space, use polynomial of degree d, and get a higher dimensional space then do a linear classifier in this space.

Can give you nonlinear boundaries (good) but the feature space can get really large really quickly.
Example mapping of data that is not linearly separable to a separable higher dimension space:



The decision boundary would be a circle.
If you can't figure out what features you should use, this is a good approach. You don't need the features themselves (??).
?? You just need the kernel/similarity? matrix.??
? gram matrix?
General idea:
If $\boldsymbol{x}$ is in R^n, then $\phi(\boldsymbol{x})$ is in R^m for $m > n$.
We can now learn feature weights $\boldsymbol{w}$ in R^m and predict using $y = sign(\boldsymbol{w} \cdot \phi(\boldsymbol{x}))$.

A linear function in the higher dimensional space will be non-linear in the original space.

Say you had a 100,000 points, each of which have a100-dimensional (binary) feature vector. The chances of being able to find a hyperplane that divides random assignments of binary values is very good. Almost all of those vertices are empty. So your chance of finding a desirable hyperplane increases dramatically. ??? Did I get this right? Seems like we need to be comparing a feature vector to a transformed feature vector. ???

Danger of Mapping to a Higher Dimensional Space:

The number of terms in a polynomial of degree d for m input features is $\binom{d+m-1}{d} = \frac{d+m-1}{d!(m-1)!}$.
This grows fast! For $d = 6$, $m = 100$ you get about 1.6 billion terms.
We are taking a dot product of m rows for the feature space times d columns of polynomial. But you also have terms for combinations of features. E.g.

$$\phi(x) = \begin{pmatrix} x_1 \\ \dots \\ x_n \\ x_1x_2 \\ x_1x_3 \\ \dots \\ e_{x_1} \\ \dots \end{pmatrix}$$

Efficient dot-product of polynomials

For polynomials of degree exactly d, the dot product in higher dimensional space can be written as a dot product in lower dimensional space.
For $m = 2$ (2 features):

d=1

$$\phi(u).\phi(v) = \begin{pmatrix} u_1 \\ u_2 \end{pmatrix} \cdot \begin{pmatrix} v_1 \\ v_2 \end{pmatrix} = u_1v_1 + u_2v_2 = u.v$$

d=2

$$\phi(u).\phi(v) = \begin{pmatrix} u_1^2 \\ u_1u_2 \\ u_2u_1 \\ u_2^2 \end{pmatrix} \cdot \begin{pmatrix} v_1^2 \\ v_1v_2 \\ v_2v_1 \\ v_2^2 \end{pmatrix} = u_1^2v_1^2 + 2u_1v_1u_2v_2 + u_2^2v_2^2 = (u_1v_1 + u_2v_2)^2 = (u.v)^2$$

note: here the . is a dot product, not horizontal space fill like a previous lecture.
u_1 is feature 1's value, and u_2 is feature 2's value.
Polynomial of exactly degree d means we don't have a constant bias term like in homework 3 with $[1, \sqrt{2}u, u^2]$
This is just factoring.

Proof not shown, but for any d:
$K(u, v) = \boldsymbol{\phi}(u).\boldsymbol{\phi}(v) = (u.v)^d$

The "Kernel Trick"

We like the idea of going to higher dim, but we don't like the idea of operating in it. Expensive.

With this trick, we won't ever compute a dot product in the expensive space. Not all ϕ satisfy, but there are big categories that do it. Need to be able to represent a higher dim space as one in a lower dim space. Usually exponentiation is used.

A **kernel function** defines a dot product in some feature space:

$$K(\boldsymbol{u}, \boldsymbol{v}) = \boldsymbol{\phi}(\boldsymbol{u}) \cdot \boldsymbol{\phi}(\boldsymbol{v})$$

Where $\boldsymbol{u}$, $\boldsymbol{v}$ are points in your training or test set, each with their own features.
Or $\boldsymbol{u}$ could be a data point and $\boldsymbol{v}$ could be the $\boldsymbol{w}$ of the hyperplane that you need to dot against to classify new points.
If $\boldsymbol{u}$, $\boldsymbol{v}$ each have two dimensions (2 features):
Example:
2-dimensional vectors $\mathbf{u}=[u_1\ u_2]$ and $\mathbf{v}=[v_1\ v_2]$; let $K(\mathbf{u},\mathbf{v})=(1 + \mathbf{u}\bullet\mathbf{v})^2$,
Need to show that $K(\mathbf{x}_i,\mathbf{x}_j)= \boldsymbol{\phi}(\mathbf{x}_i) \bullet \boldsymbol{\phi}(\mathbf{x}_j)$:

$K(\mathbf{u},\mathbf{v})=(1 + \mathbf{u}\bullet\mathbf{v})^2 = 1+ u_1^2v_1^2 + 2\,u_1v_1\,u_2v_2 + u_2^2v_2^2 + 2u_1v_1 + 2u_2v_2 =$
$= [1, u_1^2, \sqrt{2}\,u_1u_2,\ u_2^2, \sqrt{2}u_1, \sqrt{2}u_2] \bullet [1,\ v_1^2, \sqrt{2}v_1v_2, v_2^2, \sqrt{2}v_1, \sqrt{2}v_2] =$
$= \boldsymbol{\phi}(\mathbf{u}) \bullet \boldsymbol{\phi}(\mathbf{v})$, where $\boldsymbol{\phi}(\mathbf{x}) = [1,\ x_1^2, \sqrt{2}\,x_1x_2,\ x_2^2,\ \sqrt{2}x_1, \sqrt{2}x_2]$

Thus, a kernel function *implicitly* maps data to a high-dimensional space without the need to compute each $\boldsymbol{\phi}(\boldsymbol{x})$ explicitly.
If we had $d = 100$ instead, our savings would be huge.
Translation for this particular kernel:
We want to get the perks of the high dimensional space given by $\boldsymbol{\phi}(\boldsymbol{x}) = [1, x_1^2, \sqrt{2}x_1x_2, x_2^2, \sqrt{2}x_1, \sqrt{2}x_2]$ where $\boldsymbol{x}$ is either $\boldsymbol{u}$ or $\boldsymbol{v}$.
But this is computationally expensive because there are many terms in that thing.
If we take the dot product of two vectors $\boldsymbol{\phi}(\boldsymbol{u})$, $\boldsymbol{\phi}(\boldsymbol{v})$ we get some magic:

$$\begin{aligned} \boldsymbol{\phi}(\boldsymbol{u}) \cdot \boldsymbol{\phi}(\boldsymbol{v}) &= [1, u_1^2, \sqrt{2}u_1u_2, u_2^2, \sqrt{2}u_1, \sqrt{2}u_2] \cdot \quad \text{(line wrapped)} \\ &\quad [1, v_1^2, \sqrt{2}v_1v_2, v_2^2, \sqrt{2}v_1, \sqrt{2}v_2] \\ &= (1 + \boldsymbol{u} \cdot \boldsymbol{v})^2 \end{aligned}$$

Can do $(1 + \boldsymbol{u} \cdot \boldsymbol{v})^2$ with a dot product that only includes multiplying $[u_1, u_2]^T[v_1, v_2]$.
That's a lot less computation/memory expensive than the full dot product above.
If that leads to good separation, you are a happy machine learner!
This is true for other kernels in general.
$((1 + \boldsymbol{u} \cdot \boldsymbol{v})^2$ would take other forms for other cases).

It is essential that everything you want to do with your transformed vectors is representable by simple dot products. If you can't simplify it down to dot products of the original vector then there's no point.

"Kernel Trick" for the Perceptron

Intro to $w = \sum_k a^k \phi(x^k)$

If we wanted to apply a transformation of the data $\phi(x^i)$ to each of our x^i training points, we would do:
For $t = 1 \ldots T$, $i = 1, \ldots n$:

- $y = sign(w \cdot \phi(x^i))$
- if $y \neq y^i$: (if class wasn't predicted correctly)
  - $w = w + y^i \phi(x^i)$

We can re-write w as $\sum_j y^j x^j$ for one loop over the data, if j is the index of the points that were wrongly classified and thus contributed to the weights w. but we have multiple epochs: $t = 1 \ldots T$. The point might be classified wrong the first time but right the 2nd and 3rd times.
So we can use different notation to keep track of how each point adds to the weights depending on whether it was predicted correctly and whether it was a $y = 1$ or $y = -1$.
This looks like $w = \sum_k a^k \phi(x^k)$

We get a^k from the modified loop over the data points shown below. Note: we switch from index i over the data points to k because we might see data points that are identical in the training set. We don't need to keep track of those $\phi(x^k)$ points separately with separate k indices and hence a^k values if they are the same. Just keep summing up their signs as needed to modify w.
New protocol:
For $t = 1 \ldots T$, $i = 1, \ldots n$:

- $y = sign(w \cdot \phi(x^i))$
- if $y \neq y^i$: (if class wasn't predicted correctly)
  - $a^i += y^i$

For example, say you loop over data point k with $\phi(x^k) = [1, 2, 3]$ and label $y^k = -1$ three times. If you got it wrong the first two times and right the third time you would have this point contributing $w = -[1, 2, 3] - [1, 2, 3] + 0[1, 2, 3]$.
In the loop we would have had $a^k = -1 - 1$ for the two passes through that we got it wrong.
If we kept track of those a values for all i or k data points, then we get that thing above: $w = \sum_k a^k \phi(x^k)$

The Kernelized Perceptron

Protocol:
Set $a^i = 0$ for each example i
For $t = 1 \ldots T$, $i = 1, \ldots n$:

- $y = w \cdot \phi(x^i)$
  use $w = \sum_k a^k \phi(x^k)$
  $= sign((\sum_k a^k \phi(x^k)) \cdot \phi(x^i))$
  $= sign(\sum_k a^k K(x^k, x^i))$
- if $y \neq y^i$: (if class wasn't predicted correctly)
  - $a^i += y^i$

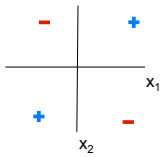
The points:

- We never compute the features explicitly.
  If x^i has 3 features, $\phi(x^i)$ might have 9 items such as polynomials.
  But we don't have to calculate those 9 features if they correspond to a Kernel.
  We might be able to use something like the simplification to $(1 + \boldsymbol{u} \cdot \boldsymbol{v})^2$ simplification just before this.
- We compute the dot products in "closed form":
  $K(u, v) = \phi(u) \cdot \phi(v)$
  A "closed form" expression is a mathematical expression that can be evaluated in a finite number of operations. (You can calculate it efficiently; it is possible you would have an infinite number of features.)

Kernelized Perceptron Example

- set a^i=0 for each example i
- For t=1..T, i=1..n:
  - $y = sign(\sum_k a^k K(x^k, x^i))$
  - if $y \neq y^i$
    - $a^i += y^i$

x_1	x_2	y
1	1	1
-1	1	-1
-1	-1	1
1	-1	-1



Initial:
- a = [a¹, a², a³, a⁴] = [0,0,0,0]

t=1,i=1
- $\Sigma_k a^k K(x^k, x^1)$ = 0x4+0x0+0x4+0x0 = 0, sign(0)=-1
- $a^1 += y^1 \rightarrow a^1 += 1$, new a= [1,0,0,0]

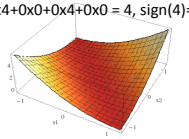
t=1,i=2
- $\Sigma_k a^k K(x^k, x^2)$ = 1x0+0x4+0x0+0x4 = 0, sign(0)=-1

t=1,i=3
- $\Sigma_k a^k K(x^k, x^3)$ = 1x4+0x0+0x4+0x0 = 4, sign(4)=1

t=1,i=4
- $\Sigma_k a^k K(x^k, x^4)$ = 1x0+0x4+0x0+0x4 = 0, sign(0)=-1

t=2,i=1
- $\Sigma_k a^k K(x^k, x^1)$ = 1x4+0x0+0x4+0x0 = 4, sign(4)=1

...



Converged!!!
- $y = \Sigma_k a^k K(x^k, x)$
  $= 1 \times K(x^1, x) + 0 \times K(x^2, x) + 0 \times K(x^3, x) + 0 \times K(x^4, x)$
  $= K(x^1, x)$
  $= K([1,1], x)$ (because $x^1 = [1,1]$)
  $= (x_1 + x_2)^2$ (because $K(u,v) = (u \bullet v)^2$)

K(u,v) = (u•v)²
e.g.,
K(x¹,x²)
= K([1,1],[-1,1])
= (1x-1+1x1)²
= 0

K	x^1	x^2	x^3	x^4
x^1	4	0	4	0
x^2	0	4	0	4
x^3	4	0	4	0
x^4	0	4	0	4

Common Kernels

Polynomials of degree exactly *d*

$$K(\mathbf{u}, \mathbf{v}) = (\mathbf{u} \cdot \mathbf{v})^d$$

Polynomials of degree up to *d*

$$K(\mathbf{u}, \mathbf{v}) = (\mathbf{u} \cdot \mathbf{v} + 1)^d$$

Gaussian kernels

$$K(\mathbf{u}, \mathbf{v}) = \exp\left(-\frac{||\mathbf{u} - \mathbf{v}||}{2\sigma^2}\right)$$

Sigmoid

$$K(\mathbf{u}, \mathbf{v}) = \tanh(\eta \mathbf{u} \cdot \mathbf{v} + \nu)$$

And many others: very active area of research!

How do we pick the kernel? You will get a poor result if you pick the wrong Kernel. There is no theory that tells you which kernel to use, so try a few and see which works bets.
For our class:

- try the polynomial of up to degree d.
- Gaussian is probably the most common.
- Sigmoid pushes one side down and the other side up.

Kernels: Overfitting

With Kernels we have a huge feature space, so over-fitting can happen.
It turns out, however, that it can be robust to overfitting.
People have spent a lot of time figuring out regularization for different Kernel methods.
Can do things like limiting the number of updates the Perceptron does.
SVMs have have a clearer story for avoiding overfitting.
Do keep in mind that **everything overfits sometimes!**
You can also control by:

- Choosing a better Kernel
- Varying parameters of the Kernel, such as the width of Gaussian.
- limit dimensionality, add regularization terms (e.g. L1, L2)

Kernels in Logistic Regression

Had:
$P(Y = 0 | \boldsymbol{X} = \boldsymbol{x}, \boldsymbol{w}, w_0) = \frac{1}{1 + \exp(w_0 + \boldsymbol{w} \cdot \boldsymbol{x})}$

We can define weights in terms of data points:
$\boldsymbol{w} = \sum_j \alpha^j \phi(\boldsymbol{x}^j)$

That leads to

$$P(Y = 0 | \boldsymbol{X} = \boldsymbol{x}, \boldsymbol{w}, w_0) = \frac{1}{1 + \exp(w_0 + \sum_j \alpha^j \phi(\boldsymbol{x}^j) \cdot \phi(\boldsymbol{x}))}$$
$$= \frac{1}{1 + \exp(w_0 + \sum_j \alpha^j K(\boldsymbol{x}^j, \boldsymbol{x})}$$

We can derive the gradient descent rule on α^j, w_0. ???
Similar tricks for all linear models: SVMs, etc.

Kernels: overfitting & number of parameters

Most of the time Kernels are better behaved for over-fitting (? than ___). But they are more expensive. Why? For the perceptron, we optimize $\boldsymbol{w}$. Each time, we update $\boldsymbol{w}$. The dimensionality of $\boldsymbol{w}$ is d or $d+1$, depending on how you consider it. With kernel we are optimizing $\boldsymbol{a}$. Dimensionality of $\boldsymbol{a}$ is way bigger than d, the dimensionality of the data. We are optimizing for way more dimensions.
We are explaining points based on other points in the data set How similar is a new point to x1, x2, xn. That's why we are worried about how many summations/iterations you will do.

Kernels for SVM = not easy

. For SVM using the kernel isn't as easy to apply. In Logistic regression it is easy to optimize. SVM requires you to write stuff down on paper.

Kernel Summary

Summary:

- Expand the data to a higher dimension
- might be expensive
- arrive at kernel trick: approximates dot prod on higher dim space by forming dot prod in original space
- we derived polynomial kernels.

Question: When I see a new instance in Kernelized perceptron, do I need to add it to my matrix?
Anser: no. If you do the matrix, you compute it all in the beginning.
Note: for homework we did it on-line.
Method:

- Separate training and test data.

- During training, give n data points all beforehand. Allows you to form matrix.
- Train to get a values and kernel matrix.
- Then done training.
- Go to testing phase. Give new instance and ask to classify.
- Don't need to add new instance to kernel matrix.
- Compute similarity to all of the points you have seen.

You do need to store the points you got right. Good to stick to sparse ___ for a .

Ensemble Methods

Vocab

- **decision tree stump**:
- **decision stub**: (used in lecture 10 pg 14: boosting) ?? horizontal or vertical line only?
- **axis aligned classifier**

Instead of learning a single classifier, learn many weak classifiers that are good at different parts of the data. The output class is a weighted vote of each classifier.

- classifiers that are most "sure" will vote with more conviction
- classifiers will be most "sure" about a particular part of the space.
- on average, these will do better than a single classifier.

This is better than breaking up the space into a bunch of sub-spaces and making single classifiers for each. If you had single classifiers for sub-spaces, you would be losing information about the surroundings. It is better to have all classifiers cover the whole space, but let them vote.

Bagging vs Boosting

Bagging:

- parallel ensemble: each model is built independently
- aim to decrease variance, not bias
- suitable for high variance low bias models (complex models)
- **samples are drawn with replacement**
- each model in the ensemble vote with equal weight
- an example of a tree based method is random forest, which develop fully grown trees (note that RF modifies the grown procedure to reduce the correlation between trees)

Boosting:

- **incrementally building an ensemble by training each new model instance to emphasize the training instances that previous models mis-classified.**
- aim to decrease bias, not variance
- suitable for low variance high bias models
- an example of a tree based method is gradient boosting
- In some cases, boosting has been shown to yield better accuracy than bagging, but it also tends to be more likely to over-fit the training data.
- AdaBoost is the most common

Bagging

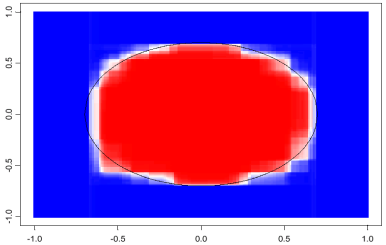
"Bagging" = Bootstrap AGGregation. For $i = 1, 2, \ldots, K$: (?? translate to english ??)

- $T_i \leftarrow$ randomly select M training instances with replacement.
- $h_i \leftarrow \text{learn}(T_i)$

Then combine the h_i together with uniform voting ($w_i = 1/K$ for all i).

Example: CART decision boundary

CART is a decision tree learning algorithm.
100 bagged trees: shades of blue/red indicate the strength of votes for particular classifications.
Picked random subsets of the data, and built classifiers. These classifiers are weighted!!
?? (not the strength of different classifiers.)
?? Is white an overall uncertain vote ??



Approximating the circle with a set of lines. Piecewise linear functions.
The more trees you have, the smoother the boundary will be.

Fighting the bias-variance tradeoff

Simple (a.k.a. weak) learners are good.
Examples of weak learners we can use:
Naive Bayes, logistic regression, perceptron, decision stumps, shallow decision trees, etc.
These learners have low variance; they don't usually overfit.
But simple (a.k.a. weak) learners are also bad:
They have high bias (high error), so you can't solve hard learning problems.
The solution: Boosting.

Boosting

TA summary:
We take a "weak" learner (e.g. a decision tree with depth 1), and learn the optimal classifier h_1. We then reweight the data points, increasing the relative weight of the misclassified points, and learn an optimal classifier h_2 which is heavily biased against mislabelling the same points. We repeat this process, and output a final classifier which uses a linear combination of the weak learners' predictions:
$H(x) = sgn(\sum_t \alpha_t h_t(x))$. Although the weak learners may have high-bias, and therefore simple decision boundaries, the "strong" classifier can be a complicated decision boundary.

- Combine weak classifiers to get very strong classifier. The weak classifiers only have to be slightly better than random on the training data. You end up with a very strong classifier. You can get zero training error.
- AdaBoost is the most common algorithm
- Similar to logistic regression:
  - both linear models. Boosting "learns" features.
  - similar loss functions
  - single optimization (Logistic Regression) versus incrementally improving classification (Boosting)
- boosting with a weak classifier is better than using a fancy classifier. A boosted version will always do better than the vanilla one.

An approach to calculate the output using several different models and then average the result using a weighted average approach. By combining the advantages and pitfalls of these approaches by varying your weighting formula you can come up with a good predictive force for a wider range of input data, using different narrowly tuned models.

Boosting is ensemble method.
The idea: given a weak learner, run it multiple times on (reweighted) training data, then let the learned classifiers vote.
On each iteration t:

- weight each training example by how incorrectly it was classified.
- learn a hypothesis: h_t
- Use strength α_t for this hypothesis.

Final classifier: $h(x) = sign\left(\sum_i \alpha_i h_i(x)\right)$

This is both useful in a practical sense and theoretically interesting.

Can use boosting with any kind of classifier.

Bagging

Stands for Bootstrap Aggregation.
The way decrease the variance of your prediction by generating additional data for training from your original dataset using combinations with repetitions to produce multisets of the same cardinality/size as your original data. By increasing the size of your training set you can't improve the model predictive force, but just decrease the variance, narrowly tuning the prediction to expected outcome.

Bagging allows encoding a curvy decision boundary with a bunch of weak classifiers.

By averaging a bunch of low bias, high variance functions, we can reduce the variance without significantly increasing the bias.
Making a strong classifier out of a set of weak classifiers.

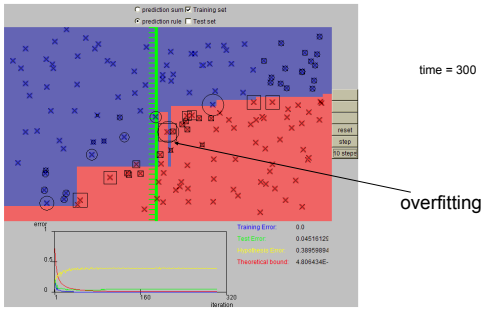
We should always avoid making hard decisions early. So don't put a lot of trust in classifiers early on in bagging. If we did, the result might not generalize well.
Instead, pay more attention to the ones you got wrong and less attention to the ones we got right.

We want instance based weighting, not classifier based weighting.

We want to have weights for each instance.
Protocol:

- Start with uniform weights. Each value is equally important.
- Train classifier 1. After this, instances are not equally important.
- The points classifier 1 got correctly are now considered less important.
- We want classifier 2 to be more worried about the ones that we got wrong.

Example



After making 300 classifiers, we can get a step-like decision boundary. Stopping at 100 would have been better; it is not over-fit (not shown). The blue, green, yellow, and red lines:

- blue = training error = same as usual. Randomly choose a subset to hold-out for testing
- green = test error = same as usual. Randomly choose a subset to hold-out for testing
- yellow = hypothesis error = the error of the new weak learner generated on that step (i.e. a decision stump)
  "Last classifier we added was 38% wrong. " for time = 100 when Hypothesis error said 0.38
- red = theoretical bound = the bound on the training error, derived in the later slides

Learning from weighted data

Consider a weighted data set:

- $D(i)$ is the weight of the i^{th} training example/point (not classifier!). It gets bigger each time point i is predicted incorrectly. It doesn't get smaller, but you normalize (see below).
  Point = $(\boldsymbol{x}^i, \boldsymbol{y}^i)$
- interpretations:
  - the i^{th} training example counts as if it occurred $D(i)$ times
  - these extra counts mean that if we were to "resample" data, we would get more samples of "heavier" data points.
- Now we always do weighted calculations:
  - e.g. MLE for Naive Bayes
  - redefine Count(Y=y) to be a weighted count:
    $Count(Y = y) = \sum_{j=1}^{n} D(j)\delta(Y^j = y)$
  - ?? Is this counting the number of points we have right?? No.. what is it counting?
  - if point j has been wrong many times before, it becomes more important, as reflected by $D(j)$ being large.
  - setting $D(j) = 1$ (or any constant value!) for all j recreates the unweighted case.

Note, we can use decision stumps for boosting. Can also use logistic regression, but it isn't as easy to show as our clas stump example.

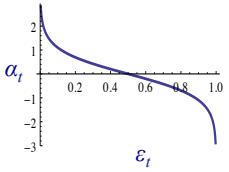
That's just about weighing the samples. How do we weight the classifiers? We want to weight across <u>all</u> data points. Use α to allow classifiers to have different votes.

Algorithm: Binary case

<u>Given</u>: points $(x^1, y^1), \ldots, x^m, y^m$.
For this case, $x^i \in \mathbb{R}$, and binary labels: $y^i \in \{-1, +1\}$
<u>Initialize</u>: $D_1(i) = 1/m$ for $i = 1, ;m$.
For $t = 1, \ldots, T$:

- Train base classifier $h_t(x)$ using D_t
- Chose the weight of importance for this classifier, α_t.
  Note that this comes after training the classifier.
  There are many possibilities for choosing α, which are discussed later.
- Update, for $i = 1 \ldots m$:
  $D_{t+1}(i) \propto D_t(i) exp(-\alpha_t y^i h_t(x^i))$
  with normalization constant $\sum_{i=1}^{M} D_t(i) \exp(-\alpha_t y^i h_t(x^i))$
  - The D is getting reweighted for the next round.
    What's happening inside:
    If $y^i h_t(x^i) > 0$, h_i was correct.
    But if $y^i h_t(x^i) < 0$, h_i was wrong.
    You multiply by α_t, which can flip the sign inside the exp().
    If h_i is correct and $\alpha > 0$, then $D_{t+1}(i) < D_t(i)$.
    But if h_i is wrong and $\alpha > 0$, then $D_{t+1}(i) > D_t(i)$.
- Output a final classifier: $h = sign\left(\sum_{i=1}^{T} \alpha_t h_t(x)\right)$
  This is a linear sum of "base" (weak) classifier outputs.
  Note that D is no longer in the picture.

If you had two classifiers, your result would be $\alpha_1 h_1 + \alpha_2 h_2$
How to chose α:
First calculate $\epsilon_t = \sum_{i=1}^{m} D_t(i)\delta(h_t(x^i \neq y^i)))$
This is the error of h_t, weighted by D_t
Then use $\alpha_t = \frac{1}{2} \ln\left(\frac{1-\epsilon_t}{\epsilon_t}\right)$ (derived below)
This transforms alpha according to:



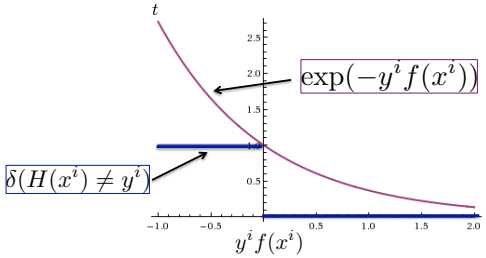
- no errors: $\epsilon_t = 0 \rightarrow \alpha_t = \infty$
- all errors: $\epsilon_t = 1 \rightarrow \alpha_t = -\infty$
- random: $\epsilon_t = 0.5 \rightarrow \alpha_t = 0$

Truncate at 2 or 3. Don't want to allow weight = infinity.
Think of this like NB with weighting.
<u>Why would we want/have negative weights on classifiers?</u>
A classifier that does worse than chance is backwards. If it is worse than chance, flip the sign then use it. A classifier that is wrong 90% of the time is a good classifier. A truly random classifier does not provide information. Thow those away.

If you get something right a bunch of times, the weight should converge to zero.
We want to run this loop until it converges. But if we run it too long, it will overfit.

How to chose α_t for hypothesis h_t?

It would be cool to find the α_t that works best for that classifier, take derivative with respect to α_t, and find an α that minimizes error.
We can't optimize the training set error, but we can minimize a bound on it.
$\sum_{i=1}^{m} \delta(H(x^i) \neq y^i) \leq \sum_{i=1}^{m} D_t(i) \exp(-y^i f(x^i))$
where $f(x) = \sum_t \alpha_t h_t(x)$; $H(x) = sign(f(x))$



We know the training set error (# of instances wrong) it is bounded by the sum on the right. Why? The left side of the equality is the left piece of the step function. The exponential function of label*prediction has the expo curve. Note that **the exponential curve is always above the step function**. So we can use expo as the upper bound. This is kind of like logistic regression: same form.

You can choose α_t to minimize the error bound.

Note that each classifier is independent of alpha. Each classifier is already a classifier by itself. We want to train the weighted combinations of classifiers. Note we are not directly using the error. We are using a function that maps the error to a # we can use. That's the role of the alpha vs epsilon curve. We liked its properties. There are two parameters in the exp(): h and α. You can optimize for h and alpha together. We have a way to make the classifier sand learn the weights at the same time. We might still learn the classifiers first. For those of you who care about joint optimization, you can flip between optimizing the two.

Idea: choose α_t to minimize a bound on training error!

$$\frac{1}{m}\sum_{i=1}^{m} \delta(H(x^i) \neq y^i) \leq \frac{1}{m}\sum_{i=1}^{m} D_t(i) \exp(-y^i f(x^i)) = \prod_t Z_t$$

Where
$f(x) = \sum_t \alpha_t h_t(x); H(x) = sign(f(x))$
And
$Z_t = \sum_{i=1}^{m} D_t(i) \exp(-\alpha_t y^i h_t(x^i))$

This equality isn't obvious! Can be shown with algebra (telescoping sums)!

If we minimize $\prod_t Z_t$, we minimize our training error!!!

- We can tighten this bound greedily, by choosing α_t and h_t on each iteration to minimize Z_t.
- h_t is estimated as a black box, but can we solve for α_t?

We can squeeze this bound by choosing α_t on each iteration to minimize Z_t.

$$Z_t = \sum_{i=1}^{m} D_t(i) \exp(-\alpha_t y^i h_t(x^i))$$

$$\epsilon_t = \sum_{i=1}^{m} D_t(i) \delta(h_t(x^i) \neq y^i)$$

For boolean Y: differentiate, set equal to 0, there is a closed form solution! [Freund & Schapire '97]:

$$\alpha_t = \frac{1}{2} \ln \left(\frac{1-\epsilon_t}{\epsilon_t} \right)$$

Note that our D values are not going up by more than 1 each time, like the intro to the concept showed. There is an exponential term that generally shrinks the sum of the D values to 1 (at least in the class example).

After the third iteration for our class example:

Initialize: $D_1(i) = 1/m, \text{ for } i = 1, \ldots, m$

For t=1…T:

- Train base classifier $h_t(x)$ using D_t
- Choose α_t: $\epsilon_t = \sum_{i=1}^{m} D_t(i) \delta(h_t(x^i) \neq y^i)$, $\alpha_t = \frac{1}{2} \ln \left(\frac{1-\epsilon_t}{\epsilon_t} \right)$
- Update, for i=1..m: $D_{t+1}(i) \propto D_t(i) \exp(-\alpha_t y^i h_t(x^i))$

Output final classifier: $H(x) = \text{sign} \left(\sum_{i=1}^{T} \alpha_t h_t(x) \right)$

x_1	y
-1	1
0	-1
1	1

- D_3 = [D_3(1), D_3(2), D_3(3)] = [0.33,0.5,0.17]

t=3:

- Train stub [work omitted; different stub because of new data weights D; breaking ties opportunistically (will discuss at end)]
  - h_3(x)=+1 if x_1<-0.5, -1 otherwise
- $\epsilon_3 = \Sigma_i D_3(i)\ \delta(h_3(x^i) \neq y^i)$ = 0.33×0+0.5×0+0.17×1=0.17
- α_3=(1/2) ln((1-ϵ_3)/ϵ_3)=0.5×ln(4.88)= 0.79
- Stop!!! How did we know to stop?

H(x) = sign(0.35×h_1(x)+0.55×h_2(x)+0.79×h_3(x))

- h_1(x)=+1 if x_1>0.5, -1 otherwise
- h_2(x)=+1 if x_1<1.5, -1 otherwise
- h_3(x)=+1 if x_1<-0.5, -1 otherwise

The ϵ values are dependent on the previous iteration's D values, but the current iterations h classifier.
Once you find your new ϵ from your old Ds and new h, you can find α and thus your new Ds for the next round.
?? Is it typical to have your alpha weights grow ?? Can I explain what is driving this ??

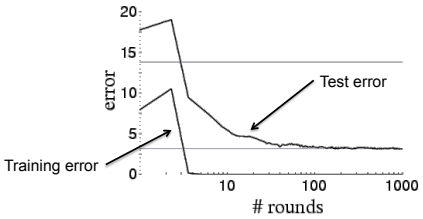
The D after normalization is a probability.

Assembling weak classifiers

If each classifier is (at least slightly) better than random: $\epsilon_t < 0.5$:
Another bound on error:

$$\frac{1}{m} \sum_{i=1}^{m} \delta(H(x^i) \neq y^i) \leq \prod_t Z_t \leq \exp \left(-2 \sum_{t=1}^{T} (\frac{1}{2} - \epsilon_t)^2 \right)$$

This implies training error will reach zero exponentially fast.
The error is bounded by an exponential of how far you are from random.
Note that it isn't too hard to achieve better than random training error, especially for binary classification.
Boosting is powerful!
Note that test error can continue to decrease after training error goes to zero.



Also in this figure, the lack of up-tick at the right shows that boosting does not overfit.

Using your classifier

Say you have a trained/converged classifier. That means you have hs and αs : weak classifiers and their corresponding weights. Plug in new x into $H(x)$

How do we know when to stop?
If the training error keeps getting better we keep going. This could lead to over-fitting, but it turns out boosting is robust to overfitting. (We didn't actually conclude when to stop in class.)
There was a theory (Freund & Schapire, 1996) that suggested

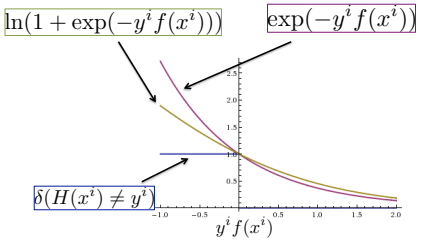
$$error_{true}(H) \leq error_{train}(H) + \tilde{\mathcal{O}} \left(\sqrt{\frac{Td}{m}} \right)$$

Suggests you don't want to use complicated classifiers in your boosting. d would get high, we could end up over-fitting.
T: number of boosting rounds. Higher $T \rightarrow$ looser bound.
d: VC dimension of weak learner. Measures complexity of classifier.
m: number of training examples. More data $\rightarrow$ tighter bound.

It turns out boosting is robust to overfitting. The test set error decreases even after the training error reaches zero. So this theory doesn't hold.

Boosting and Logistic Regression

Both smooth approximations of 0/1 step loss.



Logistic regression:

- Minimize loss fn $\sum_{i=1}^{m} \ln(1 + \exp(-y^i f(x^i)))$
- Define $f(x) = \sum_j w_j x_j$ where each feature x_j is predefined
- Jointly optimize parameters $w_0, w_1, \ldots w_n$ via gradient ascent.

Boosting:

- Minimize loss fn $\sum_{i=1}^{m} \exp(-y^i f(x^i))$
- Define $f(x) = \sum_t \alpha_t h_t(x)$ where $h_t(x)$ learned to fit data
- Weights α_j learned incrementally (new one for each training pass)

As noted above: Similar to logistic regression:

- both linear models. Boosting "learns" features.
- similar loss functions
- single optimization (Logistic Regression) versus incrementally improving

Clustering

- Unsupervised learning: detect patterns in unlabeled data. Sometimes labels are too expensive, unclear, etc. to get them. Examples:
  - group e-mails or search results
  - find categories of customers
  - detect anomalous program executions
- Useful when you don't know what you are looking for.
- Requires a definition of "similar". One option: small (squared) euclidean distance.
- You can label then use the clusters, or use the clusters for the next level of analysis.

K-Means

- An iterative clustering algorithm.
- No step size. Discrete optimization.
- Hard assignments. Each point gets classified by one and only one cluster.
- will converge, but may converge on local (not global) optimum.
  - Every time you start the algorithm, you could end up n a different place.
  - Can run it a bunch of times.
  - you are running a non-convex optimization: your final output is dependent on your initialization.
- you have to chose a number of clusters.
- Objective: minimize the distances between each point and closest center.
- You want your output to have a large distance between clusters and a small distance between points in a cluster. (intra vs inter cluster distance). You want it to latch onto clumps of the data that are far apart from each other.
  - intra:
    E.g. measure $|x_i - c_i|_2^2$ for each cluster.
  - inter:
    Dist between closest two points in different clusters.
    Distance between means.
    Standard deviation of cluster distances.

Pick K random points as cluster means: $c^1, \ldots, c^k$.
Alternate:

- Assign each example x^i to the mean c^i that is closest to it
- Set each mean c^i to the average of its assigned points.

Stop when no points' assignments change.
Minimizing a loss that is a function of the points, assignments, and means:

$$L(\{x*i\}, \{a*j\}, \{c*k\}) = \sum_i dist(x^i, c^{a^i})$$

Coordinate gradient descent on L.
More formally:

- Data: $\{x^j | j = 1 \ldots n\}$
- For $t = 1 \ldots T$: (or stop if assignments don't change):
  Fix means (c) while you change the assignments (a):

– for $j = 1 \ldots n$: (recompute cluster assignments):

$$a^j = \arg\min_i dist(x^j, c^i)$$

- fix assignments (a) while you change the means (c):
  for $j = 1 \ldots k$: (recompute cluster centers)

$$c^j = \frac{1}{|\{i|a^i = j\}|} \sum_{\{i|a^i=j\}} x^i$$

Note: the point y with minimum squared Euclidean distance to a set of points x is their mean

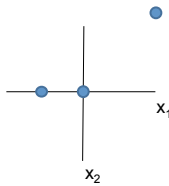
Pick K random cluster centers, $c^1 \ldots c^k$
For t=1..T:
- for j = 1.. n: [recompute assignments]

$$a^j = \arg\min_i dist(x^j, c^i)$$

- for j= 1…k: [recompute cluster centers]

$$c^j = \frac{1}{|\{i|a^i = j\}|} \sum_{\{i|a^i=j\}} x^i$$

x_1	x_2
-1	0
0	0
2	2



$$dist(x, x') = \sum_i (x_i - x'_i)^2$$

Random cluster means:
- c^1=[-1,0], c^2=[0,0]

t=0:

d(x^i,c^j)	x^1	x^2	x^3
c^1	0	1	13
c^2	1	0	8

- a^1 = argmin$_j$ dist(x^1,c^j) = 1
- a^2 = argmin$_j$ dist(x^2,c^j) = 2
- a^3 = argmin$_j$ dist(x^3,c^j) = 2
- c^1 = (1/1) * [-1,0] = [-1,0]
- c^2 = (1/2) * ([0,0]+[2,2]) = [1,1]

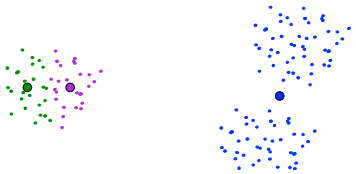
t=1:

d(x^i,c^j)	x^1	x^2	x^3
c^1	0	1	13
c^2	4	4	18

- a^1 = argmin$_j$ dist(x^1,c^j) = 1
- a^2 = argmin$_j$ dist(x^2,c^j) = 1
- a^3 = argmin$_j$ dist(x^3,c^j) = 2
- c^1 = (1/2) * ([-1,0]+[0,0]) = [-0.5,0]
- c^2 = (1/1) * ([2,2]) = [2,1]

t=2:
- Stop!! (cluster assignments a^j won't change in next round; you can verify!)

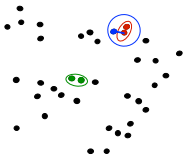
K-Means gets stuck in local optima.



Agglomerative Clustering

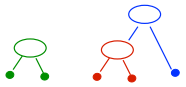
Will not converge to a global optimum.
Will not always find the true patterns in the data.
It has to deal with the issue of local optima just like k-means (which can prevent you from finding the "true" patterns)
First merge very similar instances.
Then incrementally build larger clusters out of smaller clusters.
Limiting the number of pairs of pairs, we can control the number of clusters.

- Distance = 0 means each point is its own cluster
- Distances = infinity –¿ all in one cluster.

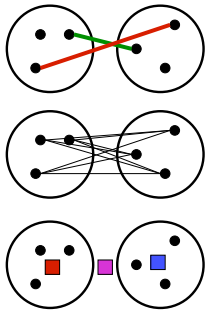


Algorithm:

- Maintain a set of clusters.
- Initially each instance is its own cluster
- Repeat:
  - pick the two closest clusters
  - merge them into a new cluster
  - stop when there is only one cluster left.
- produces not one clustering, but a family of clusterings represented by a dendogram.



- How should we define "closest" for clusters with multiple elements?
- Many options:
  - Closest pair (single-link clustering)
  - Farthest pair (complete-link clustering)
  - Average of all pairs
  - Ward's method (min variance, like k-means)
- Different choices create different clustering behaviors



The intra-cluster distance: a metric of how well the clustering algorithm works

$$S_1 = \sum_{j=1}^{3} \sum_i |x_i - x_c|_2^2 + \sum_{i,j} |c_j, c_i|_2^2$$

You have to be extremely lucky to find a data set where the result isn't dependent on the start.
Can run it a bunch of times. For each pair of points, we have a vote: Do they belong to the same cluster?
Look at score from score of clustering algorithm
We get a full graph where edge scores are "do they belong to the same cluster" (and more audio I missed?)
Then you need to find out which components are connected.
For each pair of points, we have an edge distance.
Within-cluster edges should be strong edges.
This strength should be common across clustering results.
Cut the graph into three pieces.
The score of the cut is the summation of the edges you break.
Cutting edges with small scores is good.

Probabilistic Clustering

- Can use a probabilistic model that allows cluster overlaps, clusters of different sizes, etc.
- You can tell a generative story for the data.
  $P(X|Y)P(Y)$ is common.
- The challenge: estimate model parameters without labeled data.

Gaussian Mixture Models

- We have clumps of data. Each clump is described with a gaussian.
- Like softening k-means. You belong to cluster 1 with a score of 0.1, cluster 2 with score of 0.3, cluster 3 with score of 0.6
- Think of clusters as probabilistic.
- Assume m-dimensional data points.
- P(Y) is still multinomial, with k classes.
- $P(\mathbb{X}|Y = i), i = 1 \ldots k$ are k multivariate Gaussians.
  - mean μ_i is a m-dimensional vector.
  - variance Σ_i is an m by m matrix.
  - $|x|$ is the determinant of matrix x.

$$P(X = x|Y = i) = \frac{1}{\sqrt{(2\pi)^m |\Sigma_i|}} \exp\left(\frac{1}{2}(x - \mu_i)^T \Sigma_i^{-1}(x - \mu_i)\right)$$

GMM is not Gaussian Naive Bayes

(We did GNB before logistic regression)
Gaussian Naive Bayes : multinomial over clusters Y, Gaussian over each X_i given Y:

$$P(Y_i = y_k) = \theta_k$$

(Again, θ is the model parameters)

$$P(X_i = x|Y = y_k) = \frac{1}{\sigma_{ik}\sqrt{2\pi}} \exp\left(\frac{-(x - \mu_{ik}^2}{2\sigma_{ik}^2}\right)$$

? Would assume the input dimensions X_i do not co-vary.
If the input dimensions X_i do co-vary, we can use Gaussian Mixture Models.

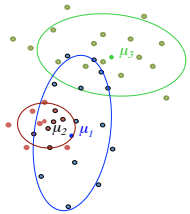
Gaussian Mixture Model Assumption

We want to do something like MLE but now we have multiple Gaussians.
You don't know which label should be used for each data point(which is red, blue, green).
Need to guess k Gaussians without knowing the μs.
You can marginalize:
Model probability without knowing who belongs to who: marginalize over all possible y values.
You are estimating $P(X|Y)$, but you don't know Y.
You can get rid of Y and get $P(X)$ by summing over y_i.
Get Y out of the equation by summing over all possible values.
If it was a probability table, we would be losing a column.

- $P(Y)$: there are k components
- $P(X|Y)$: each component generates data from a Gaussian with mean μ_i and covariance matrix Σ_i
- Assume each of the features are independent of each other.
  Then can write down $P(X|Y)$ as prod of $P(X_i|Y)$.
  Each of them will be a Gaussian distribution.
- Can encode the whole $P(X|Y)$ with a multi-dimensional gaussian.
  For 2D data, this gives a circle.
  For 3D data, this gives a bump.
  When we go to 100-dim space, we also have a mu.
  Distribution is 100-dimensional.
  Sigma in 100-dim space is 100 by 100 covariance matrix.

Each data point is sampled from a **generative process**

- Pick a component at random:
  choose component i with probability $P(y = i)$
- Datapoint $\sim N(\mu_i, \Sigma_i)$



Supervised MLE for GMM

(Detour/review)
How do we estimate parameters for Gaussian Mixtures with fully supervised data?
Define objective and solve optimization:
From above:

$$P(X = x|Y = i) = \frac{1}{\sqrt{(2\pi)^m|\Sigma_i|}} \exp\left(\frac{1}{2}(x-\mu_i)^T\Sigma_i^{-1}(x-\mu_i)\right)$$

And we know $\mu_{ML} = \frac{1}{n}\sum_{i=1}^{n} x^i$ and

$\Sigma_{ML} = \frac{1}{n}\sum_{i=1}^{n}(x^i-\mu_{ML})(x^i-\mu_{ML})^T$

But we don't know Y, so we can't do that.
Instead, we maximize the marginal likelihood. (marginal means a variable is integrated out).

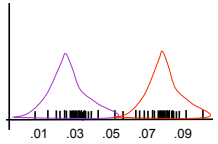
$$\arg\max_\theta \prod_i P(x^j;\theta) = \arg\max \prod_j \sum_{i=1}^{k} P(y^j = i, x^j;\theta)$$

This is always a hard problem.
There is usually no closed form solution.
Even when $P(X,Y;\theta)$ is convex, $P(X;\theta)$ generally isn't.
For all but the simplest $P(X;\theta)$, we will also have to do gradient ascent, in a big messy space with lots of local optima.

Simple GMM example: learn means only

Consider:

- 1D data, m points
- Mixture of k=2 Gaussians
- Variances fixed to σ=1
- Dist'n over classes is uniform
- Need to estimate μ_1 and μ_2



$$\prod_{j=1}^{n}\sum_{i=1}^{k} P(X = x^j, Y = i) \propto \prod_{j=1}^{n}\sum_{i=1}^{k} \exp\left(-\frac{1}{2\sigma^2}(x^j-\mu_i)^2\right)$$

We solve this using EM below.

Learning general mixtures of Gaussian

$$P(X = x|Y = i) = \frac{1}{\sqrt{(2\pi)^m|\Sigma_i|}} \exp\left(-\frac{1}{2}(x-\mu_i)^T\Sigma_i^{-1}(x-\mu_i)\right)$$

- Marginal likelihood, for data {x^j | j = 1..n}:

$$\prod_{j=1}^{n} P(x^j) = \prod_{j=1}^{n}\sum_i P(X = x^j, Y = i) = \prod_{j=1}^{n}\sum_i P(X = x^j|Y = i)P(Y = i)$$
$$= \prod_{j=1}^{n}\sum_i \frac{1}{\sqrt{(2\pi)^m|\Sigma_i|}} \exp\left(-\frac{1}{2}(x^j-\mu_i)^T\Sigma_i^{-1}(x^j-\mu_i)\right) P(Y = i)$$

- Need to differentiate and solve for μ_i, Σ_i, and P(Y=i) for i=1..k
- There will be no closed for solution, gradient is complex, lots of local optimum
- Wouldn't it be nice if there was a better way!

EM for GMM: learn means of 1D data

Iterate: On the *t*'th iteration let our estimates be

$\theta_t = \{\mu_1^{(t)}, \mu_2^{(t)} \dots \mu_k^{(t)}\}$

E-step

Compute "expected" classes of all datapoints

$$p(y = i|x^j;\theta_t) \propto \exp\left(-\frac{1}{2\sigma^2}(x^j-\mu_i)^2\right)$$

M-step

Compute most likely new **μ**s given class expectations, by doing weighted ML estimates:

$$\mu_i = \frac{\sum_{j=1}^{m} p(y = i|x^j;\theta_t)x^j}{\sum_{j=1}^{m} p(y = i|x^j;\theta_t)}$$

Pick K random cluster centers, $\mu_1 \dots \mu_k$
For t=1..T:

- E step:

$$p(y = i|x^j;\theta_t) \propto \exp\left(-\frac{1}{2\sigma^2}(x^j-\mu_i)^2\right)$$

- M step:

$$\mu_i = \frac{\sum_{j=1}^{m} p(y = i|x^j;\theta_t)x^j}{\sum_{j=1}^{m} p(y = i|x^j;\theta_t)}$$

x_1
-1
0
2

Initialization, random means and σ=1:

- μ_1=-1, μ_2=0

t=0:

- P(y=1|x^1) α exp(-0.5×(-1+1)²) = 1
- P(y=2|x^1) α exp(-0.5×(-1-0)²) = 0.6
  - P(y=1|x^1) = 0.63, P(y=2|x^1)=0.37
- P(y=1|x^2) α exp(-0.5×(0+1)²) = 0.6
- P(y=2|x^2) α exp(-0.5×(0-0)²) = 1
  - P(y=1|x^2) = 0.37, P(y=2|x^2)=0.63
- P(y=1|x^3) α exp(-0.5×(2+1)²) = 0.07
- P(y=2|x^3) α exp(-0.5×(2-0)²) = 0.93
  - P(y=1|x^3) = 0.01, P(y=2|x^3)=0.93
- μ^1 = (0.63×-1+ 0.37×0 + 0.07×2) / (0.63 + 0.37 + 0.07) = -0.45
- μ^2 = (0.37×-1+ 0.67×0 + 0.93×2) / (0.37+0.67+0.93)=0.75

t=1:

- learning continues, when do we stop?

EM for GMM in general

Iterate: On the *t*'th iteration let our estimates be, for y with k classes

$\theta_t = \{\mu_1 \dots \mu_k, \Sigma_1 \dots \Sigma_k, p_1,\dots,p_k\}$

E-step

Compute "expected" classes of all datapoints for each class

$$P(y = i|x^j;\theta_t) \propto \frac{1}{\sqrt{(2\pi)^m|\Sigma_i|}} \exp\left(-\frac{1}{2}(x^j-\mu_i)^T\Sigma_i^{-1}(x^j-\mu_i)\right) p_i$$

Evaluate a Gaussian at x^j

M-step

Compute weighted MLE for **μ** and **Σ** given expected classes above

$$\mu_i = \frac{\sum_{j=1}^{m} p(y = i|x^j;\theta_t)x^j}{\sum_{j=1}^{m} p(y = i|x^j;\theta_t)} \qquad \Sigma_i = \frac{\sum_{j=1}^{m} p(y = i|x^j;\theta_t)(x^j-\mu_i)(x^j-\mu_i)^T}{\sum_{j=1}^{m} p(y = i|x^j;\theta_t)}$$

$$p_i = \frac{1}{m}\sum_{j=1}^{m} p(y = i|x^j;\theta_t)$$

EM with hard assignments, and only learning means → K-means

E-step / Compute cluster assignment

Compute "expected" classes → set most likely class

$$p(y = i|x^j;\theta_t) = \exp\left(-\frac{1}{2\sigma^2}\|x^j-\mu_i\|_2^2\right) \Rightarrow a^i = \arg\min_j dist(x^i, c^j)$$

$dist(x,x') = \|x-x'\|_2^2$

M-step / Recompute cluster mean

Compute most likely new **μ**s → averages over hard assignments

$$\mu_i = \frac{\sum_{j=1}^{m} p(y = i|x^j;\theta_t)x^j}{\sum_{j=1}^{m} p(y = i|x^j;\theta_t)} \Rightarrow c^i = \frac{1}{|\{j|a^j = i\}|}\sum_{\{j|a^j=i\}} x^j$$

With hard assignments and unit variance, EM is equivalent to k-means clustering algorithm!!!

Expectation Maximization

A clever method for maximizing marginal likelihood, where you alternate between computing an expectation and a maximization.
It is not magic: it is still optimizing a non-convex function with lots of local optima. The computations are just easier.

- as in GMM, the objective is:

$$\arg\max_\theta \prod_i P(x^j;\theta) = \arg\max \prod_j \sum_{i=1}^{k} P(y^j = i, x^j;\theta)$$

- **E step:** Compute the expectations to "fill in" the missing y values according to the current parameters.
  For all examples j and values i for y, compute: $P(y^j = i|x^j,\theta)$.
- **M step:** Re-estimate the parameters with "weighted" MLE estimates: Set

$$\theta = \arg\max_\theta \sum_j \sum_{i=1}^{k} P(y^j = i|x^j,\theta) log P(y^j = i, x^j|\theta)$$

- this is especially useful when the E and M steps have closed form solutions.

General Vocab

- **classification** - ??? Finding a f that converts X to Y where Y are categorical. (Not regression).
- **supervised learning** - at training time we are given a set of features with discrete class labels.
- **held-out data**: the terms "held-out" and "validation" are usually synonymous
- **hypothesis space**: ? E.g. binomial distribution for coin flip.
- **prediction error**: measure of fit (?)
- **regularization**: a process of introducing additional information in order to solve an ill-posed problem or to prevent overfitting
- **norm**. A scalar (not vector!) measure of distance between two vectors. You can divide by this scalar to normalize, but that is just one use case. You can also use either norm as a regularization term.
- **L1** norm. Just add the absolute values of the components.
- **L2** norm. Can be used to regularize, or to normalizing features.
  - Euclidean vector length. Pythagoras style.
  - You can do l_2 normalization for a feature vector to get a unit vector: Convert x to $\hat{x}$ so that if you form $||\hat{x}||_2^2 = 1$ Can also do l_1
- **L2** distance: Pythagoras distance between two vectors.
- **convergence** - if you add more data and you don't get different parameters, the model has converged.
- **kernel**: some transformation of your features that improves your classification
- **affine**: indicates that the subspace need not pass through the origin.
- **support vector**: data points that support the maximal margin hyperplane in the sense that if these points were moved slightly then the maximal margin hyper- plane would move as well.
- **marginal likelihood**:
- **marginalized out** = integrated out

Concepts: **likelihood vs posterior**:
likelihood*prior = constant*posterior. $P(Y|X)$ is likelihood, $P(X|Y)$ is posterior.

Normalizing data

Options:

- min/max
- sigmoid
- L_1 norm
- L_2 norm

Note L_1/L_2 normalization versus penalty/regularization!!

Size for modeling $\mathbf{P}(Y = y|X)$

How many parameters are needed to model $P(y|x_1, x_2, ..., x_d)$?
Assume Y is discrete and all the x_i are binary. You have d binary features, so you have 2^d probabilities in order to classify every possible input.

The number of parameters in the PDF of $P(Y = y|X)$ is 2^d (+ 1 for bias sometimes). The size (# of nodes??) of the conditional probability tree grows exponentially. The data is just a table with Y and X columns.

We only need 2^d (or perhaps $2^{(d+1)}$) to get all possibilities specified. If each feature can take m values instead then we have m^d. Not binary any more! The number of parameters can get very big but Naive Bayes can handle it.
What is the order of the size of the parameters you need to do the full conditional probability? For Naive Bayes it is d; **linear**. And the full conditional would be exponential!

Maximum Likelihood Estimation (MLE)

Take log, take derivative, set equal to zero.
Memorize. Likelihood is DATA.
The best mu for a gaussian is the mean. Maximized probability of this data being produced by the distribution. The data is most likely to be generated if the mean is mu. Did same for sigma. We realized best way is to use the variance.
Wikipedia:

To use the method of maximum likelihood, one first specifies the joint density function for all observations. For an independent and identically distributed sample, this joint density function is:

$$f(x_1, x_2, \ldots, x_n \mid \theta) = f(x_1 \mid \theta) \times f(x_2|\theta) \times \cdots \times f(x_n \mid \theta).$$

Now we look at this function from a different perspective by considering the observed values $x1, x2, \ldots, xn$ to be fixed "parameters" of this function, whereas ? will be the function's variable and allowed to vary freely; this function will be called the likelihood:

$$\mathcal{L}(\theta\,;\, x_1, \ldots, x_n) = f(x_1, x_2, \ldots, x_n \mid \theta) = \prod_{i=1}^{n} f(x_i \mid \theta)$$

Note that " ; " denotes a separation between the two input arguments: θ and the observations $x_1, \ldots, x_n$.

In practice it is often more convenient to work with the logarithm of the likelihood function, called the log-likelihood:

$$\ln \mathcal{L}(\theta\,;\, x_1, \ldots, x_n) = \sum_{i=1}^{n} \ln f(x_i \mid \theta)$$

or the average log-likelihood:

$$\hat{\ell} = \frac{1}{n} \ln \mathcal{L}$$

The hat over ℓ indicates that it is akin to some estimator. Indeed, $\hat{\ell}$ estimates the expected log-likelihood of a single observation in the model.

The method of maximum likelihood estimates θ_0 by finding a value of θ that maximizes $\hat{\ell}(\theta; x)$. This method of estimation defines a maximum-likelihood estimator (MLE) of θ_0:

$$\{\hat{\theta}_{\text{mle}}\} \subseteq \{\arg\max_{\theta \in \Theta} \hat{\ell}(\theta\,;\, x_1, \ldots, x_n)\}$$

MLE vs MAP

- both MLE and MAP are point estimates. No estimate of uncertainty.
- MLE fits a probabilistic model $P(x|\theta)$ to data to estimate θ. You chose the parameters θ that maximize $\ln P(X|\theta)$
- MLE is more likely to overfit; MAP regularizes to prevent overfitting.
- MAP doesn't have all the nice asymptotic relationship, but tends to look like MLE asymptotically. As your data goes to infinity, your prior foes to the data.
- unlike MLE, MAP is not invariant under reparameterization. (A disadvantage)
- in MAP, you have to chose a prior. Sometimes not fun to pick.

TA e-mail
In both of these problems, we assume that we have some i.i.d. data $\{x_1, ..., x_M\}$ which was drawn from a distribution $P(x_i|\theta)$, and we want to estimate the parameters θ).
In the MLE scenario, we directly optimize the (log) likelihood of the data $P(X|\theta)$, to get the estimate of θ under which the data has maximum likelihood.
In the MAP scenario, we also have some prior knowledge about what θ should be, e.g. $P(\theta) = \text{Normal}(0, \sigma)$. We can then use Bayes' rule to get $P(\theta|X)$:

$$P(\theta|X) = P(X|\theta)P(\theta)/Z$$

where Z is the normalizing constant
That is, we want our estimate of theta to:
1) Model the data well,
2) Have a high prior probability.
Note that when $P(\theta)$ is constant, i.e. we don't have any prior knowledge, MAP just reduces to MLE.

How Naive Bayes, MLE, and MAP fit together

In order to do Bayesian inference, you put your likelihood into Bayes rule. That has nothing to do with Naive Bayes. If you just maximize the ML equation, that gives you MLE. If you maximize the posterior that you get from Bayes rule then you have MAP. All that is true across analysis.
A full Bayesian analysis for many classification problems is too hard. Too parameter rich, too computationally expensive. You can simplify by making the Naive Bayes assumption. In machine learning world you will usually do this simplification.
Note: Erick doesn't think he will ever use Naive Bayes. He does statistical analysis, not Naive Bayes. Naive Bayes is a much smaller and specialized thing than general Bayesian analysis. That's the statistician perspective.
For ML, it is a nice way of gettitng regularized estimates. You could also use Bayes to relate parameters in a model. That's what Erick does a lot.

Smoothing

Can reduce sensitivity to zero values when multiplying probabilities.

- prior distribution for binaries: Beta distribution.
- Laplace smoothing for multinomial.

  Bayes b/c you aren't going to see a feature vector that matches one in training. We are **not** going to see a feature that is the exact same as a feature in the training set. That's **why** we estimate w/ Bayes.

Generate vs. Discriminative

Discriminative tries to learn $P(y|x)$, whereas a generative model tries to learn $P(x, y) = P(y|x)P(x)$.
Since the generative model learns $P(x)$ as well as $P(y|x)$, it often requires much more data, but can do things like actually generating sampled data from scratch by drawing from $P(x)$ and then from $P(y|x)$ (hence the term 'generative' model). Given that, you could classify the algorithms above as follows:
Generative:

- Naive Bayes (since we learn class distributions $P(y_i)$ and models $P(x_i j|y_i)$, a bit backwards from above, but we're still learning a generative model)

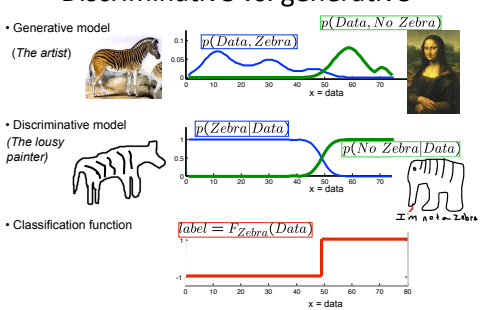
Discriminative:

- Decision Trees
- Perceptron
- SVM

Point estimation and linear regression don't really fit as well into this framework, since we're usually talking about classifiers when making this distinction. For further info you can read the discussion here:
Stats.stackexchange
logistic: discriminative
Naive Bayes: generative

One can only distinct between whether it is something, the other can say how likely.
Two big categories of approaches for ML:
Generative: Those that try to estimate the joint distributions between labels and features/data. Model the joint distributions.
$P(X, Y) = f(X|Y)P(Y) or P(Y|X)f(x)$. "Class conditional densities" are modeled. **If you can create a distribution, you can sample from it.** More powerful if you have enough data to estimate the densities, but worse without enough data. Natural interpretation. Bayes classification is an example. Naive Bayes is one that can produce p(Data,Zebra), except you've made a lot of assumptions. P(Data,Zebra) is a pdf over samples. If you didn't relax all the constraints when going from Bayes to Naive Bayes, you could paint a zebra.
A joint probability model with evidence variable.

Discriminative: No generative model, no Bayes rule, often no probabilities at all!
Those that directly estimate the decision boundary. "discriminative decision boundary". Find $P(Y|X)$
Describing how to generate random instances X conditioned on the target attribute Y.
The discriminative classifier is like a lazy painter.
You can get decision boundary out of a generative model but it is over-kill if you only want to produce a label.
85% of time discriminative outperforms. B/c p(Data,Zebra) is hard to estimate.

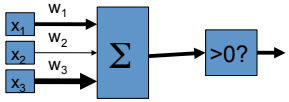


Why does the generative p graph max out at 0.1 and the discriminative at 1? Discriminating against zebra or not zebra. Has to sum to 1. The discriminative doesn't have to sum to 1. Many other things have to sum with it to sum to 1.

TA review:
In discriminative models we just want to discriminate between classes given the data: learning $P(y_i|X_i)$. In generative models we are trying to learn the joint distribution $P(y_i, X_i) = P(y_i)P(X_i|y_i)$, which we can then use to determine $P(y_i|X_i)$ by Bayes rule. Although this requires more data since we're learning $P(X_i|y_i)$, these models can be much more interpretable than discriminative models.

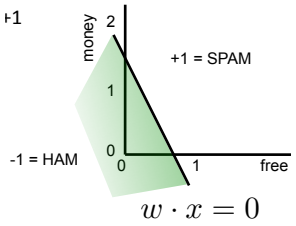
Linear Classifiers

Inputs are feature values.
Each feature has a weight.
Sum is the activation. $\text{activation}_w(x) = \sum_i w_i x_i = w \cdot x$
If the activation is positive, chose output class 1.
If the activation is netative, chose output class 2.



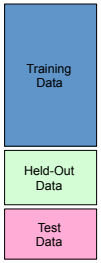
For a binary decision rule:
In the space of feature vectors:

- examples are points
- any weight vector is a hyperplane
- one side corresponds to y = +1
- the other side corresponds to y = -1
- ??? The $w \cdot x = 0$ is the solution to the line.



NB, LR, Perceptron

Three Views of Classification
(more to come later in course!)



- Naïve Bayes:
  - Parameters from data statistics
  - Parameters: probabilistic interpretation
  - Training: one pass through the data
- Logistic Regression:
  - Parameters from gradient ascent
  - Parameters: linear, probabilistic model, and discriminative
  - Training: gradient ascent (usually batch), regularize to stop overfitting
- The perceptron:
  - Parameters from reactions to mistakes
  - Parameters: discriminative interpretation
  - Training: go through the data until held-out accuracy maxes out

Gradient Ascent/Descent vs Coordinate Ascent/Descent

We discussed gradient descent with logistic regression.
We mentioned coordinate descent when getting ready to talk about EM.

Gradient descent

For finding the local minimum.

- Takes steps proportional to the negative of the gradient (or of the approximate gradient) of the function at the current point.
- If instead one takes steps proportional to the positive of the gradient, one approaches a local maximum of that function; the procedure is then known as gradient ascent.

Coordinate descent

- A non-derivative optimization algorithm
- To find a local minimum of a function, one does line search along one coordinate direction at the current point in each iteration. One uses different coordinate directions cyclically throughout the procedure.
- Has problems with non-smooth functions
- Coordinate descent **does** have step size parameter. To prevent over-shooting, you many need to take smaller steps.
- Does converge under two big assumptions:
  (1) fixing one works.
  (2) need loss to get smaller than it was before.
- Won't always converge to global optima.
- For coordinate descent, you have to be extremely careful that each step reduces the loss function. If you can't prove that you should not use coordinate descent.

K-means does this: alternate between holding the assignments and the centers fixed.