CS 229 NOTES: MACHINE LEARNING

ARUN DEBRAY
DECEMBER 2, 2013

CONTENTS

Part 1. Supervised Learning

1. INTRODUCTION: 9/23/13

> *"A few days ago, the enrollment in this class was 650 students. I imagine it's a bit higher now."* [Ed. note: There are 735 people enrolled in this class as of today.[1]]

First, it's worth asking: what *is* machine learning? It grew out of work in AI and attempts to understand data and create self-customizing programs that can't be made by hand. Machine learning algorithms are everywhere, such as web search, photo tagging, and spam detection. Machine learning has become the preferred approach to handling large amounts of data.

[1] For comparison, there are "only" 656 people in CS 106A.

In the past several years, there have been a surprising number of startups that use machine learning to deal with large amounts of data. This class is not about startups, but this is an illustration of its increasing utility. Interestingly, some of these companies are very happy to hear that you get an A in this class!

Prerequisites. There are three kinds of prerequisites:

- It will be necessary to know enough computer science principles to be able to write programs.
- It's going to be necessary to understand basic probability theory, e.g. random variables and such. If this is a problem, there will be a review session.
- Linear algebra on the order of Math 51 or 104.[2] This is things such as the idea of vectors and matrices, matrix inverses, etc. Knowing things like eigenvalues and eigenvectors is great, but not strictly necessary.

In addition to this, it will be necessary to use Matlab (or its open source alternative, Octave) for some of the programming assignments. It is one of the best languages for prototyping machine learning algorithms, though larger-scale algorithms tend to be reimplemented in C++ or Java. SciPy and NumPy are also useful prototyping choices. There are other reasons Matlab is useful, too. However, some of the Friday discussion sections will teach Matlab for those who do not know it.

Project. This class will have a significant final project at the end. Most people approach this by using machine learning algorithms to implement something interesting. When choosing your project, take a look at the past projects posted on the course website.

There are lots of places to get project ideas. In addition to looking at past projects, there are research groups across campus have project ideas that they would be happy to see you work on. A list of this will be posted on the course website at some point.

Cool Stuff. As is traditional, we will begin with a definition.

Definition (Arthur Samuel, 1959)**.** Machine learning is the field of study that gives computers the ability to learn without being explicitly programmed.

Samuel wrote a checkers-playing program that played thousands of games against itself, and eventually managed to consistently beat him. This was a milestone in the history of artificial intelligence, one of the first (if not the first) of the programs to exceed the ability of its creator.

There's a more modern definition that is more commonly cited today:

Definition (Tom Mitchell, 1998)**.** A computer program is said to *learn* from an experience E with respect to some task T and some performance measure P if its performance on T, measured with respect to P, improves after E.

Example 1.1 (Supervised Learning and Regression)**.** Consider a graph that plots the area of a house against its price. Then, the goal is to understand how much to sell a house for given its size. This is an example of supervised learning; for example, the algorithm might fit a line or a quadratic curve to the data. Specifically, this is an example of a regression problem, because the price is a continuous variable (or at least close enough).

Example 1.2 (Classification)**.** If the dependent variable is discrete, then one has a classification problem. For example, one may have a set of images of a tumor and want to predict whether it is malignant or benign (e.g. given some feature of it, such as its size). One may also estimate probabilities rather than just picking one output.

Though both examples above make predictions based on a single feature, other problems might need to use multiple features (e.g. the tumor size and the age of the patient for Example 1.2). Then, classifying these into some number of features can be viewed as drawing a line or hyperplane between two sets of points. In general, knowing which sets of features to use is one of the most difficult questions in machine learning, and can lead to much more or less effective results, and there is even an effective algorithm with an *infinite* number of features. This leads to questions in infinite-dimensional space, and boils down to a powerful algorithm called a support vector machine (SVM). This leads to questions about how one can hold an infinite list in a finite computer's memory, but there is a nice way to handle this — stay tuned.

Supervised learning has nice applications to autonomous driving. Here,[3] the human driver demonstrates correct driving behavior in various conditions, and the car picks up on this to determine what to do in various situations. There's a lot of trickery in getting the driver's motions and the "circumstances" into plain, precise data. Also, if the experiment is successful enough, the car learns the particular habits of its driver… Impressively, the algorithms used for that car will be superseded by things taught in this class in week four.

This class will also spend some time on the theory of learning.[4] As always, learning the theoretical tools will help one create better applications, such as by knowing which of many approaches is the most likely to be successful, causing one to use

[2] Dr. Ng said Math 103, but I don't believe that course exists.

[3] A video was shown from 20 years ago that demonstrated this. Scarily enough, it contained the sentence, "At first, the car steers randomly."

[4] The professor has a mathematician friend who claims that he does math to understand the truth and beauty of the universe, but then claimed that in this class, "we're not going to see any truth or beauty."

time more efficiently. Stanford is one of a very few places where the theory is so stressed, and this might have something to do with the competence of its CS 229 alums.

Another major topic is unsupervised learning, where one takes a set of data and asks what sort interesting things one can see in it. For example, in a set of data, one might use a clustering algorithm to search for clusters in the data, or sets of points that are closer to each other than to other points. This is distinct from supervised learning because no correct examples are given.

Example 1.3 (Cocktail Party Problem)**.** Imagine a party where there are two people speaking and two microphones. Each microphone captures a different combination of the speakers' voices. Then, the goal is to reconstruct the original sound file.

The name of the problem comes from the idea that it's hard to hear people at a crowded, lively cocktail party. It has applications to separating voices out of background noise, music, etc.

This algorithm can be implemented in one line, but that line was the result of years of research!

Unsupervised learning has the general goal to obtain structure out of data.

Finally, reinforcement learning is used to do things like fly helicopters and teach robots. We will learn about that too.

2. Linear Regression, Gradient Ascent, and the Normal Equation: 9/25/13

> *"One lesson you will learn over and over this quarter is that machine learning people just aren't good at naming stuff."*

The more basic algorithms presented today will provide a solid foundation for more powerful algorithms presented later in the course. Additionally, a lot of the math behind the algorithmic analysis or the algorithms themselves will be deferred to section or to the lecture notes.

Regresssion, in which one predicts a continuous-valued output, is one of the two major types of supervised learning problems (the other is classification, for discrete outputs).

As a running example for today, consider a set of data comparing house sizes with their listed prices. The goal is to predict the price of a house as a function of its size.

The input data is called the training set (the data we already know), and the learning algorithm takes this and produces a function h called the hypothesis,[5] which takes a house size and outputs a predicted (estimator) price for it. In other words, $h : X \to Y$.

A standardized notation is very useful for making things easier to understand. Thus, in these kinds of supervised learning problems:

- m denotes the number of training examples (number of data points),
- x denotes an input variable (sometimes also called a feature),
- y denotes an output feature,
- (x, y) is a training example, and
- $(x^{(i)}, y^{(i)})$ is the i^{th} training example.

Since this is a CS class, there will be ambiguity as to whether $=$ means assignment or assertion of equality. Thus, $\coloneqq$ will be used to denote assignment, and $=$ as an assertion.

The first question for developing an algorithm is, *how does one represent h*? In this lecture, h will be represented as a linear function[6] $h(x) = \theta_0 + \theta_1(x)$. If there were multiple features x_1 and x_2 (e.g. the size and the number of bedrooms), this function would take the form $h(x_1, x_2) = \theta_0 + \theta_1 x_1 + \theta_2 x_2$. Here, the θ_i are called the parameters of h.

The last bits of notation (for now) will be the convention that $x_0 = 1$; this makes it possible to take a dataset with n features and define

$$h(x) = \sum_{j=0}^{n} \theta_j x_j = \theta^{\mathrm{T}} \mathbf{x},$$

where

$$\theta = \begin{bmatrix} \theta_1 \\ \vdots \\ \theta_n \end{bmatrix} \quad \text{and} \quad \mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}.$$

Sometimes, to emphasize the role of θ, one writes $h_\theta(\mathbf{x})$ for the hypothesis.

[5] This terminology is basically due to tradition, even though it might be more accurate to call it a prediction function. There doesn't appear to be a deep connection between h and the hypotheses of experimental science.

[6] Those of you intimately familiar with the terminology will know this is an affine function, but the distinction is not essential here.

Since one wants $h_\theta(\mathbf{x})$ to be close to y, one might try to minimize the distance (squared error) between the two, so the goal is to minimize

$$J(\theta) = \frac{1}{2}\sum_{i=1}^{n} h_\theta\left(x^{(i)} - y^{(i)}\right)^2.$$

Here, the $1/2$ is there for convenience (since it's minimized anyways, this will make life easier). Squared-error is particularly helpful because it implies that the errors tend to be Gaussian, but other models exist.

Gradient Descent. This method for learning a linear function starts with some θ, such as $\theta = \mathbf{0}$, and then updates θ to minimize $J(\theta)$. Specifically, the goal is to update θ by a small amount in the direction of the steepest descent of $J(\theta)$ at θ.

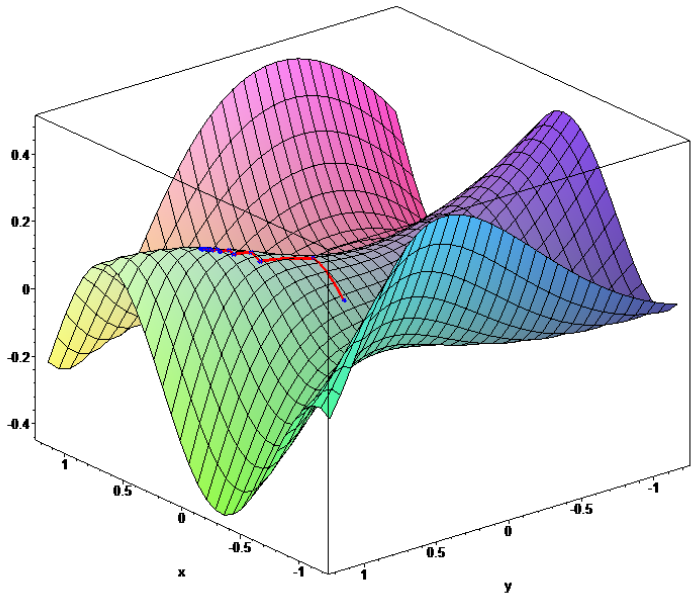


FIGURE 1. An illustration of gradient descent, from Wikipedia. "A small step for man, a giant leap for machine learning."

In each step, θ is updated as follows:

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta). \tag{1}$$

Here, α is a constant called the learning rate. Larger values of α correspond to larger step sizes overall, though as the minimum is approached the step sizes lessen.

As an example of why one would want to compute the partial derivative, imagine $m = 1$. Then,

$$\begin{aligned}
J(\theta) &= \frac{1}{2}(h_\theta(x) - y)^2 \\
\implies \frac{\partial}{\partial \theta_j} J(\theta) &= \frac{\partial}{\partial \theta_j}\left(\frac{1}{2}(h(x) - y)^2\right) \\
&= 2 \cdot \frac{1}{2} \cdot (h_\theta(x) - y)\frac{\partial}{\partial \theta_j}(h_\theta(x) - y) \\
&= (h_\theta(x) - y)\frac{\partial}{\partial \theta_j}(\theta_0 x_0 + \theta_1 x_1 + y) \\
&= (h_\theta(x) - y)x_j.
\end{aligned}$$

Now, (1) is iterated until the algorithm converges to within some desired tolerance. By the above, this can be simplified slightly to

$$\theta_j := \theta_j - \alpha \sum_{i=1}^{n} (h_\theta\left(x^{(i)} - y^{(i)}\right) x_j^{(i)}.$$

In gradient descent, it is important to remember the difference between a local and a global minimum; otherwise, the function could just get caught in a local minimum. However, if $J(\theta)$ is convex, then there is a single local minimum and it is also a global minimum, so gradient descent necessarily converges to the optimum.

In some sense, the function is minimized by finding a θ such that $\nabla J(\theta) = 0$; that is exactly what gradient descent finds, and other methods do the same thing in a different way.

One disadvantage of gradient descent is that each step requires summing over all m examples, which is extremely bad when m is large (what if you're working with $m = 3 \cdot 10^8$, as in the United States Census data?). This algorithm is specifically called *batch gradient descent*, which means that the entire input is looked at in each step. A different algorithm called *stochastic gradient descent* deals with this problem better and faster; but it will be covered in more depth in the lecture notes.

For some classes of functions, one can analytically solve for the minimum. This tends to involve lots of ugly calculations, but a simplified notation of calculus of matrices makes all of these derivations much less complicated. These will be covered in better depth in the lecture notes and in section.

Definition. The derivative of J with respect to θ is the vector

$$\nabla_\theta(J) = \begin{bmatrix} \frac{\partial J}{\partial \theta_0} \\ \vdots \\ \frac{\partial J}{\partial \theta_n} \end{bmatrix} \in \mathbb{R}^{n+1},$$

since $\theta \in \mathbb{R}^{n+1}$ as well.

Now, it is possible to simplify gradient ascent into

$$\theta \coloneqq \theta - \alpha \nabla_\theta J.$$

The n lines of equations are efficiently condensed into one.

More generally, suppose $A \in \mathbb{R}^{m \times n}$ is a matrix and $f : \mathbb{R}^{m \times n} \to \mathbb{R}$, such as $A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$ and $f(A) = A_{11} + A_{12}^2$. Then,

$$\nabla_A f(A) = \begin{bmatrix} \frac{\partial f}{\partial A_{11}} & \cdots & \frac{\partial f}{\partial A_{1n}} \\ \vdots & \ddots & \vdots \\ \frac{\partial f}{\partial A_{m1}} & \cdots & \frac{\partial f}{\partial A_{mn}} \end{bmatrix} \in \mathbb{R}^{m \times n}.$$

This is a compact way of collecting all of the component partial derivatives. The derivative matrix always has the same dimension as the original one.

With f as above, $\nabla_A f(A) = \begin{bmatrix} 1 & 2A_{12} \\ 0 & 0 \end{bmatrix}$.

Now, some properties of matrices will be presented without proof. You can prove them on your own if you like.

Definition. If A is a square $n \times n$ matrix, the trace is the sum of its diagonal entries: $\operatorname{tr} A = \sum_{i=1}^n A_{ii}$.

Though this is a function, it can use this operator-like notation. Also, note the special case $a \in \mathbb{R}$, for which $\operatorname{tr} a = a$. We have the following facts:

- If A and B are square matrices of the same dimension, then $\operatorname{tr} AB = \operatorname{tr} BA$.
- The cyclic permutation of trace: $\operatorname{tr} ABC = \operatorname{tr} CAB = \operatorname{tr} BCA$.
- $\nabla_A \operatorname{tr} AB = \nabla_A \operatorname{tr} BA = B^{\mathrm{T}}$. This should be thought of as a function in A, where B is a provided constant: $f(A) = \operatorname{tr} AB$.
- $\nabla_A \operatorname{tr} AA^{\mathrm{T}}C = CA + C^{\mathrm{T}}A$.

Definition. The design matrix of some training data is made by stacking all of the training examples as rows:

$$X = \begin{bmatrix} \mathbf{x}^{(1)} \\ \vdots \\ \mathbf{x}^{(n)} \end{bmatrix} \in \mathbb{R}^{n \times n}.$$

Thus,

$$X\theta = \begin{bmatrix} \mathbf{x}^{(1)\mathrm{T}} \cdot \theta \\ \vdots \\ \mathbf{x}^{(n)\mathrm{T}} \cdot \theta \end{bmatrix} = \begin{bmatrix} h(\mathbf{x}^{(1)}) \\ \vdots \\ h(\mathbf{x}^{(n)}) \end{bmatrix},$$

so it conveniently represents the predictions. Also, let

$$\mathbf{y} = \begin{bmatrix} y^{(1)} \\ \vdots \\ y^{(n)} \end{bmatrix}.$$

Then, one obtains the normal equation after a short derivation:

$$J(\theta) = \frac{1}{2}(X\theta - \mathbf{y})^{\mathrm{T}}(X\theta - \mathbf{y}).$$

Here, $(X\theta - \mathbf{y})$ is the error, so you can see how it is squared.

Now, one can take the derivative, though there's a lot skipped in the derivation:

$$\begin{aligned} \nabla_\theta \frac{1}{2}(X\theta - \mathbf{y})^{\mathrm{T}}(X\theta - \mathbf{y}) &= \frac{1}{2}\nabla_\theta(\theta^{\mathrm{T}}X^{\mathrm{T}} - \mathbf{y}^{\mathrm{T}})(X\theta - \mathbf{y}) \\ &= X^{\mathrm{T}}X\theta - X^{\mathrm{T}}\mathbf{y}. \end{aligned}$$

Setting the last equation to zero gives the normal equation:

$$X^{\mathrm{T}}X\theta = X^{\mathrm{T}}\mathbf{y}.$$

This is useful to solve in one step in certain situations: $\theta = (X^{\mathrm{T}}X)^{-1}X^{\mathrm{T}}\mathbf{y}$.

3. Linear Algebra and Matrix Calculus Problem Session: 9/27/13

This section was to review the basics of linear algebra and matrix calculus, and was taught by three of the TAs.

To start, it will be helpful to define the lecture's notation. Column vectors look like this:

$$\mathbf{x} = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} \in \mathbb{R}^n.$$

Throughout this lecture, $\mathbf{x}$ will be a vector in $\mathbb{R}^n$, and $A \in \mathbb{R}^{m\times n}$ is a matrix.

On $\mathbb{R}^n$ we have some norms:

- The 2-norm is
$$||\mathbf{x}||_2 = \sqrt{\sum_{i=1}^{m} x_i^2}.$$
- The 1-norm is $||\mathbf{x}||_1 = \sum_{i=1}^{m} |x_i|$.
- The infinity norm is $||\mathbf{x}||_\infty = \max_i\{x_i\}$.

An $m \times n$ matrix can be seen as n column vectors $A = (\mathbf{c}_1, \ldots, \mathbf{c}_n)$ or m row vectors $A = \begin{pmatrix} \mathbf{r}_1^{\mathrm{T}} \\ \vdots \\ \mathbf{r}_m^{\mathrm{T}} \end{pmatrix}$.

Two vectors $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ are multiplied with the inner product:

$$\mathbf{x}\cdot\mathbf{y} = \mathbf{x}^{\mathrm{T}}\mathbf{y} = \mathbf{y}^{\mathrm{T}}\mathbf{x} = \sum_{i=1}^{n} x_i y_i.$$

Then, $||x||_2^2 = \mathbf{x}^{\mathrm{T}}\mathbf{x}$.

Since there are several notations for a matrix, the matrix-vector product has several different meanings. Using the row vector notation, for example, $A\mathbf{x} = (\mathbf{r}_1 \ldots \mathbf{x}, \ldots, \mathbf{r}_m)^{\mathrm{T}}$ (a column vector).

Additionally, one has the matrix-matrix product, which is defined when $A \in \mathbb{R}^{m\times n}$ and $B \in \mathbb{R}^{n\times p}$; then, their product is in $\mathbb{R}^{m\times p}$. If $C = AB$, then $c_{ij} = \sum_{k=1}^{n} a_{ik}b_{kj}$.

Here are some properties of matrix multiplication:

- The order of multiplication is not important (associativity): $(AB)C = A(BC)$.
- Distributivity: $A(B+C) = AB + AC$.
- In general, the matrix product is *not* commutative: $AB \neq BA$. Do not make this mistake.

These properties are relatively easy to prove using the formula given above.

Now, a few operators on matrices: the transpose operator just flips matrices. $(A^{\mathrm{T}})_{ij} = A_{ji}$. This has a few nice properties:

- $(A+B)^{\mathrm{T}} = A^{\mathrm{T}} + B^{\mathrm{T}}$.

- $(AB)^{\mathrm{T}} = B^{\mathrm{T}}A^{\mathrm{T}}$.
- $(A^{\mathrm{T}})^{\mathrm{T}} = A$.

Again, these are things you can prove yourself if you want.

Definition. A matrix is symmetric if $A^{\mathrm{T}} = A$. A matrix for which $A^{\mathrm{T}} = -A$ is called anti-symmetric.

Symmetric matrices have interesting properties, and thus come up in the material more often.

The inverse of a matrix doesn't always exist; if it does, then it is the matrix A^{-1} of A such that $A^{-1}A = AA^{-1} = I$ (the identity matrix, with 1 along the diagonal and 0 elsewhere). If such an inverse exists, A is called nonsingular. We have some more properties:

- $(A^{-1})^{-1} = A$.
- $(A^{\mathrm{T}})^{-1} = (A^{-1})^{\mathrm{T}}$. Sometimes, this is denoted $A^{-\mathrm{T}}$.
- $(AB)^{-1} = B^{-1}A^{-1}$.

These properties hold when everything is well-behaved and invertible, so don't forget that and misapply them.

We saw the trace $\mathrm{tr}(A)$ in class, which is linear: $\mathrm{tr}(A+B) = \mathrm{tr}(A) + \mathrm{tr}(B)$. $\mathrm{tr}(A^{\mathrm{T}}) = \mathrm{tr}(A)$, additionally, and $\mathrm{tr}(AB) = \mathrm{tr}(BA)$. This last property is particularly useful for computations, and can be generalized to larger products: $\mathrm{tr}(ABC) = \mathrm{tr}(CAB) = \mathrm{tr}(BCA)$ and so on. However, this is not true for random orders: they must be cyclical.

Definition. The Frobenius norm of a matrix A is

$$||A||_{\mathrm{F}} = \sqrt{\mathrm{tr}(AA^{\mathrm{T}})} = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n} A_{ij}^2}.$$

Proving the last equality above would be a good way to get better at using these definitions.

Another useful function is the determinant. It will be briefly discussed here, and a better explanation with a more geometric angle[7] is presented. The formal definition is

$$\det A = |A| = \sum_{i=1}^{m} (-1)^{i+j} a_{ij} \left| A_{/i/j} \right|,$$

where $A_{/i/j}$ (the cofactor matrix) is the matrix obtained by removing the i^{th} row and the j^{th} column. Also, note that j doesn't appear in the definition; its choice happens to be arbitrary.

Here are some properties of the determinant:

- $\det(AB) = \det(A)\det(B)$.
- $\det(A) = 0$ iff A is singular (i.e. noninvertible).
- $\det(A^{\mathrm{T}}) = \det(A)$.
- If A is nonsingular, $\det(A^{-1}) = 1/\det(A)$.

Definition. A family $\mathbf{u}_1, \ldots, \mathbf{u}_n$ of vectors is linearly independent if for every family $(\alpha_1, \ldots, \alpha_n) \in \mathbb{R}^n$, $\alpha_1\mathbf{u}_1 + \cdots + \alpha_n\mathbf{u}_n = 0$ implies that $\alpha_1 = 0, \ldots, \alpha_n = 0$.

The notion of linear independence can be used to define the rank of a matrix; see the lecture notes for this concept.

Definition. The range of a matrix A, denoted $\mathrm{R}(A)$, is the set $\mathrm{R}(A) = \{A\mathbf{x} \mid \mathbf{x} \in \mathbb{R}^m\}$. The null space of A, denoted $N(A)$, is the set $N(A) = \{\mathbf{x} \in \mathbb{R}^m \mid A\mathbf{x} = 0\}$.

Notice that $\mathrm{R}(A) \subset \mathbb{R}^n$, but $N(A) \subset \mathbb{R}^m$, which can be remembered by tracking which corresponds to the input space and which corresponds to the output space.

The null space is sometimes also denoted the kernel, but not in this class.

Definition. $(\lambda, \mathbf{x})$ is an eigenvalue-eigenvector of A if $\mathbf{x} \neq 0$ and $A\mathbf{x} = \lambda\mathbf{x}$.

For small matrices, one finds the eigenvalues by computing the roots to $\det(\lambda I - A) = 0$. This polynomial (since it ends up being a polynomial) is called the characteristic polynomial. Eigenvectors and eigenvalues play a role in a lot of fields, but will be the most useful in this class in principal component analysis.

Definition. A matrix $U \in \mathbb{R}^{m \times m}$ is orthogonal if $U^{\mathrm{T}}U = UU^{\mathrm{T}} = I$.

In other words, $U^{-1} = U^{\mathrm{T}}$.

Claim. If U is an orthogonal matrix, then $||U\mathbf{x}||_2 = ||\mathbf{x}||_2$ for any $\mathbf{x} \in \mathbb{R}^m$.

[7] No pun intended.

Proof. $||U\mathbf{x}||_2^2 = (U\mathbf{x})^{\mathrm{T}}U\mathbf{x} = \mathbf{x}^{\mathrm{T}}U^{\mathrm{T}}U\mathbf{x} = \mathbf{x}^{\mathrm{T}}\mathbf{x} = ||\mathbf{x}||_2$. ⊠

This result is very useful in the matrix calculus, so don't forget it!

Claim. Let A be a symmetric[8] $n \times n$ mstrix and $\lambda_1, \ldots, \lambda_n$ be its eigenvalues. Then, $\mathrm{tr}(A) = \sum_{i=1}^n \lambda_i$.

Proof. One useful property is that every symmetric matrix A is of the form $A = U\Lambda U^{\mathrm{T}}$, where U is orthogonal and Λ is diagonal. Because of this equality, $\mathrm{tr}(A) = \mathrm{tr}(U\Lambda U^{\mathrm{T}}) = \mathrm{tr}(\Lambda U^{\mathrm{T}}U) = \mathrm{tr}(\Lambda) = \sum_{i=1}^n \lambda_i$. ⊠

The key ingredient in the proof, other than the decomposition, is that $\mathrm{tr}(AB) = \mathrm{tr}(BA)$. Similarly to the above proof, it is possible to show that $\det(A) = \prod_{i=1}^n \lambda_i$.

If $A \in \mathbb{R}^{m\times m}$ is a symmetric matrix, then the quadratic form associated with A is the function $\mathbf{x}^{\mathrm{T}}A\mathbf{x} \in \mathbb{R}$.[9]

Definition.

- A matrix A is positive definite if for all $\mathbf{x} \in \mathbb{R}^n \setminus 0$, $\mathbf{x}^{\mathrm{T}}A\mathbf{x} > 0$.
- A matrix A is semi-positive definite if for all $\mathbf{x} \in \mathbb{R}^n$, $\mathbf{x}^{\mathrm{T}}A\mathbf{x} \geq 0$.
- Negative definiteness and semi-negative definiteness are defined similarly.

An important fact is that all of the eigenvalues of a positive definite matrix are positive, and all those of a semi-positive definite matrix are nonnegative. These are also good, not too difficult things to prove. Similar results hold for negative (semi)-definite matrices.

Matrix Calculus. Suppose $f : \mathbb{R}^n \to \mathbb{R}$ (i.e. it is a vector-valued function), such as $f(\mathbf{a}) = \mathbf{b}^{\mathrm{T}}\mathbf{a}$ for some $\mathbf{b} \in \mathbb{R}^n$. Then, the gradient of f is defined to be $\nabla_{\mathbf{v}} f(\mathbf{x})_i = \frac{\partial f}{\partial x_i}$. With f as in the example, $\frac{\partial f}{\partial x_k} = b_k$, so $\nabla_{\mathbf{v}} f(\mathbf{x}) = \mathbf{b}$ for all $\mathbf{v} \in \mathbb{R}^n$.

Now, if f is a function of a matrix, as in $f(A) = \mathbf{x}^{\mathrm{T}}A^{\mathrm{T}}A\mathbf{x}$ for some $\mathbf{x}$, then $(\nabla_A f)_{pq} = \frac{\partial f}{\partial A_{pq}}$. Ask yourself: how is this function defined on A_{pq}? Then,

$$\mathbf{x}^{\mathrm{T}}A^{\mathrm{T}}A\mathbf{x} = \sum_{i,j=1}^n x_i A_{ij}^2 x_j.$$

if $M = A^{\mathrm{T}}A$, then this simplifies a bit: $\mathbf{x}^{\mathrm{T}}M\mathbf{x} = \sum x_i M_{ij} x_j$. Thus, $\nabla_M f_k(\mathbf{x})$ is nonzero when $i = k$ or $j = k$; in the first case, this gives $\sum_{j=1}^m M_{kj}\mathbf{x}_j$, and when $k = j$, one adds $\sum_{i=1}^n AM_{ik}x_i$, so the result is

$$\sum_{j=1}^m M_{kj}\mathbf{x}_j + \sum_{i=1}^n AM_{ik}x_i.$$

Since M is symmetric, this is the k^{th} component of $2M\mathbf{x}$, so $\nabla_{\mathbf{x}} f(\mathbf{x}) = 2M\mathbf{x}$.

For another example, if $f(\mathbf{x}) = ||A\mathbf{x} - \mathbf{b}||_2^2$, then

$$\begin{aligned} f(\mathbf{x}) &= (A\mathbf{x} - \mathbf{b})^{\mathrm{T}}(A\mathbf{x} - \mathbf{b}) \\ &= \mathbf{x}^{\mathrm{T}}A^{\mathrm{T}}A\mathbf{x} - \mathbf{b}^{\mathrm{T}}A\mathbf{x} - \mathbf{x}^{\mathrm{T}}A^{\mathrm{T}}\mathbf{b} + \mathbf{b}^{\mathrm{T}}\mathbf{b} \\ &= \mathbf{x}^{\mathrm{T}}A^{\mathrm{T}}A\mathbf{x} - 2\mathbf{b}^{\mathrm{T}}A\mathbf{x} + \mathbf{b}^{\mathrm{T}}\mathbf{b}. \end{aligned}$$

Since the gradient is linear, this simplifies to

$$\nabla_{\mathbf{x}} f(\mathbf{x}) = 2A^{\mathrm{T}}A\mathbf{x} - 2A^{\mathrm{T}}\mathbf{b},$$

since $\mathbf{b}^{\mathrm{T}}A\mathbf{x} = (A^{\mathrm{T}}\mathbf{b})^{\mathrm{T}}$, and the gradient of the inner product is already known.

Since we wish to find the global minimum, where this gradient is nonzero, then $A^{\mathrm{T}}A\mathbf{x} = A^{\mathrm{T}}\mathbf{b}$ at the gradient, or $\mathbf{x} = (A^{\mathrm{T}}A)^{-1}A^{\mathrm{T}}\mathbf{b}$.

4. Locally Weighted Regression, MLE, and Logistic Regression: 9/30/13

"I'm still not entirely sure why I showed you that [video]."

Recall the definitions provided last lecture: the function $h_\theta(\mathbf{x}) = \sum \theta_j x_j = \theta^{\mathrm{T}}\mathbf{x}$, the function $J(\theta) = (1/2)\sum h_\theta(x^{(i)} - y^{(i)})$ is the sum of squares (to be minimized), and a direct solution can be given by $\theta = (X^{\mathrm{T}}X)^{-1}X^{\mathrm{T}}\mathbf{y}$.

[8] This condition happens to be completely unnecessary for the result; it is true for any square matrix A.

[9] Consult the lecture notes to see why symmetry is necessary.

Locally Weighted Regression. Often, one has a bunch of data that doesn't really work with a linear regression. There are lots of other options for feature selection, such as $\theta_0 + \theta_1 x + \theta_2 x^2$, or maybe $\theta_0 + \theta_1 x + \theta_2 \sqrt{x}$, but neither of these might be what one looks for. Sometimes, the data fits a very clear curve but there's no obvious set of features to achieve that curve.

In machine learning, there are two kinds of algorithms:

- In a *parametric learning algorithm*, there is a fixed number of parameters θ_i, and it is OK to disregard some data to make predictions with the θ_i.
- In a *non-parametric learning algorithm*, one keeps the entire data set; the amount of data one needs to make the predictions grows linearly with m (the number of features).

Example 4.1. This kind of confusing at first glance, so consider the example of house price prediction from before. A linear regression takes some data and fits a straight line to it.

Locally weighted regression fits a line within a neighborhood of each data point. Additionally, the whole dataset isn't analyzed at once; the next time, someone may query a different point, and the regression is then calculated using that point.

The goal of locally weighted regression is to, given some x, fit θ to minimize

$$\sum_{i=1}^{m} w^{(i)} \left(y^{(i)} - \theta^{\mathrm{T}} x^{(i)}\right)^2, \quad \text{where} \quad w^{(i)} = e^{-((x^{(i)}-x)^2)/2}.^{10}$$

If $|x^{(i)} - x|$ is small, then $w^{(i)} \approx e^0 = 1$, but if $|x^{(i)} - x|$ is large, then $w^{(i)}$ involves taking the exponent of a large negative number, and is close to zero.

Sometimes, $w^{(i)}$ is calculated as $w^{(i)} = \exp(-(x^{(i)} - x)^2/2\tau^2)$. Here, τ is called the bandwidth; larger values of τ correspond to larger neighborhoods under consideration. Like most things here, this satisfies a Goldilocks property: it is best when not too small, not too large, but just right.

The Probabilistic Interpretation of Linear Regression. In short, why are we minimizing the sum-of-squares? Here are some assumptions that we think are true about the world:

- The price of a house (or whatever the model is) is linear plus some error: $y^{(i)} = \theta^{\mathrm{T}} x^{(i)} + \varepsilon^{(i)}$, where the error is due to unlabeled effects (features not tracked) and some amount of random noise.
- The errors are independently and identically distributed normal random variables: $\varepsilon^{(i)} \sim \mathsf{N}(0, \sigma^2)$, so that

$$P(\varepsilon^{(i)}) = \frac{\exp(-\varepsilon^{(i)2}/2\sigma^2)}{\sigma\sqrt{2\pi}},$$

or

$$P(y^{(i)} \mid x^{(i)}; \theta) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(y^{(i)} - \theta^{\mathrm{T}} x^{(i)})^2}{2\sigma^2}\right),$$

so that $(y^{(i)} \mid x^{(i)}; \theta) \sim \mathsf{N}(\theta^{\mathrm{T}}\mathbf{x}, \sigma^2)$. The semicolon in $P(y^{(i)} \mid x^{(i)}; \theta)$ should be read as "parameterized by." In this frequentist world, θ isn't a random variable, but rather a true, but unknown value.

The likelihood function measures the probability that some parameters θ are correct given the data. Notice that this is different from the probability of the data given the parameters; a probability is calculated in both cases, but whether the parameters are data or predicted values depends differs.

The function itself is given by

$$\begin{aligned} L(\theta) &= P(\mathbf{y} \mid X; \theta) \\ &= \prod_{i=1}^{m} P(y^{(i)} \mid \mathbf{x}^{(i)}; \theta) \\ &= \prod_{i=1}^{n} \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(y^{(i)} - \theta^{\mathrm{T}}\mathbf{x}^{(i)})^2}{2\sigma^2}\right). \end{aligned}$$

The goal is to choose the value of θ that maximizes $L(\theta)$, called the maximum likelihood estimation.

[10] Though $w^{(i)}$ looks like a Gaussian model, this is a coincidence, and the two are unrelated.

To make calculations more tractable, one takes the logarithm, producing the creatively named log-likelihood:

$$\begin{aligned}\ell(\theta) = \log L(\theta) &= \log \prod_{i=1}^{n} \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(y^{(i)} - \theta^{\mathrm{T}}\mathbf{x}^{(i)})^2}{2\sigma^2}\right) \\ &= \sum_{i=1}^{n} \left(\log\left(\frac{1}{\sigma\sqrt{2\pi}}\right) + \log\exp\left(-\frac{(y^{(i)} - \theta^{\mathrm{T}}\mathbf{x}^{(i)})^2}{2\sigma^2}\right)\right) \\ &= m\log\frac{1}{\sigma\sqrt{2\pi}} + \sum_{i=1}^{m} -\frac{(y^{(i)} - \theta^{\mathrm{T}}\mathbf{x}^{(i)})^2}{2\sigma^2},\end{aligned}$$

and since the first term is invariant, then to maximize $\ell(\theta)$ and therefore $L(\theta)$, one wishes to minimize

$$J(\theta) = \frac{1}{2}\sum_{i=1}^{m} \frac{(y^{(i)} - \theta^{\mathrm{T}}\mathbf{x}^{(i)})^2}{2\sigma^2}.$$

Notice that the final answer doesn't depend on σ^2. This will be significant when this class turns to generalized linear models.

Logistic Regression. Now we turn to a classification problem, where $y^{(i)} \in \{0, 1\}$. Here, a linear regression isn't the smartest, though it does work sometimes. One can add a training example with $y^{(m+1)} = 1$ to the far right, which makes the data fit very differently. This is a problem, because it shouldn't make any difference to the prediction…

This is why it is bad to use linear regression on classification problems; sometimes it works, but often it doesn't, and in any case predicting values outside of $[0, 1]$ is a problem. Thus, an algorithm called logistic regression is used to address both of these problems.[11] Here, $h_\theta(\mathbf{x}) \in [0, 1]$ is given by

$$h_\theta(\mathbf{x}) = \frac{1}{1 + e^{-\theta^{\mathrm{T}}\mathbf{x}}} = g(\theta^{\mathrm{T}}\mathbf{x}).$$

Here, $g(z) = 1/(1 + e^{-z})$ is called the sigmoid or logistic function, with graph given in Figure 2. This is convenient because

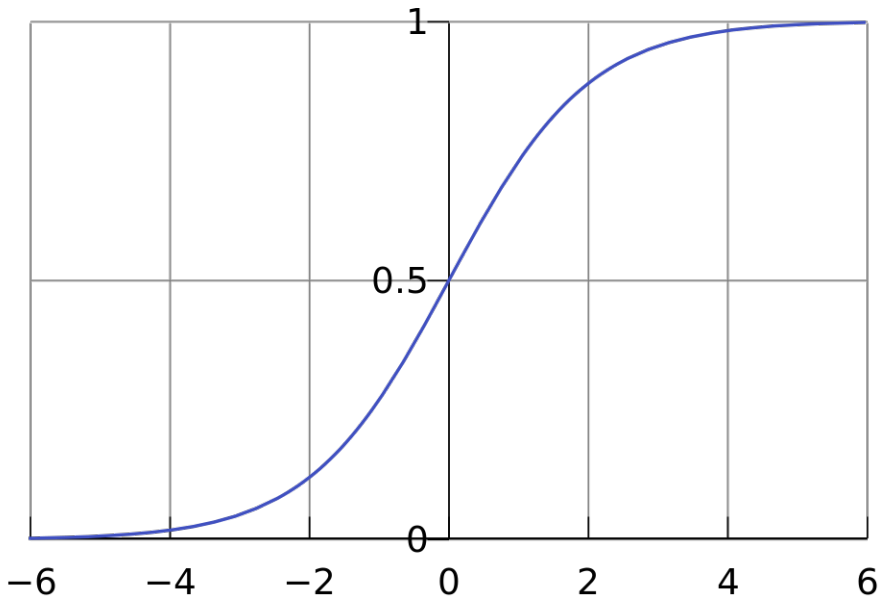


FIGURE 2. The sigmoid function $g(z)$. Source: Wikipedia.

statistical outliers that messed up the linear model are okay for the logistic one, and all of the predicted values lie in $[0, 1]$. The function smoothly goes between 0 and 1 along the real line, but it's particularly useful to make the algorithm convex (which implies a unique global solution), and because it ends up being a special case of a generalized linear model. This makes it possible for more powerful, general algorithms to work on logistic regression.

[11] Logistic regression is a classification algorithm, not a regression algorithm. Take care to not confuse this.

The probabilistic interpretation of this model is that $P(y = 1 \mid \mathbf{x}; \theta) = h_\theta(\mathbf{x})$, and similarly, $P(y = 0 \mid \mathbf{x}; \theta) = 1 - h_\theta(\mathbf{x})$. In other words, since $y \in \{0, 1\}$, then

$$P(y \mid \mathbf{x}; \theta) = h_\theta(\mathbf{x})^y (1 - h_\theta(\mathbf{x}))^{1-y}.$$

This can be checked by filling in the two possibilities for y.

Now, let's apply a maximum likelihood estimate:

$$\begin{aligned} L(\theta) &= \prod_{i=1}^{m} P(y^{(i)} \mid \mathbf{x}^{(i)}; \theta) \\ &= \prod_{i=1}^{m} h(\mathbf{x}^{(i)})^{y^{(i)}} (1 - h(\mathbf{x}^{(i)}))^{1-y^{(i)}}. \end{aligned}$$

It is again possible to calculate the log-likelihood $\ell(\theta) = \log L(\theta)$ as before. This allows one to use gradient ascent to maximize the log-likelihood, by updating the parameters by

$$\theta_j \coloneqq \theta_j + \alpha \frac{\partial}{\partial \theta_j} \ell(\theta),$$

where we are trying to maximize $\ell(\theta)$ rather than minimizing (which is the same after flipping a sign, so it's not huge). Here α is again the learning rate. After all the math is bashed out (as is done in the provided lecture notes), the rule is

$$\theta_j \coloneqq \theta_j + \alpha \sum_{i=1}^{m} (y^{(i)} - h_\theta(\mathbf{x}^{(i)})) x_j^{(i)}.$$

The surprising thing about this is that cosmetically, it looks identical to the rule provided for gradient-descent for linear regression. These two algorithms were invented independently, and people at first wondered if they were the same, but since h_θ is different in each case, then these are different algorithms, and in particular will have different results when run on different data.

These algorithms aren't particularly new; computers called Perceptrons used a similar algorithm in the 1960s. In general, logistic regression is a very common machine learning algorithm. Targeted advertisements, for example, often use it or a small variant.

Newton's Method. There are multiple ways of optimizing a function; here's another one called Newton's method. For the presentation in today's lecture, we will consider $\theta \in \mathbb{R}$ rather than $\theta \in \mathbb{R}^{n+1}$. Then, the goal is to maximize the log-likelihood, so define $f(\theta) = \frac{\mathrm{d}}{\mathrm{d}\theta} \ell(\theta)$, so the goal is to find a theta such that $f(\theta) = 0$.

Unlike linear regression, there is no known way to solve this in closed form.[12] However, Newton's method is a very effective way of approximating solutions. Similarly to gradient-descent, this method starts with an initial guess θ_0. Take the tangent like to f at θ_0, and let θ_1 be the place where that line intersects the x-axis. Then,repeat with the tangent line to f at θ_1, and so on.

Now for the math: how does one obtain θ_1 from θ_0? Let $\Delta = |\theta_1 - \theta_0|$. Thus, the slope of f at θ_0 is $f'(\theta_0)$, but also $f(\theta_0)/\Delta$, so one calculates $\Delta = f(\theta_0)/f'(\theta_0)$, so the updating step is

$$\theta^{(i+1)} \coloneqq \theta^{(i)} - \frac{f(\theta^{(i)})}{f'(\theta^{(i)})}.$$

Alternatively, dispense with f entirely to get

$$\theta^{(1)} = \theta^{(0)} - \frac{\ell'(\theta)}{\ell''(\theta)}.$$

Newton's method converges very quickly. Specifically, it has quadratic convergence; each iteration, the error is roughly squared (which sounds bad, but for small errors is of course very good!). This is much faster than gradient descent, though it is more computationally intensive.

In the more general case where $\theta \in \mathbb{R}^{n+1}$, this becomes

$$\theta^{(t+1)} = \theta^{(t)} + H^{-1}(\nabla_\theta J),$$

where H is the Hessian matrix of $\ell(\theta)$. This is explained more fully in the notes and the problem set. Inverting a matrix is generally computationally expensive, which is the primary drawback of this method.

[12] Hello, Galois!

5. Generalized Linear Models: 10/2/13

Though the logistic equation was presented in the last class, it's not clear why it was used; there are lots of smooth functions from 0 to 1. That answer, as well as others, will be given below.

The Exponential Family of Probability Distributions. Recall that the Bernoulli distribution on ϕ is $\mathsf{Bernoulli}(\phi) = P(y = 1; \phi) = \phi$. It is convenient to think of this as the set of all Bernoulli distributions for all parameters ϕ. Similarly, consider the set of all Gaussians $\mathsf{N}(\mu, \sigma^2)$, with elements such as $\mathsf{N}(0, 1)$, $\mathsf{N}(10, 0.5)$, etc.

It turns out that both of these are special cases of the exponential class[13] of distributions.

Definition. A class of distributions is an exponential family if it can be written

$$P(y; \eta) = b(y) \exp(\eta^{\mathrm{T}} T(y) - a(\eta)) \tag{2}$$

for some choices of b, η, T, and a, and such that the resulting function is still a probability distribution. Here, η is called the natural parameter, and $T(y)$ is called the sufficient statistic.

In most but not all examples of exponential distributions, $T(y) = y$. This definition yields a set of distributions, parameterized by η once b, T, and a are fixed; one says "a distribution is in the exponential family" if some choice of parameters b, T, and a yield that distribution, as a set of probability distributions on y.

For example, there must be a bijection $\phi \mapsto \eta$ for some choice of parameters to demonstrate that the Bernoulli distribution: Rewrite the Bernoulli distribution as $P(y; \phi) = \phi^y (1 - \phi)^{1-y}$. Thus,

$$\begin{aligned} P(y; \phi) &= \exp\left(\log\left(\phi^y (1 - \phi)^{1-y}\right)\right) \\ &= \exp\left(\log \frac{\phi}{1 - \phi} y + \log(1 - \phi)\right). \end{aligned}$$

Thus, setting $b(y) = 1$, $T(y) = y$, $a(\eta) = \log(1 - \phi)$, and $\eta = \log(\phi/(1 - \phi))$, one obtains (2). Solving for ϕ, one obtains $\phi = 1/(1 + e^{-\eta})$ (which is just the logistic formula!), and $a(y) = \log(1 + e^{\eta})$.

It turns out there are multiple mappings for which this works, such as multiplying T by $1/2$ and $a(y)$ by 2, but usually there is a most natural one.

Now, consider the Gaussian. For the rest of this lecture, follow the convention that for $\mathsf{N}(\mu, \sigma^2)$, $\sigma^2 = 1$. The simplification, after a bit of algebra, is

$$\frac{1}{\sqrt{2\pi}} \exp\left(-\frac{1}{2}(y - \mu)^2\right) = \underbrace{\frac{1}{\sqrt{2\pi}} e^{-y^2/2}}_{b(y)} \exp\Bigg(\mu y - \underbrace{\frac{1}{2}\mu^2}_{a(\eta)}\Bigg),$$

and the natural mapping is just $\mu \mapsto \eta$. $T(y) = y$ again.

Most of the distributions mentioned in introductory probability classes are exponential: the multivariate Gaussian, the Poisson, the Beta, the Gamma, the Dirichlet, and of course the Exponential distribution. Since these cover a large number of different real-world scenarios, abstracting one away to an exponential distribution allows the use of a few general algorithms (specifically, the GLM described below) in many situations.

The Generalized Linear Model (GLM). Given some distribution $P(y; x, \theta)$, one can use a generalized linear model under the following assumptions:

(1) $y \mid x, \theta \sim \mathsf{ExpFam}(\eta)$ for some choice of parameters.
(2) The goal is to model $P(y \mid x, \theta)$, and to calculate the expected value $E[T(y) \mid x; \theta]$.
(3) The relationship between η and θ is $\eta = \theta^{\mathrm{T}} x$. This assumption leads to the very nice algorithm.

Example 5.1. Suppose one has a bunch of features of tumors, and wants to classify them as malignant or benign. Choose parameters such that $\mathsf{ExpFam}(\eta)$ is the Bernoulli set of distributions, and assume $y \mid x, \theta \sim \mathsf{ExpFam}(\eta)$, and that $\eta = \theta^{\mathrm{T}} x$, and that the algorithm outputs $h_\theta(x) = E[y \mid x; \theta] = P(y = 1 \mid x; \theta)$, since the expected value of a Bernoulli random variable is just the probability that it is zero. Write this probability as ϕ, though there is an unstated dependence $\phi(x, \theta)$. Since we know how η depends on x and θ, this simplifies to

$$h_\theta(x) = \phi = \frac{1}{1 + e^{-\eta}} = \frac{1}{1 + e^{-\theta^{\mathrm{T}} x}}.$$

Thus, the logistic regression algorithm just falls out of the mathematics!

In general, the steps are to:

[13] Here, class just means set. There aren't any fancy set-theoretic difficulties to be found here that I know of.

(1) Write out the formula for the distribution that is being learned.
(2) Show that it is an exponential family for some appropriate choice of parameters.
(3) Setting $\eta = \theta^{\mathrm{T}} x$, write $h_\theta(x)$ in terms of θ and x as in Example 5.1.

This has a big advantage in that the GLM is guaranteed to have a convex optimization problem when one uses maximum likelihood estimation. This means that gradient ascent always finds the global maximum, which is really nice. Thus, the formula

$$\theta_j := \theta_j - \alpha \sum_{i=1}^{m} \left(h_\theta(x^{(i)} - y) \right) x_j^{(i)}$$

also still applies.

The assumptions and machinery behind GLMs are kind of tricky; while the mathematics are all fairly reasonable, the overarching understanding, seeing how the pieces fit together, is harder and can be seen by deriving one of the examples for oneself. For example, a good choice (e.g. on our problem set) is the Poisson model.

Softpass Regression. Using the generalized linear model, one can use the multinomial distribution to obtain a useful algorithm. This is a bit more complex, because $T(y) \neq y$, but it's useful for algorithms that classify y into $k > 2$ choices, which lends itself to many types of applications. Geometrically, this algorithm draws boxes across the data, rather than a separating line (hyperplane).

For this derivation, assume $k = 4$; the lecture notes contain the general case, but this will simplify the derivation a bit.

The first step is to express the multinomial distribution as an exponential one. Let the parameters be $\phi_1, \ldots, \phi_3$ so that $P(y = i) = \phi_i$. However, since the probabilities have to sum to 1, $\phi_4 = 1 - (\phi_1 + \phi_2 + \phi_3)$, so they're not independent. Let $\boldsymbol{\phi} = (\phi_1, \phi_2, \phi_3)$; ϕ_4 exists but is not considered a separate parameter. These can be compressed into more compact notation, similarly to what was done in the case of the Bernoulli distribution earlier: let $T(i) = \mathbf{e}_i$ for $i = 1, \ldots, 3$, and $T_4 = 0$; in general, $T(i) \in \mathbb{R}^{k-1}$ if there were k parameters.

It will be useful to have something called indicator notation, which can be considered as mapping true statements to 1 and false ones to zero: $I\{\text{True}\} = 1$ and $I\{\text{False}\} = 0$. For example, $I\{2 + 3 = 5\} = 1$. This means that $T(y)_i = \{y = i\}$, and

$$\begin{aligned} P(y) &= \prod_{j=1}^{k} \phi_j^{I\{y=j\}} \\ &= \phi_1^{T(y)_1} \phi_2^{T(y)_2} \phi_3^{T(y)_3} \phi_4^{1 - T(y)_1 - T(y)_2 - T(y)_3}. \end{aligned}$$

After some amount of massaging, this becomes

$$= b(y) \exp(\boldsymbol{\eta}^{\mathrm{T}} T(y) - a(\boldsymbol{\eta})),$$

where T is as above, $a(\boldsymbol{\eta}) = -\log \phi_4$, $b(y) = 1$, and $\boldsymbol{\eta} = (\phi_1/\phi_4, \phi_2/\phi_4, \phi_3/\phi_4)^{\mathrm{T}}$. Once these assumptions have been made, along with $\eta_i = \theta_i^{\mathrm{T}} \mathbf{x}$, then under this model,

$$P(y = i) = \frac{\theta_i^{\mathrm{T}} x}{1 + e^{\theta_1^{\mathrm{T}} x} + e^{\theta_2^{\mathrm{T}} x} + e^{\theta_3^{\mathrm{T}} x}}. \tag{3}$$

In order to fit this model, the final step is to write down the learning algorithm: if $(\mathbf{x}^{(i)}, y^{(i)})$ is the training set, with $i = 1, \ldots, m$, then one can make the maximum likelihood estimate

$$L(\theta) = \prod_{i=1}^{m} P(y^{(i)} \mid \mathbf{x}^{(i)}; \theta) = \prod_{i=1}^{m} \prod_{j=1}^{4} \phi_j^{I\{y^{(i)}=j\}},$$

where the ϕ_i are given as in (3). Given a training set, one fits the parameters to the model by writing out the likelihood and maximizing it as a function of θ, which is nice because it's a convex function.

6. Probability and Statistics Problem Session: 10/4/13

"Statistics is rigorous machine learning." – Steven Boyd

This session will give a handwavy definition of probability, followed by useful properties of it for the problem sets. First, consider an example:

Example 6.1. If one rolls a die, the sample space is the set of all possible outcomes: $\Omega = \{1, 2, 3, 4, 5, 6\}$. One might have subsets of these, such as $F = \{1, 2\}$.

Formally, probability is a way of assigning numbers to subsets of a sample space Ω. It is required to satisfy certain axioms:

- $P(A) \geq 0$ for every $A \subset \Omega$.

- If A_1 and A_2 are disjoint sets, $P(A_1 \cup A_2) = P(A_1) + P(A_2)$. Geometrically, assigning probability in a sense like area makes these notions both make sense.
- $P(\Omega) = 1$.
- If $A \subset B$, then $P(A) \leq P(B)$. This also makes sense geometrically: their intersection is A itself.
- More generally, for any sets A and B, $P(A \cap B) \leq \min(P(A), P(B))$. This is akin to the statement that the area of two overlapping sets A and B is less than the area of A (since the intersection is contained in A), and less than the area of B (since it's also contained in B).
- $P(\neg A) = 1 - P(A)$.

There's also something called the Law of Total Probability: if $\{A_i\}$ is any collection of disjoint events, then $\sum_i P(A_i) = 1$.

Next, consider conditional probability. This is the chance of something happening given (already known) that something else happened. For example, for dice the probability of rolling a 1 given that an odd number is rolled is $P(1 \mid 1,3,5) = 1/3$, which differs from the total probability $P(1) = 1/6$. There's a nice formula for this, which is $P(X \mid Y) = P(X \cap Y)/P(Y)$. In terms of a Venn diagram, one just works inside one blob Y rather than the whole space; then, the goal is to ask how much of this blob is X.

Conditional probability leads naturally into independence: two events A and B are independent if $P(A \cap B) = P(A)P(B)$. Intuitively, this means that the two events have no effect on each other, or no causal relationship. It can be more complicated than that, but this is a useful first way to think about it. If A and B are independent events, then $P(A \mid B) = P(A)$; intutively, this says that if two events are independent, then knowing one has no effect on the other.

Though thus far only events have been discussed, but one can speak of random variables. A random variable is a variable that could take on several values corresponding to different events. Though this sounds questionable, it's mostly a piece of useful notation, as it allows for one to chart the different outcomes of an experiment. In the discrete case, a random variable X takes on at a value x the probability $P(x)$, which makes a lot of sense, but in the continuous case things are a bit weirder, since at a single value $P(x) = 0$ is perfectly fine, but over some *interval* $[x_1, x_2]$ the probability $P([x_1, x_2])$ is nonzero, and in fact is given by $P(a \leq x \leq b) = \int_a^b p(x)\,\mathrm{d}x$. In the discrete case, the random variable described above gives the function called the probability mass function; in the continuous case, $p(x)$ is called the probability density function.

Interestingly, the probability density function may be larger than 1 at some value, as long as $\int_{-\infty}^{\infty} p(x)\,\mathrm{d}x = 1$. For example, if $p(x) = 0$ if $|x| \geq 1/4$ and $p(x) = 2$ otherwise, then p is a valid probability density function, but it is everywhere larger than 1.

In a maximum likelihood estimate, one has a distribution parameterized by θ to maximize $P(y \mid x, \theta)$. This can be thought of as changing the probability distribution (which is what is given by θ) to fit the model.

Another useful idea is that of expectation. The expectation of a function g is $E[g(x)] = \sum_x g(x)p(x)$. This is the "average value" of $g(x)$, weighted by the probability of each input x. For example, in a lottery with a 0.25 chance of winning \$100, 0.5 chance of winning \$50, and 0.25 chance of winning nothing, then in expectation, one would win \$50 from this lottery (which is the average, albeit over a large number of games). For continuous variables, the sum becomes an integral:

$$E[g(x)] = \int_{-\infty}^{\infty} g(x)p(x)\,\mathrm{d}x,$$

where $p(x)$ is now a probability density function, not a probability mass function.

Some useful properties of expectation:

- The expected value of a constant is that constant: $E[a] = a$ for $a \in \mathbb{R}$.
- Constants can be pulled out of an expectation: $E[af(x)] = aE[f(x)]$.
- Expectation commutes with sums: $E[f(x) + g(x)] = E[f(x)] + E[g(x)]$.

There's another quantity, called variance. This is defined as $\mathrm{Var}(X) = E[(X - E[X])^2]$. This measures how far all of the data is from the mean. The units of variance are the units of the x-axis, squared, which could be important in some sorts of problems.

It's possible to derive a useful formula for the variance: let $\mu = E[X]$ be the mean of the dataset.

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

since μ is a constant. Sometimes this is also written as $E[X^2] - E[X]^2$. This tends to be much easier to compute than the formula given in the definition.

Some properties of variance:

- If a is a constant, then $\mathrm{Var}(a) = 0$.
- $\mathrm{Var}(af(x)) = a^2 \mathrm{Var}(f(x))$, where a is again a constant. Notice that this is different from the result for the expected value!

It's possible to have multiple random variables in a scenario, in which case one has joint probability distributions. Then, one takes a double sum or double integral over all values (depending on discreteness or continuity) to get 1. Similarly, one has

$$P(Y=y)=\sum_x P(X=x,Y=y).$$

This is useful if one is given the joint distributions, but not the individual ones. One can also write down a conditional distribution: $P_{Y|X}(Y\mid X)=P_{X,Y}(X,Y)/P_X(X)$. This works in both the continuous and the discrete cases. Additionally, one has Bayes' rule: $P(A\mid B)=P(B\mid A)P(A)/P(B)$. This has far-reaching implications into law, medicine, etc. relating to how two important events are related, in some sense reversing a correlation to understand if it is causation. I don't know why this was lumped into the section on multiple variables, because it's so much more general than that.

With two random variables, there's a generalization of this to covariance: $\mathrm{Cov}[X,Y]=E[(X-E[X])(Y-E[Y])]$. Notice that if $X=Y$, then $\mathrm{Cov}(X,X)=\mathrm{Var}(X)$; the covariance is large implies that X is large when Y is large, and vice versa. If X and Y are not very related (no correlation), the covariance is about zero, but if they are oppositely related, then the covariance is a negative number. This can take on any real-numbered value depending on the dataset in question.

The covariance matrix Σ of n variables $X_1,\dots,X_n$ has $(i,j)^{\text{th}}$ entry $\mathrm{Cov}(X_i,X_j)$ (so that the diagonal entries are the variances of these n variables). This allows one to extend the Gaussian to multiple dimensions: in the single-variable case, one has

$$f(x;\mu,\sigma)=\frac{1}{\sigma\sqrt{2\pi}}e^{-(x-\mu)^2/2\sigma^2},$$

but in the multidimensional case, σ becomes the matrix Σ:

$$f(x_1,\dots,x_n;\mu,\Sigma)=\frac{1}{(2\pi)^{n/2}\det(\Sigma)^{1/2}}\exp\left(-\frac{1}{2}(\mathbf{x}-\mu)^{\mathrm{T}}\Sigma^{-1}(\mathbf{x}-\mu)\right). \tag{4}$$

Notice that the mean becomes a vector of means of $X_1,\dots,X_n$.

7. Generative Learning Algorithms: 10/7/13

Given some data that can be classified into two parts, logistic regression attempts to find a line that separates them. This is an example of a discriminative algorithm. However, one might use a different approach; instead of finding $P(y\mid x)$ directly (or $h(x)$), one could learn both $P(x\mid y)$ and $P(y)$: in the tumor classification example, what do the features look like in a benign tumor and a malignant tumor? Here, $P(y)$ is called the class prior, which is useful for knowing the odds of malignancy already. By Bayes' rule, these allow one to compute $P(y\mid x)=P(x\mid y=1)P(y=1)/P(x)$. Thus, $P(x)=P(x\mid y=1)P(y=1)+P(x\mid y=0)P(y=0)$.

Gaussian Discriminant Analysis (GDA). For this algorithm, assume the data is from a continuous distribution, so that $\mathbf{x}\in\mathbb{R}^n$. Notice that $x_0=1$ is not used here. Furthermore, it is assumed that $P(\mathbf{x}\mid y)$ is (multivariate) Gaussian.

The multivariate Gaussian is a generalization of the single-variable normal distribution to vectors. It takes the form $\mathbf{z}\sim\mathsf{N}(\mu,\Sigma)$, where $\mathbf{z},\mu\in\mathbb{R}^n$ and $\Sigma\in\mathbb{R}^{n\times n}$. Then, $E[\mathbf{z}]=\mu$, and $\mathrm{Cov}(\mathbf{z})=E[(\mathbf{z}-\mu)(\mathbf{z}-\mu)^{\mathrm{T}}]=\Sigma$. For example, when $n=2$,

$$\Sigma=\begin{pmatrix}E[(z_1-\mu_1)^2] & E[(z_1-\mu_1)(z_2-\mu_2)]\\ E[(z_2-\mu_2)(z_1-\mu_1)] & E[(z_2-\mu_2)^2]\end{pmatrix}$$

If the values on the diagonal of Σ are made larger, then the variables are more correlated, and the data clusters more strongly (thinner, taller graph). The off-diagonal values increase correlations between two variables, distorting the graph into an ellipse. This represents that the two values are more strongly correlated. Then, varying μ changes the center of the data. The probability density function is as in (4). Nobody actually memorizes this, but make sure it's around where you can refer to it. Then, $P(y)=\phi^y(1-\phi)^{1-y}$, so that $P(\mathbf{x}\mid y=0)$ has a multivariate Gaussian distribution with mean μ_0, and $P(\mathbf{x}\mid y=1)$ has a multivariate Gaussian distribution with mean μ_1. The same covariance matrix is used in both calculations. Thus, with parameters ϕ,μ_0,μ_1, and Σ and a training set $(\mathbf{x}^{(1)},y^{(1)}),\dots,(\mathbf{x}^{(m)},y^{(m)})$, one wishes to maximize the joint likelihood

$$L(\phi,\mu_0,\mu_1,\Sigma)=\prod_{i=1}^m P(\mathbf{x}^{(i)},y^{(i)})=\prod_{i=1}^m P(\mathbf{x}^{(i)},y^{(i)})P(y^{(i)}).$$

Compare this to the likelihood function used in a discriminative algorithm, which looks very different. This is because in that case, $P(y\mid\mathbf{x})$ is not defined, so a different tack had to be taken. After taking the log-likelihood, the maximum likelihood

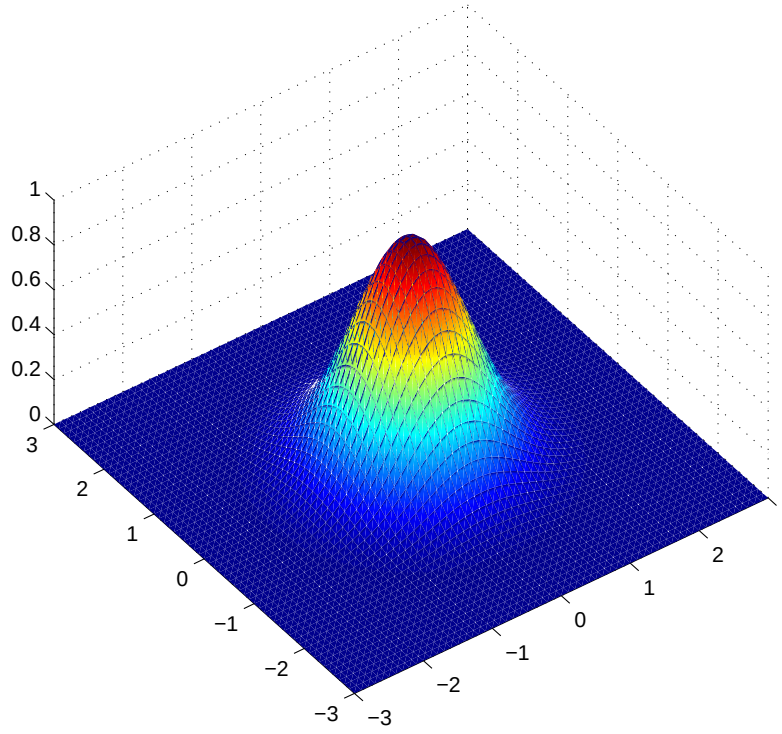


FIGURE 3. A graph of the multivariate Gaussian when $n = 2$. Source: Wikipedia.

estimate takes on these values:

$$\phi = \frac{1}{m}\sum_{i=1}^{m} y^{(i)} = \frac{1}{m}\sum_{i=1}^{m} 1\{y^{(i)} = 1\}.$$
$$\mu_0 = \frac{\sum_{i=1}^{m} 1\{y^{(i)} = 0\}\mathbf{x}^{(i)}}{\sum_{i=1}^{m} 1\{y^{(i)} = 0\}}.$$

This is the mean of all of the examples $\mathbf{x}^{(i)}$ such that $y^{(i)} = 0$, which is what makes the most sense. Thus, μ_1 is calculated in effectively the same way:

$$\mu_1 = \frac{\sum_{i=1}^{m} 1\{y^{(i)} = 1\}\mathbf{x}^{(i)}}{\sum_{i=1}^{m} 1\{y^{(i)} = 1\}}.$$

Σ is given in the lecture notes, since it's a little messier. Then, the prediction for y is

$$\arg\max_y P(y \mid \mathbf{x}) = \arg\max_y \frac{P(\mathbf{x} \mid y)P(y)}{P(\mathbf{x})}.$$

If you haven't seen arg max before, it just means to choose the value of y such that the quantity is maximized: $\min_z (z-5)^2 = 0$, but $\arg\min_z (z-5)^2 = 5$.

In some sense, rather than partitioning the plane into two parts, one calculates Gaussians for each of the positive and negative datasets, and asks which one is more likely. Thus, the decision boundary is actually different in these two cases: if one plots $P(y = 1 \mid \mathbf{x})$ (for example, in a single-variate distribution), it has a logistic form as one goes from μ_0 to μ_1, given the assumptions of GDA. However, the converse is not true: if this function is logistic, the assumptions of the generative model might not hold. This means that generative algorithms make stronger assumptions than discriminatory ones. Specifically, these assume $y \sim \text{Bernoulli}(\phi)$ and $\mathbf{x} \mid y = 0$ and $\mathbf{x} \mid y = 1$ are both Gaussian. In these cases, of course, this model will perform better, but otherwise it might be smarter to try something else.

Here's another set of assumptions that make $P(y = 1 \mid \mathbf{x})$ logistic: that $\mathbf{x} \mid y = 0 \sim \text{Poi}(\lambda_0)$, $\mathbf{x} \mid y = 1 \sim \text{Poi}(\lambda_1)$, and $y \sim \text{Bernoulli}(\phi)$ again. This is actually a much more general assumption. However, logistic regression can handle any

exponential model, which shows that it is more robust in some cases; however, generative models are often more useful for small datasets. There are many such assumptions that lead it to be logistic.

Naïve Bayes. One nice advantage of GLMs is that they are fairly easy to implement and train, since there's no iteration, and so on. Thus, though they might not be the best algorithms for the job, generative algorithms are useful for quick prototyping (such as for the project).

A good example of this is the Naïve Bayes algorithm, which is rarely the best algorithm for the problem, but is fairly quick to set up.

For a motivational example, consider spam detection, in which one examines an email and tries to classify it as spam or ham. More generally, one might want to sort emails into categories such as social, professional, etc., or eBay might wish to classify its auctions into different genres (which is actually what happens; they don't ask the user to choose a category).

Emails are interesting because they are different lengths, so obtaining a feature set requires a little thinking. A common way to do this is to take a list of English words $\{\text{a}, \text{aa}, \dots, \text{zymurgy}\}$, and let the feature vector of an email be a vector of the size of this list with a 1 in a given entry if the corresponding email has that word and a 0 if otherwise.[14] So write $n = 10000$, so that $\mathbf{x} \in \{0,1\}^{10000}$. Thus, there are 2^{10000} possibilities for $\mathbf{x}$, so modeling this distribution explicitly over all of these outcomes (e.g. a multinomial distribution), unfortunate things clearly happen, so a different approach is necessary.

Naïve Bayes adds one additional assumption, called the conditional independence assumption or the Naïve Bayes assumption: that the x_i are independent given y. This means that

$$P(x_1, x_2, x_3, \dots, x_{10000} \mid y) = \prod_{i=1}^{10000} P(x_i \mid x_1, \dots, x_{i-1}, y),$$

which is always true, but under the Naïve Bayes assumption, this simplifies to

$$P(x_1, x_2, x_3, \dots, x_{10000} \mid y) = \prod_{i=1}^{10000} P(x_i \mid y).$$

One way of explaining this is: suppose x_3 is the word "buy," and x_{1000} is the word "drugs." Plenty of emails associate the two words. But: *once you know an email is spam*, there's no correlation between these two words. This is not completely true, but it is convenient enough to be useful, and is the "conditional" in conditional independence.

The parameters of this model are $\phi_{j|y=1} = P(x_j = 1 \mid y = 1)$, or the probability of a word appearing in a spam message, and $\phi_{j|y=0} = P(x_j = 1 \mid y = 0)$ (probability of a word appearing in a ham message), and $\phi_j = P(y = 1)$. Then, one can write down the joint likelihood

$$L(\phi_y, \phi_{j|y=1}, \phi_{j|y=0}) = \prod_{i=1}^{m} P(\mathbf{x}^{(i)}, y^{(i)}).$$

Once one has a training set, one can fit the parameters by making a maximum likelihood estimate of the parameters as

$$\phi_y = \frac{1}{m} \sum_{i=1}^{m} 1\{y^{(i)} = 1\}$$

$$\phi_{j|y=1} = \frac{\sum_{i=1}^{m} 1\{x_j^{(i)} = 1, y^{(i)} = 1\}}{\sum_{i=1}^{m} 1\{y^{(i)} = 1\}} \tag{5}$$

$$\phi_{j|y=0} = \frac{\sum_{i=1}^{m} 1\{x_j^{(i)} = 1, y^{(i)} = 0\}}{\sum_{i=1}^{m} 1\{y^{(i)} = 0\}} \tag{6}$$

For the $\phi_{j|y}$, the intuition is to ask: of all emails in this bucket (spam or ham), what fraction of them contain this word?

To implement a Naïve Bayes classifier, compute these parameters and then classify by setting $P(y \mid \mathbf{x}) = P(\mathbf{x} \mid y)P(y)$, which follow from the above.

In summary, generative models look at each component of the classification separately, forming independent models that lead to the classification.

8. Laplace Smoothing: 10/9/13

"I actually know the president of [the University of] Arizona pretty well. He's a nice guy, an interesting guy."

[14] One more practical strategy is to take only the 10000 words most commonly seen in the recipient's inbox.

Though the spam detector model was discussed in the last section, it didn't explicitly mention what it did at classification time. The goal is to predict $P(y=1 \mid \mathbf{x}) = P(\mathbf{x} \mid y=1)P(y=1)/(P(\mathbf{x} \mid y=0)+P(\mathbf{x} \mid y=1)P(y=1))$.

As currently discussed, this algorithm only partially works. The maximum likelihood estimate for the parameters $\phi_{j|y}$ given in (5) and (6). This is fine, but suppose that after a few months one starts getting email about, for example, NIPS (a machine learning project).[15] Suppose that NIPS corresponds to wors 17 in this classifier. Then, until the first such NIPS email, $P(x_{17}=1 \mid y=1)=0$, but $P(x_{17}=1 \mid y=0)=0$, since there are 0 examples over all spam or non-spam emails. This is an issue particular to generative models, because they calculate these probabilities separately. This means that when Bayes' rule is used to calculate $P(y=1 \mid \mathbf{x})$, it evaluates to $0/0$, which is fairly useless.

The rationale behind this is that statistically, one shouldn't assume that the probability of something is zero if it hasn't been seen before, but that's how the MLE works.

Laplace Smoothing. For now, put aside Naïve Bayes, and just estimate a single Bernoulli random variable. For example, suppose a football fan attended every Stanford away game in 2009. Specifically, Stanford lost to Wake Forest, then lost to Oregon, then lost to Arizona and to… Caltech. Plausible.

What are the odds of Stanford winning the final game, against Oklahoma? Let y be the random variable indicating this, so that y is Bernoulli. Then, $P(y=1)$ is estimated as the number of wins over the total number of games, so it is equal to zero.

Since it's a bad idea to conclude that something is impossible based just on its absence in a single dataset, Laplace smoothing adds one win and one loss to the dataset. Then, the estimated probability of a win becomes 1/6, which is a bit more reasonable.[16]

Similarly, if $y \in \{1,\dots,k\}$ is given by a multinomial distribution, one example of $y=i$ is added for each i, so the estimated probability is

$$P(y=i) = \frac{\sum_{i=1}^{m} 1\{y^{(i)}=i\}+1}{m+k}.$$

Returning to Naïve Bayes, the new maximum likelihood estimates are

$$\phi_{j|y=0} = \frac{\sum_{i=1}^{m} 1\{x_i^{(i)}=1, y^{(i)}=0\}+1}{\sum_{i=1}^{m} 1\{y^{(i)}=0\}+2},$$

and similarly for $\phi_{j|y=1}$. This solves the issue of zero probability, so one ends up with $P(y=1 \mid \mathbf{x})=\varepsilon$ for some small ε, but $\varepsilon \neq 0$, which is nice. The spam classifier thus indicates that it doesn't know much about this word. With this correction, the spam classifier is actually pretty effective.

To generalize this problem a little bit, we have $x_i \in \{1,2,\dots,k\}$, which is useful in some other contexts than word processing. Now, $P(x_i \mid y)$ takes on a multinomial distribution. This applies in the real world when one wants to classify some dataset into more than two parts. Often, this is done by discretizing the features, (e.g. houses less than 400 square feet, houses between 400 and 800 square feet, etc.).

Multinomial Event Model. But there is another variation on Naïve Bayes, specific to spam classification (or text processing). One downside of the algorithm described thus far is that it ignores how often a word appears in a message. Thus, a new event model is necessary. Given a piece of email, there is once again a feature vector $\mathbf{x} \in \mathbb{R}^n$, where n is the number of words in the email. Thus, emails of different lengths will have feature vectors of different sizes. The i^{th} entry in the feature vector is the index of the i^{th} word in the email in the big list of words used to construct the feature vectors in the earlier model (called the multivariate Bernoulli event model for some terrible reason).[17]

Once again, use the Naïve Bayes assumption that the x_j are conditionally independent given y. Then,

$$P(\mathbf{x},y) = P(\mathbf{x} \mid y)P(y) = P(y)\prod_{j=1}^{n} P(x_j \mid y),$$

which looks superficially similar to the previous probability, but all of the meaning is different, e.g. n is now just the length of the email. The Naïve Bayes assumption is that the identities of the words are conditionally independent given y.

The parameters are $\phi_y = P(y=1)$ again, as well as $P(x_j=k \mid y=0)=\phi_{k|y=0}$. Assume this is independent of j (i.e. that the index of a word in an email is not important once it is known to be ham or known to be spam). Then, the MLEs of these

[15] Hint, hint. The paper submission deadline is in May.

[16] Laplace actually invented this technique, for use in philosophy! He was interested in the philosophy of cause and effect; there is no reason to assume that the sun will rise tomorrow given that it has risen every morning. Laplace showed that assuming a uniform prior on the sun rising tomorrow (and a Bayesian interpretation of probability), the Laplace estimator is optimal; if you've seen the sun rise 10000 times, then the probability that the sun will rise tomorrow is 10001/10002.

[17] "I'm actually pretty good friends with the guy that wrote that paper that introduced that term, and I think even he regrets introducing it."

parameters are

$$\phi_{k|y=0} = \frac{\sum_{i=1}^{m} 1\{y^{(i)}=0\} \sum_{j=1}^{n_i} 1\{x_j^{(i)}=k\}}{\sum_{i=1}^{m} 1\{y^{(i)}=0\} n_i},$$

where n_i is the length of the i^{th} email, so that $\mathbf{x}^{(i)} \in \mathbb{R}^{n_i}$. The numerator looks at all spam emails and counts up the number of times a given word is counted, and the denominator normalizes this by the total length of all ham emails.

Though this was only written for ham, there is a $\phi_{k|y=1}$ which looks very similar. However, we need to add Laplace smoothing to it, yielding

$$\phi_{k|y=0} = \frac{1 + \sum_{i=1}^{m} 1\{y^{(i)}=0\} \sum_{j=1}^{n_i} 1\{x_j^{(i)}=k\}}{10000 + \sum_{i=1}^{m} 1\{y^{(i)}=0\} n_i}.$$

If 10000 seems arbitrary, then it is because that is the number of words in the dictionary. This classifier is better at handling variable-length data.

As mentioned, this is not a particularly competent algorithm. However, it is easy to build and trains very quickly, so where it really shines is where one wants to quickly implement a learning algorithm on some dataset. Then, one can look at all of the mistakes that algorithm makes and fix them. This is particularly effective because one's gut feelings about what is easy, difficult, or effective tend to be misleading, so getting some quick insight or feedback on a quick project (such as this class' project) is very helpful. Note that the professor, one of the leading experts on spam and machine learning, doubts his intuition, which is significant.

Spammers have of course tried to work around these classifiers. The most obvious solution is to misspell words to trip up the classifier, and some other spammers also spoof their email headers. And on the other side, the creation of honeypots has lead to an immense amount of training data. Thus, there are lots of potential approaches to a project, which are selected by gut feeling surprisingly often. However, clever or complicated models sometimes have hidden problems behind them that make them not work as effectively, so especially in industry or elsewhere where a working program is better than a clever one, looking for a simpler algorithm's mistakes is easier than doing something fancy, and is likely to end up being more effective.

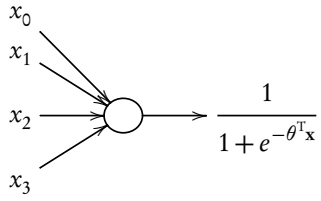
Thus, a good starting point is to initialize a dataset and quickly build an end-to-end system, even if it's known to not be very good. Then, it's easier to find places to focus one's time and optimize the algorithm. Notice how incompatible this is with the advice from CS 106A, but it jives with the notion that premature optimization is the root of all evil.

Other than prototyping, Naïve Bayes doesn't have too many uses. In some cases, the prototype is sufficient for the application, and in others, it's so inefficient that it makes one rethink their data set. And there are plenty of other quick and dirty ways to prototype an algorithm, depending on the problem itself.

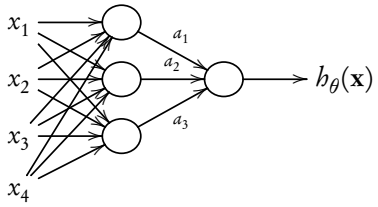
Nonlinear Classifiers and Neural Networks. Neural networks are an algorithm that were very popular about twenty years ago, and then fell out of favor relative to SVMs and the like. Though there has been a slight resurgence in its use recently, it is still a difficult technique to use. There is a lot of stuff happening in neural network research, though.

To motivate this, suppose one has a classification problem with a very clear quadratic decision boundary (or a more generally nonlinear one). How would one modify this to return a nonlinear boundary?

The following diagram can be used to represent logistic regression.



The circle in the middle is some computational unit that derives the parameters. A neural network, originally designed to model how the brain works, took logistic regression to represent a neuron. This is a bad model, but not completely awful, so all that a neural network does is take these nodes and string them together, as so:



This diagram represents the following computation: $a_1 = g(\mathbf{x}^T\theta^{(1)})$, $a_2 = g(\mathbf{x}^T\theta^{(2)})$, and $a_3 = g(\mathbf{x}^T\theta^{(4)})$, and $h_\theta(\mathbf{x}) = g(\mathbf{a}^T\theta^{(4)})$, where $\mathbf{a} = \begin{bmatrix} 1 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}$.

Thus, $h_\theta(\mathbf{x})$ is a function of the parameters $\theta^{(1)}, \dots, \theta^{(4)}$, so the goal is to minimize the squared error, or some optimization objective akin to

$$\min_\theta \frac{1}{2} \sum_{i=1}^{m} (h_\theta(\mathbf{x}^{(i)}) - y^{(i)})^2.$$

This algorithm can output nonlinear classifiers because the sigmoid function is nonlinear, so one learns some set of nonlinear features $\{a_1, a_2, a_3\}$, upon which logistic regression can be preformed again.

The maximization problem is often solved with gradient descent, though in this specific context it tends to be called back-propagation. However, this optimization problem is nonconvex, so one might not necessarily find the global optimum, but it seems to work well enough in practice, if slowly.

More complicated neural networks exist, with several layers of stacked "neurons." One use of this was handwriting analysis for zip codes (though today, this is preformed on the entire address). It as able to handle some pretty pathological examples of numbers. Another example is text-to-speech conversion.

9. MATLAB TUTORIAL: 10/11/13

This class heavily stresses MATLAB[18] for the implementation. This is because it makes for an effective prototyping tool for quick development and testing, and after everything is set up it can be translated to a faster or lower-level language.

Everything in this lecture can be found in the file `session.m`.

In MATLAB, all numbers are treated as doubles, so `1/2` evaluates to 1/2. Most operations are the same as in other languages, though $\neq$ is `~=` and xor is `xor( , )`. Strings are denoted with single quotes, and expressions are perfectly fine (e.g. `c = 3 >= 1`). Every operation has some sort of output, which can be suppressed by appending a semicolon.

How about printing? If one sets `a = pi`, which does what one might expect, printing `a` truncates it to four places. However, the `disp` and `sprintf` commands allow one to work around this: `disp(sprintf('6 decimals: %0.6f', a))` will print π to six decimal places, preceded by the words "6 decimals." Some commands make this easier: `format long` permanently sets decimals to display with higher accuracy, and `format short` returns to four digits.

The power of MATLAB is in its vectors and matrices. For example, one generates the matrix $\mathtt{A} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$ with the command `A = [1 2; 3 4; 5 6]`, and one generates the row vector $\mathtt{v} = [1\ 2\ 3]$ as `v = [1 2 3]`, and the column vector $\mathtt{v}' = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$ as `v' = [1; 2; 3]`. Vectors can be automatically generated: `v = [1:0.1:2]` generates a vector that steps from 1 to 2 with a step size of 0.1 (which defaults to 1; `1:6` is equivalent to the array `[1 2 3 4 5 6]`).

Some commonly used matrices have shortcuts: `ones(m,n)` creates an $\mathtt{m} \times \mathtt{n}$ matrix of entirely ones (which reduces to a vector if `m` or `n` is 1). The `zeros` command does the same thin, but returns the zero matrix. Similarly, the `rand` function takes arguments in the same way and returns a matrix with random numbers uniformly distributed in $[0, 1]$, The `randn` function returns random numbers chosen from the standard unit normal distribution. All of these are called in exactly the same way.

The empty matrix or vector, a 0×0 matrix, is given by `e = []`, and the $\mathtt{n} \times \mathtt{n}$ identity matrix is given by `eye(n)`.

If a matrix is read from a file (for example) and you don't know its dimension, you can use the `size` command, which returns a vector of the dimensions of the matrix (or vector, in which case one of the entries is 1). Then, the `length` command returns the length of the longest directory. This is mostly useful for vectors, and can do unexpected things with matrices if you're not careful.

Now, on to loading files: the `pwd` command does exactly what it does in any actual shell, and so do `cd` and `ls`. Then, the command `load filename` will read the files and store their contents as directed by the file. Then, one can use the `who` command to determine which variables exist (and therefore which were affected by loading the file), and `whos` provides more information (e.g. the dimensions of the variables). The `clear` command deletes a given variable, but without an argument it deletes *every* variable, so be careful! Dually, the `save filename variable` saves the given variable to the specified file. Note that Octave appends an Octave-specific header unless one passes the `ascii` flag to `save`.

[18] Or its open-source equivalent, Octave.

The command `v = A(1:10, 1)` returns the first column of the first ten rows of `A` and places it into `v`. Notice that everything is one-indexed! The second row of `A` is given by `A(2, :)`, and the second column by `A(:, 2)`. The word `end` is understood to be the last column or row, so `A(1, end)` returns the last row of `A`.

There are lots of interesting indexing conventions in MATLAB; for example, `A(1, 3)` returns the $(1,3)^{\text{rd}}$ element of `A`, but `A[1,3], :)` returns the first and third columns of `A`. Matrices can be edited, as `A(:, 2) = [10; 11; 12]`, or even joined, as `A = [A, [100; 101; 102]]`. This usually is efficient if you don't abuse it too strongly. Finally, the notation `A(:)` returns all of the elements of `A`, flattened into a vector.

Matrix operations are pretty nice. Matrix multiplication (non-pointwise) is `A*B`, and pointwise multiplication is `A .* C`. Each of these only works when the dimensions are valid. Element-wise powers of a matrix (taking each element to some power) is `A .^ 2`, and the inverse of each element is `1 ./ A`. Then, logarithm, exponential, negation, addition with scalars, and absolute value are all element-wise by default. The transpose of a matrix `A` is `A'`.

There are lots of functions available by default in MATLAB, some of which will be discussed here. If `a` is a vector, then `max(a)` returns the maximum value of the list. Alternatively, assigning `[val, ind] = max(a)` stores the maximum in `val` and the argmax in `ind`. The `help` command accepts functions as arguments and returns documentation on them. The `find` function returns the indices matching the predicate: for example, `find(a < 3)` returns the indices in `a` for which the element is less than 3. The command `magic(n)` returns an `n` $\times$ `n` magic square.

The `sum` command allows summing across lists and matrices. If `A` is a matrix, then `sum(A, 1)` is the list of its column sums, and `sum(A, 2)` is the list of row sums. Maximums also work: `max(A, [], 1)` returns the row-wise maximum, and `max(A, [], 2)` returns the column-wise maximum. If no other arguments are passed, `sum(A)` computes the sum of all elements, and `product` is analogous.

The inverse of a matrix is given bt `A^-1`. The pseudoinverse of a not necessarily invertable matrix can be given by `pinv(A)`. The function `isempty` returns whether a list or matrix is empty, and `numel(A)` returns the number of elements in a matrix, akin to the product of the elements in `size(A)`.[19]

One can do matrix reshaping. Suppose `A` is a 4×3 matrix. To rearrange `A` to have four rows and three columns, but in th same order, use the command `reshape(A, [4,3])`. The general syntax is the matrix first, and then the dimension. Note that the resulting matrix is not equal to the transpose A^{T}. More exciting possibilities exist, such as `reshape(A, [2,6])`.

If `A` is a matrix and `v` a vector, then `A + v` returns `A` with `v` added to every column. One can do more interesting things with `repmat(v, [1,4])`, which makes a matrix of four columns, each column of which is `v`. A function called `bsxfun` allows for more nifty binary operations; for example, `bsxfun(@plus, A, v)` does the same as the operations described above by expanding the third argument into a matrix of the same size as the second argument, and then takes the first argument and acts on the rest.

Finally, it will be useful to know how to use the graphical aspect of MATLAB.[20] Let `t` be some array, such as `t = [0:0.01:1]`, and then `y1 = sin(2*pi*4*t)`, so that y_1 is a function. Then, one plots y_1 against `t` as `plot(t,y1)`. More than one graph is allowed on the same plot, but the `hold on` command must be used to signify this. Further arguments to the `plot` function determine the style of the graph; for example, `plot(t,y1,'r--')` will color it red (and the rest of the string can determine the line style, and so on). Commands such as `xlabel`, `ylabel`, and `title` accept strings and do the obvious thing.

Loops are reasonable: the syntax is `for i = arr`, where `arr` is some sort of array. Then, the loop is terminated with an `end` statement. The `while` loop is similar: `while length(w) < 3 ... end`.

Control statements are also reasonable: `if w(1) == 0 ... end`. Chains of cascading `else` statements are accomplished with `elseif` and then finally `else`.

Finally, we will implement the logistic regression function in MATLAB, which will provide a useful example of working code. Recall that the formula was given by $\nabla_\theta = (y - h_\theta(\mathbf{x}))\mathbf{x}$, leading to the update rule $\theta \coloneqq \theta + \alpha X^{\mathrm{T}}(\mathbf{y} - h_\theta(X))$. The important part is that J is no longer part of this, which makes the logic and implementation easier.

```
% Matlab functions are a little bit interesting. The function name must be the
% same as the filename.

function [theta, ll] = logistic_grad_ascent(X,y)

% rows of X are training samples
% rows of Y are corresponding 0/1 values

% output ll: vector of log-likelihood values at each iteration
% ouptut theta: parameters

alpha = 0.0001;
```

[19] Please do not confuse the `numel` function with the Pokémon Numel.

[20] Any good mad scientist understand the importance of plotting.

```
[m,n] = size(X);

max_iters = 500;

% It's necessary to append a column of ones so that there are the correct
% number of parameters, so that there is a nonzero intercept term.
X = [ones(size(X,1),1), X];

theta = zeros(n+1, 1);  % initialize theta
for k = 1:max_iters
  % This requires the file sigmoid.m to be in the same directory.
  % A copy can be found at
  % http://cs229.stanford.edu/section/matlab/sigmoid.m
  hx = sigmoid(X*theta);
  theta = theta + alpha * X' * (y-hx);
  % log-likelihood
  ll(k) = sum( y .* log(hx) + (1 - y) .* log(1 - hx) );
end
```

10. Support Vector Machines: 10/14/13

Support vector machines (SVMs) are among the most powerful off-the-shelf learning algorithms. The goal is to build a classifier that can handle nonlinear data that doesn's necessariy have a clear margin of separation. (That said, the first algorithm we consider will only work in the case of linearly separable data.)

The Functional and Geometric Margins. The SVM is a classification algorithm. Recall that such a classifier of the form $h_\theta(\mathbf{x}) = g(\theta^{\mathrm{T}}\mathbf{x})$; if $\theta^{\mathrm{T}}\mathbf{x} > 0$, then predict 1, and otherwise predict 0. For a good algorithm, one would want $\theta^{\mathrm{T}}\mathbf{x} \gg 0$ if $y^{(i)} = 1$, and $\theta^{\mathrm{T}}\mathbf{x} \ll 0$ if $y^{(i)} = 0$.

Now, look at the idea of a geometric margin. Some set of data is separable if it is possible to achieve zero training error. Then, there are multiple different lines that can be used to separate this data. One might want a way of measuring how "good" a line is under this schema, which is the geometric margin.

In order to do this, it will be necessary to introduce a slight change in notation, which will follow us throughout the discussion on SVMs. Thus:

- Instead of $y \in \{0,1\}$, one writes $y \in \{1,1\}$.
- h outputs values in $\{-1,1\}$ as well, rather than being a smooth function.
- $g(z)$ is effectively the sign function: $g(z) = 1$ if $z \geq 0$, and $g(z) = -1$ otherwise.
- The hypothesis is now written $h_{\mathbf{w},b}(\mathbf{x}) = g(\mathbf{w}^{\mathrm{T}}\mathbf{x} + b)$, where $\mathbf{w} \in \mathbb{R}^n$ and $b \in \mathbb{R}$. Notice that $\mathbf{w}$ and $\mathbf{x}$ are in $\mathbb{R}^n$, not $\mathbb{R}^{n+1}$; given some parameters $\theta \in \mathbb{R}^{n+1}$ can be separated out into $b = \theta_0$ and $\mathbf{w} = (\theta_1, \ldots, \theta_n)^{\mathrm{T}}$.

Definition. Using this notation, the functional margin of $(\mathbf{w}, b)$ with respect to $(\mathbf{x}^{(i)}, y^{(i)})$ is $\widehat{\gamma}^{(i)} = y^{(i)}(\mathbf{w}^{\mathrm{T}}\mathbf{x}^{(i)} + b)$. The functional margin of the whole data set is $\widehat{\gamma} = \min_i \widehat{\gamma}^{(i)}$.

A large margin is good, because if $y^{(i)} = 1$, then the margin is large if $\mathbf{w}^{\mathrm{T}}\mathbf{x}^{(i)} + b \gg 0$, and similarly, if $y^{(i)} = -1$, then the margin is better if $\mathbf{w}^{\mathrm{T}}\mathbf{x} + b \ll 0$. The overall functional margin represents the worst case, the data point that is least well classified.

One interesting nuance is that if $\mathbf{w}$ is multiplied by some scalar λ, then the functional margin is also multiplied by λ, but the line of separation doesn't actually change. Thus, it might be useful to impose the normalization constraint $||\mathbf{w}|| = 1$ and $b = b/||\mathbf{w}||$. However, it will eventually be more useful if $\widehat{\gamma}$ can be scaled, so this is not done.

Definition. The geometric margin of a data point is its signed distance from the boundary line: $\gamma^{(i)} = (y^{(i)}\mathbf{w}^{\mathrm{T}}\mathbf{x} + b)/||\mathbf{w}||$. The total geometric margin is, again, $\gamma = \min_i \gamma^{(i)}$.

This means that $\gamma^{(i)} = \widehat{\gamma}^{(i)}/||\mathbf{w}||$. This is convenient; it captures the idea that normalization is a good thing. During the derivation of the SVM, optimizing the geometric margin will generally make more sense, but occasionally the functional margin will also be useful.

Like the functional margin, the geometric margin is better if it is larger, which should be no surprise considering their relationship. Notice also that if $||\mathbf{w}|| = 1$, then $\widehat{\gamma} = \gamma$, and more generally scaling $\mathbf{w}$ can affect $\widehat{\gamma}$ without changing γ.

Optimal Margin Classifier. This algorithm works only in special cases, but will be generalized in a lecture or two with something called kernels.

The goal here is to maximize the geometric margin, in particular solving for γ, $\mathbf{w}$, and b. Specifically, the goal is to capture $\max_{\gamma,\mathbf{w},b} \gamma$ subject to the constraints that $y^{(i)}(\mathbf{w}^{\mathrm{T}}\mathbf{x}^{(i)} + b) \geq \gamma$ for all $i = 1, \ldots, m$, and $||\mathbf{w}|| = 1$. If one treats this as an

optimization problem, it yields an algorithm that would return this optimal margin classifier. However, the constraints are nasty: $||\mathbf{w}|| = 1$ is a nonconvex constraint.

So instead, let's try something different: $\max_{\widehat{\gamma},\mathbf{w},b} \widehat{\gamma}/||\mathbf{w}||$ such that $y^{(i)}(\mathbf{w}^{\mathrm{T}}\mathbf{x}^{(i)} + b) \geq \widehat{\gamma}$ for $i = 1,\ldots,m$. This approach removes the bad constraint, but now the objective $\widehat{\gamma}/||\mathbf{w}||$ is nonconvex, which is again suboptimal.

Finally, after another transformation, one happens upon a much better optimization problem: $\min_{\mathbf{w},b} ||\mathbf{w}||^2$ such that $y^{(i)}(\mathbf{w}^{\mathrm{T}}\mathbf{x}^{(i)} + b) \geq 1$. This is developed because one can scale $\widehat{\gamma}$ such that it is equal to 1. The norm is squared so that the problem is convex.[21] Now, one has a problem that can be solved by software packages, though it only works in the case of separable data: if the data are inseparable, the constraints end up being inconsistent, and the solution set is empty.

Lagrange Duality. Though QP solvers (quadratic programming) can deal with this, there's yet another way to write this problem that allows it to be more useful.

Lagrange duality will today be presented as a set of facts about optimization problems, and only in the next lecture will its use in SVM problems be demonstrated. It is rooted in the method of Lagrange multipliers, which hopefully serves as a useful stepping-stone for the rest of the theory.

Suppose one wants to minimize a function $f(\mathbf{w})$ subject to the constraints $h_i(\mathbf{w}) = 0$ for $i = 1,\ldots,\ell$, the method of Lagrange multipliers instructs one to construct a Lagrangian

$$\mathscr{L}(\mathbf{w},\beta) = f(\mathbf{w}) + \sum_{i=1}^{\ell} \beta_i h_i(\mathbf{w}),$$

where the β_i are called Lagrange multipliers, and then take partial derivatives $\frac{\partial \mathscr{L}}{\partial w_i}$ and $\frac{\partial \mathscr{L}}{\partial \beta_i}$, set them to zero, and solve for $\mathbf{w}$ and β.

To derive the SVM, a variation of this is needed which uses inequality constraints, rather than equality constraints. Thus, consider a minimization problem on $f(\mathbf{w})$,[22] such that $g_i(\mathbf{w}) \leq 0$ for $i = 1,\ldots,k$, and $h_i(\mathbf{w}) = 0$ for $i = 1,\ldots,\ell$. See the lecture notes for this most general version; in the interests of time and ease of understanding, let $\ell = 0$, so that there are no equality constraints.

This aforementioned optimization problem is called the primal optimization problem, in that it's the original problem we wanted to solve. A corresponding problem called the dual problem will also show up. Thus, write down the generalized Lagrangian

$$\mathscr{L}(\mathbf{w},\alpha) = f(\mathbf{w}) + \sum_{i=1}^{k} \alpha_i g_i(\mathbf{w}),$$

or in the more general case where the equality constraints h_i exist,

$$\mathscr{L}(\mathbf{w},\alpha,\beta) = f(\mathbf{w}) + \sum_{i=1}^{k} \alpha_i g_i(\mathbf{w}) + \sum_{i=1}^{\ell} \beta_i h_i(\mathbf{w}).$$

Then, define $\Theta_p(\mathbf{w}) = \max_{\alpha_i \geq 0} \mathscr{L}(\mathbf{w},\alpha)$. (Here, the subscript p stands for "primal.") Then, let

$$p^* = \min_{\mathbf{w}} \Theta_p(\mathbf{w}) = \min_{\mathbf{w}} \left(\max_{\alpha_i \geq 0} \mathscr{L}(\mathbf{w},\alpha) \right).$$

If $\mathbf{w}$ violates the constraints of the primal optimization problem, then $g_i(\mathbf{w}) > 0$ for some i, so $\Theta_p(\mathbf{w}) = \infty$, chosen by setting $\alpha_i = \infty$, maximizing without bound. Conversely, if $\mathbf{w}$ satisfies the constraints, then $g_i(\mathbf{w}) \leq 0$ for all i, so in maximizing Θ_p over all of the α_i, all of the terms are negative, so the solution is $\alpha_i = 0$ for all i. Thus, if $\mathbf{w}$ satisfies the constraints, then $\Theta_p(\mathbf{w}) = f(\mathbf{w})$. What this means is that minimizing $\Theta_p(\mathbf{w})$ over $\mathbf{w}$ is the same as solving the original problem, since it encodes the constraints already.

In order to solve the SVM, one must write out a version of this problem called the dual optimization problem, given by defining $\Theta_D(\alpha) = \min_{\mathbf{w}} \mathscr{L}(\mathbf{w},\alpha)$. Then, let

$$d^* = \max_{\alpha_i \geq 0} \Theta_D(\alpha) = \max_{\alpha_i \geq 0} \left(\min_{\mathbf{w}} \mathscr{L}(\mathbf{w},\alpha) \right).$$

Notice how much this loops like the formula for p^*, but in that the minimum and maximum terms are changed. It is a mathematical fact that the max-min is always less than the min-max, so $d^* \leq p^*$. For example,

$$\max_{y\in\{0,1\}} \left(\min_{x\in\{0,1\}} 1\{x = y\} \right) \leq \min_{x\in\{0,1\}} \left(\max_{y\in\{0,1\}} 1\{x = y\} \right),$$

[21] This is because $f(\mathbf{x}) = ||\mathbf{x}||^2$ is a convex function, which makes the math very nice.

[22] For the optimal margin classifier, we have $f(\mathbf{w},b)$, but leaving b off for now will simplify the notation.

because the left-hand value will be maximizing zero, and the right-hand side minimizes 1. Thus, we have $0 \leq 1$, which ought to make sense.

Under certain conditions, solving the primal and dual problems give the same solution, or $p^* = d^*$. When this happens, one can replace the primal problem with the dual problem, solve the dual for some optimal value α^*, and then translate this back into a solution for the primal problem $\mathbf{w}^*$. In some cases, this is an easier problem to solve.

11. Support Vector Machines II: 10/16/13

Recall that the goal of an optimal margin classifier is to classify data $h_{\mathbf{w},b}(x) = g(\mathbf{w}^T\mathbf{x} + b)$, where g is the sign function. The constraint is that $||\mathbf{w}|| = 1$ and such that $y^{(i)}(\mathbf{w}^T\mathbf{x}^{(i)} + b) \geq 1$. This so far only worked on linearly separable data, but using some mathemagic called Lagrange duality (in essence, invoking Lagrange multipliers to transform the constrained optimization problem). The result was that

$$\Theta_p(W) = \begin{cases} \infty, & \text{if constraints } g_i(\mathbf{w}) \leq 0 \text{ satisfied,} \\ f(w), & \text{otherwise.} \end{cases}$$

The optimal value was $p^* = \min_{\mathbf{w}} \Theta_p(\mathbf{w}) = \min_{\mathbf{w}} \max_{\alpha \geq 0} \mathcal{L}(\mathbf{w}, a)$, which has a dual $d^* = \max_{\alpha \geq 0} \min_{\mathbf{w}} \mathcal{L}(\mathbf{w}, a) = \max_{\alpha \geq 0} \Theta_D(a)$. As it happens, the max-min is always less than the min-max, so $d^* \leq p^*$. But under some situations having the dual optimization problem provides a lot of insight about the primal problem, or even an explicit solution. Specifically, one will:

- Find the conditions under which $d^* = p^*$, which allow this to be useful.
- Derive $\Theta_D(\alpha)$ for the optimal margin classifier.
- Solve for α.
- Map the problem back to the primal domain, yielding the solution originally asked for.

To this end, a theorem is useful:

Theorem 11.1 (Karush-Kuhn-Tucker)**.** *Let f be a convex function and suppose the constraints g_i on f are convex and strictly feasible (i.e. there exists a $\mathbf{w}$ such that for all i, $g_i(\mathbf{w}) < 0$). Then, there exist $\mathbf{w}^*$ and α^* such that:*

(1) $\mathbf{w}^*$ solves the primal problem $\mathbf{w}^* = \arg\min_{\mathbf{w}} \Theta_p(\mathbf{w})$,
(2) α^* solves the dual problem $\alpha^* = \arg\max_{\alpha} \Theta_D(\alpha)$,
(3) $p^* = d^* = \mathcal{L}(\mathbf{w}^*, \alpha^*)$, and
(4) the following four equations hold: $\frac{\partial}{\partial w_i}\mathcal{L}(\mathbf{w}, \alpha) = 0$, $\alpha^* g_i(\mathbf{w}^*) = 0$,[23] $g_i(\mathbf{w}^*) \leq 0$, and $\alpha_i^* \geq 0$.

This theorem is also called the KKT theorem.

Now, how does this apply to the optimal margin classifier? The constraint can be restated as $g_i(\mathbf{w}, b) = -y^{(i)}(\mathbf{w}^T\mathbf{x}^{(i)} + b) + 1 \leq 0$. Thus, for any example $(\mathbf{x}^{(i)}, y^{(i)})$ where $\alpha_i > 0$, then $g_i(\mathbf{w}, b) = 0$. This is an active constraint (i.e. equality, not inequality). This means that the functional margin of $\hat{\gamma}^{(i)}$ at $(\mathbf{x}^{(i)}, y^{(i)})$ is 1. Geometrically, these training examples are the closest to the optimum line of separation, so other points are less important for the optimization problem. These most important examples are called the support vectors.

The key realization is that in an SVM, typically only a small number of $\alpha_i \neq 0$ (in well-behaved two-dimensional data sets, there are no more than three support vectors). This makes optimizing everything relatively efficient. Rather than thinking too hard about the actual applications, it might be more intuitive to just approach the optimization problem and temporarily ignore the fact that it has an application.

Now, we can derive the dual bound: the dual problem to

$$\mathcal{L}(\mathbf{w}, b, \alpha) = \frac{1}{2}||\mathbf{w}||^2 - \sum_{i=1}^{m} \alpha_i (y^{(i)}(\mathbf{w}^T\mathbf{x}^{(i)} + b) - 1)$$

is $\max_{\alpha_i \geq 0} \Theta_0(\alpha)$; $\Theta_D(\alpha) = \min_{\mathbf{w},b} \mathcal{L}(\mathbf{w}, b, a)$. Minimizing these functions is a bit of a mess, but can be found in the lecture notes:

$$\nabla_{\mathbf{w}}(\mathbf{w}, b, \alpha) = \sum_{i=1}^{m} \alpha_i y^{(i)} x^{(i)}$$

is set equal to zero, leading to

$$\min_{\mathbf{w}} \mathcal{L}(\mathbf{w}, b, \alpha) = \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i,j} y^{(i)} y^{(j)} \left\langle x^{(i)}, x^{(j)} \right\rangle - b \sum_i \alpha_i y^{(i)}.$$

[23] This is an example of a complementary condition: it implies that at least one (usually exactly one) of the elements of the product is zero.

Thus, the dual problem is to maximize

$$\Theta_D(\alpha) = \begin{cases} -\infty, & \text{if } \sum_i \alpha_i^{(i)} \neq 0. \\ \sum_i \alpha_i \frac{1}{2} \sum_{i,j} y^{(i))} y^{(j)} \alpha_i \alpha_j \left\langle \mathbf{x}^{(i)}, \mathbf{x}^{(j)} \right\rangle & \text{otherwise.} \end{cases}$$

Confusingly, the "otherwise" part of the above equation is also called W.

The hard part of this is to solve for $\mathbf{w}$ from α^*. $\mathbf{w}$ may live in a high-dimensional (or even infinite-dimensional) space. But this allows the parameters to be easily found, as seen in the lecture notes. This can all be elegantly dealt with using inner products.

To actually use this to classify things, one sets $\mathbf{w} = \sum_{i=1}^m \alpha y^{(i)} x^{(i)}$, so $h_{\mathbf{w},b}(\mathbf{x}) = g(\mathbf{w}^{\mathrm{T}}\mathbf{x} + b) = \sum_i y^{(i)} \left\langle \mathbf{x}^{(i)}, \mathbf{x} \right\rangle + b$. This is kind of nice; as long as it's possible to compute inner products, one doesn't really need to touch the rest of the examples, creating a nice increase in efficiency.

Once $\mathbf{w}$ is known, the decision boundary is some line orthogonal to it, and which line in particular is determined by b. Though $\mathbf{w} \in \mathbb{R}^n$ for a large (or infinite) n, $b \in \mathbb{R}$, which makes it a little easier to obtain. The precise formula will be found in the lecture notes.

Kernels. This algorithm has an interesting property, that it can be written out entirely in terms of inner products of the training examples. That is, the training data is only used inside these inner products. Additionally, the only reason one cares about $\mathbf{w}^*$ is to make predictions. Thus, if one can compute inner products between examples, it's not explicitly necessary to compute everything.

The idea behind kernels is that in circumstances where $\mathbf{x}^{(i)}$ is large-dimensional (or infinite-dimensional), explicitly touching it is bad, but perhaps inner products with it can be computed efficiently. This sounds magical, and indeed only works in a small subset of cases.

Example 11.1. Looking at the usual housing prediction problem, suppose x is the size of a house, and let $\phi(x) = (x, x^2, x^3, x^4)^{\mathrm{T}}$ be some set of nonlinear features added to x, called the feature vector. Then, everywhere in the algorithm where one has $\left\langle \mathbf{x}^{(i)}, \mathbf{x}^{(j)} \right\rangle$, it can be replaced with $\left\langle \phi(x^{(i)}), \phi(x^{(j)}) \right\rangle$. This can be computed relatively efficiently relative to the dimension of ϕ.

The goal is to implement a kernel function $K(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = \left\langle \phi(\mathbf{x}^{(i)}), \mathbf{x}^{(j)} \right\rangle$ that is efficient to compute, even though ϕ is not as easy to compute (since it is a high-dimensional quantity).

Suppose $\mathbf{x}, \mathbf{z} \in \mathbb{R}^n$ and $K(\mathbf{x}, \mathbf{z}) = (\mathbf{x}^{\mathrm{T}}\mathbf{z})^2$. This simplifies to

$$K(\mathbf{x}, \mathbf{z}) = \left(\sum_{i=1}^n x_i z_i \right)^2 = \sum_{i,j=1}^n x_i x_j z_i z_j.$$

In other words, $K(\mathbf{x}, \mathbf{z}) = \phi(\mathbf{x})^{\mathrm{T}} \phi(\mathbf{z})$, where $\phi(\mathbf{x}) = \sum_{i,j} x_i x_j$. Notice that $\phi(\mathbf{x}) \in \mathbb{R}^{n^2}$, so it takes $O(n^2)$ time to compute $\phi(\mathbf{x})$. However, it takes $O(n)$ time to compute $K(\mathbf{x}, \mathbf{z})$, because each inner product is $O(n)$, and then they're just multiplied and squared.

Not all terms in $\phi(\mathbf{x})$ must be quadratic in the entries of $\mathbf{x}$; some could be linear terms (e.g. multiplication by a constant). If one has $K(\mathbf{x}, \mathbf{z}) = (\mathbf{x}^{\mathrm{T}}\mathbf{z} + c)^d$, then this corresponds to $\phi(\mathbf{x})$ containing all monomials in the entries of $\mathbf{x}$ up to degree d. For example, it might have entries $x_1 x_{23} x_{14}$, $x_2^3 x_5$, etc. This means that $\phi(\mathbf{x})$ has $\binom{n+d}{d}$ features, which is $O(n^d)$ features! Thus, computing $\phi(\mathbf{x})$ is $O(n^d)$, but $K(\mathbf{x}, \mathbf{z})$ is still $O(n)$ time. Crucial to using these kernels was the fact that once the dual problem was written out, everything was in terms of inner products, meaning that kernels could be used.

When one asks if a function is a valid kernel, the question is equivalent to: is there a ϕ such that $K(\mathbf{x}, \mathbf{z}) = \phi(\mathbf{x})^{\mathrm{T}} \phi(\mathbf{z})$? For example, $K(\mathbf{x}, \mathbf{z}) = -1$ won't work. Semantically, an invalid kernel might still be called a kernel, in which case one distinguishes valid kernels from invalid ones.

The Gaussian kernel is one of the most widely used kernels in applications, and is a good one to try if you don't have any other good ideas. It looks like

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

It is given by a feature vector $\phi(\mathbf{x})$, but it turns out to be infinite-dimensional.

This allows one to implement nonlinear classifiers; for example, if $y = 0$ lies entirely within the range where $y = 1$, it might be possible to classify linearly in a very high-dimensional space, which projects down to a nonlinear separator in lower-dimensional space.

Thus, a typical path is to take some dataset, use a ϕ to map it into a higher-dimensional or infinite-dimensional feature space, and then project the classifier back down into $\mathbb{R}^n$, where it is a useful, nonlinear decision boundary.

12. Convex Optimization I: 10/18/13

Convex optimization is used in many places inside and outside of machine learning, in general wherever an optimization problem exists. These notes are based on Stephen Boyd's.

A large amount of the work in convex optimization is formulating the problem as a convex problem. Once this is done (typically by algebra stuff, such as juggling formulas around and performing variable substitutions), commercial solvers exist to assist.

Definition. A set C is convex if for any two points $x, y \in C$, any convex combination of them (i.e. an element of the form $\theta x + (1-\theta)y$, for $0 \le \theta \le 1$) is also within the set.

This means that for any two points within a set, the line between them is contained within the set. Thus, a pentagon is a convex set, but a jellybean is not. Other useful examples include $\mathbb{R}^n$ and $\mathbb{R}^n_+$ (i.e. the set of vectors with positive entries). These can both be proven, which is mostly a matter of checking the definition.

Balls under some norm also induce convex sets: for example, the set $\{\mathbf{x} \in \mathbb{R}^n \mid ||\mathbf{x}||_2 \le 1\}$ (i.e. the L^2-norm, where the balls are n-spheres; you can also use the L^1-norm, in which they are diamonds, and the L^∞ norm, where they are squares).

Another set of examples is that of affine spaces, which are sets of the form $\{\mathbf{x} \in \mathbb{R}^n \mid A\mathbf{x} = \mathbf{b}\}$, for some matrix $A \in \mathbb{R}^{n \times m}$. This is because $A(\theta\mathbf{x} + (1-\theta)\mathbf{y}) = \theta A\mathbf{x} + (1-\theta)A\mathbf{y} = \theta + (1-\theta)\mathbf{b} = \mathbf{b}$.

The intersection of two convex sets is convex; but in general, the union is not convex.

The set of positive semidefinite matrices S^n_+ is also convex (here, $S^n \subset \mathbb{R}^{n \times n}$ is the set of $n \times n$ symmetric matrices): if $\mathbf{x}^T A\mathbf{x} \ge 0$ and $\mathbf{x}^T B\mathbf{x} = 0$, then

$$\mathbf{x}^T(\theta A + (1-\theta)B)\mathbf{x} = \theta\mathbf{x}^T A\mathbf{x} + (1-\theta)\mathbf{x}^T B\mathbf{x} \ge 0,$$

since each of its components $\mathbf{x}^T A\mathbf{x}$, $\mathbf{x}^T B\mathbf{x}$, θ, and $1-\theta$ are all positive.

Definition. A convex function on a set S is a function $f : S \to \mathbb{R}$ such that for any $x, y \in S$, $f(\theta\mathbf{x} + (1-\theta)\mathbf{y}) \le \theta f(\mathbf{x}) + (1-\theta)f(\mathbf{y})$. f is called strictly convex if the inequality is strict.

If the inequality goes in the other direction, the function is called concave.

The intuition is that if you take the graph of a function, the secant line between any two points lies above the graph.

If f is differentiable (i.e. $\nabla_{\mathbf{x}} f(\mathbf{x})$ exists), then there's an equivalent characterization: that $f(\mathbf{y}) \ge f(\mathbf{x}) + \nabla_{\mathbf{x}} f(\mathbf{x})(\mathbf{y} - \mathbf{x})$ for any $\mathbf{x}, \mathbf{y} \in S$. This is saying that the tangent line lies below the graph, which is equivalent.

This is one of the reasons convex optimization is so powerful; with a derivative and a single point, we have a global underestimator of the function.

If the function is twice-differentiable, so that $\nabla^2_{\mathbf{x}} f(\mathbf{x})$ exists, then there's another equivalent condition: that the Hessian is positive semidefinite, written $\nabla^2_{\mathbf{x}} f(\mathbf{x}) \succeq 0$.[24]

The definition of a convex set can be extended to multiple points, leading to a result called Jensen's inequality: if n points are averaged, their result should still lie within the function. The result is

$$f\left(\sum_{i=1}^k \theta_i \mathbf{x}_i\right) \le \sum_{i=1}^k \theta_i f(\mathbf{x}_i),$$

as long as $\sum_{i=1}^k \theta_i = 1$ and $\theta_i \ge 0$ for all i. In other words, this works as long as there is a discrete probability distribution using the θ_i!

Thus, it seems reasonable to generalize to a continuous distribution: if $p(\mathbf{x})$ is a probability distribution on S, then Jensen's inequality states that

$$f\left(\int p(x)x\,\mathrm{d}x\right) \le \int p(x)f(x)\,\mathrm{d}x.$$

Finally, there's a statement in terms of expected value:

$$f(\mathbb{E}[x]) \le \mathbb{E}[f(x)].$$

These are all equivalent statements with the same intuition (the secant line lying above the graph).

Definition. If $f : D \to \mathbb{R}$ is a convex function, an α-sublevel set is the set $\{\mathbf{x} \in D \mid f(\mathbf{x}) \le \alpha\}$.

An α-sublevel set given by a function is also convex.

Example 12.1. Examples of convex functions:

- The exponential function $f(x) = e^{ax}$, because $f''(x) = x^2 e^{ax} \ge 0$.

[24] The notion $\succeq$ has two different meanings: for vectors $\mathbf{x}, \mathbf{y}$, $\mathbf{x} \succeq \mathbf{y}$ means that $x_i \ge y_i$ for each component, and for matrices it means positive (semi)definiteness. Thus, these are not total orders, as there exist plenty of $\mathbf{x}, \mathbf{y}$ such that $\mathbf{x} \not\succeq \mathbf{y}$ and $\mathbf{x} \not\preceq \mathbf{y}$.

- The negative logarithm $f(x) = -\log x$, because $f''(x) = 1/x^2 > 0$.
- Affine functions $f(\mathbf{x}) = A\mathbf{x} + \mathbf{b}$ are both convex and concave, because $\nabla_{\mathbf{x}}^2 f(\mathbf{x}) = 0$.
- All norms are convex functions, which follows from the triangle inequality.
- Nonnegative weighted sums of convex functions are still convex. Similarly, the maximum of a set of convex functions is still convex.

Convex Optimization Problems. Once a problem is placed in this form, off-the-shelf solvers will be able to tackle it. The specific form is to minimize $f(x)$ subject to $x \in C$, where f is a convex function and C a convex set. This is typically written as $g_i(x) \leq 0$ for some $i = 1, \dots, m$, and $h_i(x) = 0$ for $i = 1, \dots, p$. Here, the g_i are required to be convex, and the h_i are required to be *affine*. This can be intuited by looking at the 0-sublevel set.

Since f is convex, there is a unique global minimum (though there many be more than one argmin). This is generally written $p^* = \min\{f(x) \mid g_i(x) \leq 0, h_i(x) = 0\}$. Sometimes, f is unbounded below, which is written $p^* = -\infty$. Correspondingly, $p^* = \infty$ means that the constraints are infeasible.

For an example of a convex optimization problem, one has an affine function $C\mathbf{x} + \mathbf{b}$ subject to affine constraints $G\mathbf{x} \preceq \mathbf{h}$ and $A\mathbf{x} = \mathbf{b}$. These end up constraining $\mathbf{x}$ to lie within some polyhedron, and the minimum will lie at some vertex.

A generalization of this is quadratic programming, in which the goal is to minimize a quadratic function $\mathbf{x}^{\mathrm{T}} P\mathbf{x} + \mathbf{c}^{\mathrm{T}}\mathbf{x} + \mathbf{d}$ subject to the same affine constraints. Now., however, there's no guarantee that the minimum lies at one of the vertices of the polyhedron.

Another generalization is that of quadratically constrained quadratic programs. Here, the constraints are replaced with quadratic ons of the form $(1/2)\mathbf{x}^{\mathrm{T}} Q_i \mathbf{x} + \mathbf{r}_i^{\mathrm{T}}\mathbf{x} + \mathbf{s}_i = 0$ for $i = 1, \dots, m$.

Finally, there's yet another generalization called semidefinite programming, which is covered in the lecture notes (or in the convex optimization class).

Using a library called CVX (which is free for non-commercial use), one can solve convex problems in MATLAB. For example, suppose one wants to minimize $(1/2)\|\mathbf{w}\|_2^2 + C\sum \xi_i$ such that $y^{(i)}(\mathbf{w}^{\mathrm{T}}\mathbf{x}^{(i)} + b) \geq 1 - \xi_i$ for all i, and $\xi_i \geq 0$. This sort of problem is useful for SVMs. Then, the syntax is as easy as

```
cvx_begin
    variables w(n) b xi(m)
    minimize 1/2*sum(w.*w) + C*sum(xi)
    y.*(X*w + b) >= 1 - xi;
    xi >= 0;
cvx_end
```

This fills `w` with the values that are in the optimum solution, and prints a lot of information (e.g. how many iterations were necessary) to the console.

13. SUPPORT VECTOR MACHINES III: 10/21/13

Mercer Kernels. To review, we want to maximize the function

$$W(\alpha) = \sum_{i=1}^{n} \alpha_i - \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m} y^{(i)} y^{(j)} \alpha_i \alpha_j \langle \mathbf{x}^{(i)}, \mathbf{x}^{(j)} \rangle,$$

such that $\sum_i y^{(i)}\alpha_i = 0$ and $\alpha_i \geq 0$. By invoking the KKT theorem, one can relate this dual optimization problem to the primal problem, implying that

$$h_{\mathbf{w},b}(\mathbf{x}) = g\left(\sum_{i=1}^{m} \alpha_i y^{(i)} \mathbf{x}^{(i)\mathrm{T}}\mathbf{x} + b\right).$$

Defining $K(\mathbf{x}^{(i)}, \mathbf{x}) = \mathbf{x}^{(i)\mathrm{T}}\mathbf{x}$, this can be thought of as sending $\mathbf{x} \mapsto \phi(\mathbf{x})$, so that $\phi(\mathbf{x})^{\mathrm{T}}\phi(\mathbf{z}) = K(\mathbf{x}, \mathbf{z})$. K is a valid kernel iff such a ϕ exists.

One ought to think of the kernel function as a way of measuring similarity, though not all similarity functions are kernels: if there is an $\mathbf{x}$ such that $K(\mathbf{x}, \mathbf{x}) = 1$, no ϕ exists that satisfies the above equations, so that K is not a kernel.

We also have the kernel matrix $\mathcal{K}$ given by $\mathcal{K}_{ij} = K(\mathbf{x}^{(i)}, \mathbf{x}^{(j)})$. This matrix is positive semidefinite.

Theorem 13.1 (Mercer). *A function K is a valid kernel iff for any $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(d)}\}$, the corresponding kernel matrix $\mathcal{K} \in \mathbb{R}^{d \times d}$ is symmetric and positive semidefinite.*

A valid kernel is also known as a Mercer kernel. The forward direction was done before, and symmetry follows from expanding out $K(\mathbf{x}, \mathbf{z}) = \phi(\mathbf{x})^{\mathrm{T}}\phi(\mathbf{z}) = \phi(\mathbf{z})^{\mathrm{T}}\phi(\mathbf{x}) = K(\mathbf{z}, \mathbf{x})$. This is a useful test for valid kernels, but there are other tests for this.

One other kernel of note is the linear kernel $K(\mathbf{x},\mathbf{z}) = \mathbf{x}^{\mathrm{T}}\mathbf{z}$, given by $\phi(\mathbf{x}) = \mathbf{x}$. The resulting algorithm is known as the optimal margin classifier. The terminology "SVM with a linear kernel" is more common nowadays than "optimal margin classifier," and these algorithms have uses in text processing and the like.

Kernels are more general than the SVM idea, because there are other algorithms that can be written in an inner product. Then, $\left\langle \mathbf{x}^{(i)}, \mathbf{x}^{(j)} \right\rangle$ can be replaced with $K(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = \phi(\mathbf{x}^{(i)})^{\mathrm{T}}\phi(\mathbf{x}^{(j)})$. This is known as the "kernel trick," or "kernelization." Another useful kernel is the Gaussian kernel, given by

$$K(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = \exp\left(-\frac{||\mathbf{x}^{(i)} - \mathbf{x}^{(j)}||^2}{2\sigma^2}\right).$$

In this case, $\phi(\mathbf{x}) \in \mathbb{R}^{\infty}$. For intuition, think of taking the Taylor expansion, which will lead a polynomial into a kind of infinite-dimensional space. If this seems really cool, try this challenging exercise:

Exercise 13.1. Prove that the Gaussian kernel is a valid kernel.

L_1-norm soft margin. Thus far, we have had to assume that the data is linearly separable, so that the derived algorithms don't explode. Sometimes, this can be dodged (e.g. in infinite-dimensional space given by the Gaussian kernel), but it's not that nice and sometimes one likes to use the linear kernel anyways. For example, the presence of a single outlier shouldn't interfere much in classification (e.g. it can greatly change the line drawn by the optimal margin classifier).

The L_1-norm soft margin version of the SVM elegantly handles both of these issues. The primal problem is to minimize

$$\frac{1}{2}||\mathbf{w}||^2 + C\sum_{i=1}^{m}\xi_i$$

subject to the constraints

$$y^{(i)}(\mathbf{w}^{\mathrm{T}}\mathbf{w}^{(i)} + b) \geq 1 - \xi_i$$

and $\xi_i \geq 0$ for all i. (The red bits of the equations are the differences from the classical SVM primal problem). In some sense, we don't need the functional margin to be exactly 1, but maybe something slightly less than that.

Now, to find the dual problem, write down the Lagrangian. There are extra Lagrange multipliers $\mathbf{r}$ corresponding to the constraint that ξ has nonnegative entries.

$$\mathscr{L}(\mathbf{w}, b, \xi, \alpha, \mathbf{r}) = \frac{1}{2}||\mathbf{w}||^2 + C\sum_{i=1}^{m}\xi_i - \sum_{i=1}^{m}\alpha_i\left(\mathbf{y}^{(i)}(\mathbf{w}^{\mathrm{T}}\mathbf{x} + b) - 1 + \xi_i\right) - \sum_{i=1}^{m} r_i\xi_i.$$

Here, the last term $\mathbf{r}\cdot\xi$ roughly corresponds to the term $\alpha_i g_i(\mathbf{w}, b)$ such that $g_i(\mathbf{w}, b) \geq 0$; it is the constraint term. This makes the dual problem to maximize

$$W(\alpha) = \sum_{i=1}^{m}\alpha_i - \sum_{i,j} y^{(i)}y^{(j)}\alpha_i\alpha_j\langle \mathbf{x}^{(i)}, \mathbf{x}^{(j)}\rangle, \tag{7}$$

such that $\sum_{i=1}^{n} y^{(i)}\alpha_i = 0$ and $0 \leq \alpha_i \leq C$. Notice that $\mathbf{r}$ vanishes, which makes this very close to the original problem.

Exercise 13.2. Prove the result in (7); this will really develop your skills at understanding the math behind SVMs.

The constraints are a little more interesting than one might expect: the first one is a linear constraint, which is unpleasant to handle, and the second is a box constraint (α is constrained in an n-dimensional box). Finally, it is significant that (7) is a big quadratic function in terms of the α_i, which is why it is necessary for the kernels $K(\mathbf{x}^{(i)}, \mathbf{x}^{(j)})$ to be valid inner products, so that the optimization still has a well-defined solution.

Coordinate Ascent. Suppose one wants to maximize a function $W(\alpha_1, \ldots, \alpha_m)$ with (for now) no constraints on α. Coordinate ascent is an algorithm that repeatedly does the following:

- Repeat until some convergence criterion is met:
  - For each i in $1, \ldots, m$:
    - Maximize W with respect to α_i, holding $\alpha_1, \ldots, \alpha_{i-1}, \alpha_{i+1}, \ldots, \alpha_m$ fixed.

This algorithm runs through each variable, optimizing it while ignoring the others, first α_1, then α_2, and so on, until it repeats. It generally takes far more iterations to converge than gradient ascent or Newton's method, but the inner loop can be very fast in some contexts, which makes it preferred in some situations (e.g. when W is quadratic, so that an analytic value for the maximum is known).

The issue with applying this to the L_1-norm soft margin is that the first constraint isn't preserved by this algorithm.

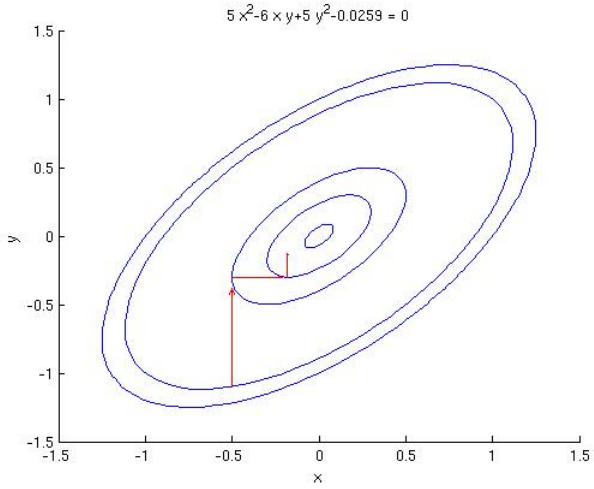


FIGURE 4. Illustration of coordinate ascent. Source: Wikipedia

SMO algorithm. SMO stands for sequential minimal optimization. In this algorithm, one sequentially minimizes the smallest possible number of variables such that the constraints are met, which turns out to be 2.

The algorithm does the following:

- Select α_i and α_j to optimize with respect to. This can be done by heuristic, but randomly choosing them also works.
- Optimize with respect to α_i and α_j, holding everything else constant. Without loss of generality, let $i = 1$ and $j = 2$ to simplify the writing. Then, by an inductive argument, we can show that $\sum_i \alpha_i y^{(i)} = 0$, so that $\alpha_1 y^{(1)} + \alpha_2 y^{(2)} = -\sum_{i=3}^{m}, \alpha + i y^{(i)}$. But since the term on the right is constant, call it ζ.

Now, since α_1 and α_2 are the only parameters, one can imagine the problem on a plane. The constraint $\alpha_1 y^{(1)} + \alpha_2 y^{(2)} = \zeta$ is a line through this plane. There are also the box constraints: $0 \le \alpha_1, \alpha_2 \le C$. Thus, the space of possible solutions is a line segment.

This step can be done very efficiently, which is the power of the SMO, and here's why: the problem can be rewritten as

$$\alpha_1 = \frac{\zeta - \alpha_2 y^{(2)}}{y^{(i)}} \cdot (y^{(1)})^2 = y^{(1)}(\zeta - \alpha_1 y^{(1)}),$$

because $y^{(1)} = \pm 1$, so $(y^{(1)})^2 = 1$. Then, plugging this into W,

$$W(\alpha_1, \alpha_2, \alpha_3, \dots, \alpha_n) = W(y^{(1)}(\zeta - \alpha_1 y^{(1)}), \alpha_2, \alpha_3, \dots, \alpha_m).$$

But since the α_i are constants when $i \ge 3$, then this is just a large quadratic function in terms of α_2, so it can be optimized very quickly by taking the derivative, or doing only two or three floating point operations. Then, using the linear constraint, one finds a value of α_1. Sometimes, this value will be outside of the box constraint, so one has to choose the end of the segment closest to the found value.

One interesting aspect of the SVM is that since only a few of the training examples are support vectors, then most of the training examples have Lagrange multipliers $\alpha_i = 0$, so initializing $\alpha = 0$ at the start of the SMO algorithm is a particularly good idea: most of them won't change.

In general, the SVM involves a lot of optimization. These algorithms are at the point where there's no need to reimplement any of it; instead, one ought to find the appropriate library and use it.

Examples of SVMs and Kernels. Suppose one is doing image processing and has an image divided into pixels. One way to construct a feature vector for such an $n \times n$ image is to list out all of the values of the pixels, giving a vector in $\mathbb{R}^{n^2}$. The Gaussian kernel works well for this problem, as well as $K(\mathbf{x}, \mathbf{z}) = (\mathbf{x}^T \mathbf{z})^d$ will give a reasonably good handwriting recognition algorithm.[25]

Another example is protein classification. A protein can be modeled as a string over an alphabet of amino acids, of which there are roughly 20 (so they're denoted with English letters). Since proteins may have different lengths, the choice for the feature vector is not immediately apparent, though it resembles the text classification problem. Thus, the feature vector $\phi(X)$ is the list of number of occurrences of all four-letter sequences of amino acids (first, the number of occurrences of $AAAA$,

[25]For classification in which y might take on more than two values, one good strategy is to train a binary classifier for each value versus everything else, giving $n - 1$ classifiers. This is known as one-versus-all classification.

then *AAAB*, and so on, to *ZZZZ*). This feature vector has dimension 20^4, which is a lot. However, there exists an efficient algorithm for computing inner products of this feature, since most of the entries are zero, and the Morris-Pratt dynamic programming algorithm is useful for handling this.

Part 2. Learning Theory

14. Learning Theory: 10/23/13

The second of four major topics we will cover in this class is that of learning theory. This is not deep, beautiful math as much as it is a deeper understanding of the material, which is very important in understanding why algorithms work or don't, and so on.

Consider a set of five data points in $\mathbb{R}^2$. One could fit a line to it, which doesn't completely accurately capture the training set. Using a fourth-degree function will fit the data perfectly, but at the expense of making the model much less appropriate for nw data samples. Somewhere in between, a second-degree or third-degree equation might be the most appropriate. These are called, respectively, underfitting (the model fails to capture important aspects of the data); overfitting (too much attention paid to the idiosyncrasies of this particular dataset), and well-fitting.

The underfit model has a problem known as bias, which is almost a mathematical idea. Dually, the overfit model has too much variance. These will be explained more precisely when uniform convergence is discussed. If an algorithm is working poorly, it's useful to know whether the hypothesis has a high bias or high variance, to determine whether (for example) more or fewer features, respectively, are needed. There are also plenty of other ways to handle these problems.

Classification problems can also suffer from high bias or high variance: logistic regression can create some surprisingly complicated separating curves, which can lead to overfit datasets, and correspondingly insufficiently complicated curves could have a high bias.

An important step in the theory is to formulate an abstract model for machine learning. This is something that isn't completely descriptive, but will be very helpful for understanding things and proving theorems nonetheless. Additionally, though the theory works equally well for classification and regression, it's easier to formulate for classification problems, which is what will be followed below.

The goal is to classify $y \in \{0,1\}$ using a hypothesis function $h_\theta \in \{0,1\}$. Then, the training set is $S = \{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^m$. For example, $h_\theta(\mathbf{x}) = g(\theta^{\mathrm{T}}\mathbf{x})$ for g as defined before, etc. Assume that the samples in S are independently and identically distributed.

Definition. The training error of the above is

$$\widehat{\varepsilon}_S(h_\theta) = \widehat{\varepsilon}(h_\theta) = \frac{1}{m}\sum_{i=1}^{m} 1\{h_\theta(\mathbf{x}^{(i)}) \neq y^{(i)}\}.$$

Since ML people aren't good at naming things, this is occasionally also known as risk.

In other words, how often does the hypothesis misclassify the sample?

Definition. The generalization error is $\varepsilon(h_\theta) = P_{(\mathbf{x},y)\sim\mathcal{D}}(h_\theta(\mathbf{x}) \neq y)$. Here, $\mathcal{D}$ is some hypothetical distribution of all possible input data.

The point of generalization error is for the algorithm to generalize well to datasets it hasn't seen before. The goal is thus to minimize this error — yet there is no way to measure it directly. Thus, training error is used as an approximation. The approximation under mild conditions gets better and better as m increases.

One can think of supervised learning as finding the hypothesis with the smallest training error: $\widehat{\theta} = \arg\min_\theta \widehat{\varepsilon}(h_\theta)$. This is known as empirical risk minimization, or ERM. This isn't implemented as a direct algorithm, as it ends up being an NP-hard problem in many cases, so supervised learning algorithms should be thought of convex relaxations of this problem.

But this is still too low-level, so abstract away the θ. Let $\mathcal{H} = \{h_\theta : \theta \in \mathbb{R}^{n+1}\}$, which is the class of all hypothesis functions. In the specific example of logistic regression, $\mathcal{H}$ is the set of all straight lines (which correspond to decision boundaries). Then, the ERM is restated as $\widehat{h} = \arg\min_h \widehat{\varepsilon}(h)$.[26] $\mathcal{H}$ is the set of functions (that we care about) mapping the $\mathbf{x}$ to $\{0,1\}$.

This now allows us to tease out the difference between high bias and high variance: in the case of high bias, $\widehat{\varepsilon}(\widehat{h})$ is high, and in the case of high variance, $\widehat{\varepsilon}(\widehat{h})$ is low (but in both cases, $\varepsilon(\widehat{h})$ is high, illustrating that an underfit model is a poor predictor and an overfit model generalizes badly).

[26] There are many variables wearing hats today; this notation is generally used to denote an approximation to the hatless quantity, as with $\widehat{\varepsilon}$ and ε.

Lemma 14.1 (Union Bound). *Let $A_1, \dots, A_k$ be k events (not necessarily independent). Then,*

$$P\left(\bigcup_{i=1}^{k} A_i\right) \leq \sum_{i=1}^{k} P(A_k).$$

This makes the most sense when one thinks about Venn diagrams: since the A_i could intersect, their total area is at most the sum of their intersections (which would imply there is no intersection). It turns out this is an axiom of probability, called the σ-subadditivity of probability.

Lemma 14.2 (Hoeffding's inequality). *Let $Z_1, \dots, Z_n \sim \text{Bernoulli}(\phi)$ be independently and identically distributed random variables and $\widehat{\phi} = 1/m \sum_{i=1}^{m} Z_i$. Then, for any γ,*

$$P(|\widehat{\phi} - \phi| > \gamma) \leq 2e^{-2\gamma^2 m}.$$

This lemma is also sometimes known as the Chernoff bound. From the Central Limit theorem, $\widehat{\phi}$ has a roughly Gaussian distribution, and this inequality looks at a specific window of the distribution as m increases, so that the variance of $\widehat{\phi}$ shrinks.

Suppose $\mathcal{H} = \{h_1, \dots, h_k\}$ is some finite set (note that this isn't logistic regression anymore). This will make some analysis easier, and can be generalized to infinite $\mathcal{H}$ later. In the finite case, a bound on $\varepsilon(\widehat{h})$ by showing that $\widehat{\varepsilon}(\widehat{h}) \approx \varepsilon(\widehat{h})$ for all h. Let $z_j = 1\{h_j(\mathbf{x}^{(i)}) \neq y^{(i)}\}$, the indicator for whether h_i is correct, and $z_i \in \{0, 1\}$. Thus, $P(z_i = 1) = \varepsilon(h_j)$. When Hoeffding's inequality is applied, this will take the form of ϕ, and $\widehat{\phi}$ is

$$\widehat{\varepsilon}(h_j) = \frac{1}{m}\sum_{i=1}^{m} z_i = \frac{1}{m}\sum_{i=1}^{m} 1\{h_j(\mathbf{x}^{(i)}) - y^{(i)}\},$$

so by Lemma 14.2,

$$P(|\widehat{\varepsilon}(h_j) - \varepsilon(h_j)| > \gamma) \leq 2\exp(-2\gamma^2 m).$$

Let A_j be the event that $|\widehat{\varepsilon}(h_j) - \varepsilon(h_j)| > \gamma$. Then, using the union bound,

$$P(\exists h_j \in \mathcal{H} \text{ s.t. } P(|\widehat{\varepsilon}(h_j) - \varepsilon(h_j)| > \gamma)) = P\left(\bigcup_{i=1}^{k} A_i\right) \leq \sum_{i=1}^{k} P(A_j) = \sum_{i=1}^{k} 2e^{-2\gamma^2 m} = 2ke^{-2\gamma^2 m}.$$

Thus, there is a very good probability that there is no h_j where the difference between the training error and the generalization error is large: formally, $P(\text{there is no } h_j \text{ where } |\widehat{\varepsilon}(h_j) - \varepsilon(h_j)| > \gamma) \geq 1 - 2ke^{-\gamma^2 m}$. This result is known as a uniform convergence bound because for it applies to every hypothesis.

Let $\delta = 2ke^{-2\gamma^2 m}$, so that the probability becomes $1 - \delta$. Then, there are three parameters floating around this result: γ, the error; m, the size of the training set; and δ, the probability. It is possible to solve for one in terms of the other two, which is useful for understanding the uniform convergence. This is explained in greater detail in the lecture notes, but for example if $m \geq (1/2\gamma^2)\log(2k/\delta)$, then, with probability $1 - \delta$, $|\varepsilon(h) - \widehat{\varepsilon}(h)| \leq \gamma$ for all $h \in \mathcal{H}$. This is known as a complexity bound. The final form, solving for γ, gives an error bound: for a fixed m and δ, then with probability at least $1 - \delta$,

$$|\widehat{\varepsilon}(h) - \varepsilon(h)| \leq \underbrace{\sqrt{\frac{1}{2m}\log\frac{2k}{\delta}}}_{\gamma}.$$

for all $h \in \mathcal{H}$.

Let $h^* = \arg\min_{h \in \mathcal{H}} \varepsilon(h)$. This is impossible to definitively determine, but it is approximated by $\widehat{h}$, the hypothesis with the lowest training error. Thus, $\widehat{\varepsilon}(\widehat{h}) \leq \widehat{\varepsilon}(h)$ for any other h, so

$$\begin{aligned}
\varepsilon(\widehat{h}) &\leq \widehat{\varepsilon}(\widehat{h}) + \gamma \\
&\leq \widehat{\varepsilon}(h^*) + \gamma \\
&\leq \varepsilon(h^*) + \gamma + \gamma = \varepsilon(h^*) + 2\gamma.
\end{aligned}$$

This also uses the fact that the training and generalization errors are within γ of each other for al hypotheses. Thus, the bound on the generalization error difference is 2γ.

The above discussion can be compacted into a theorem.

Theorem 14.3. *Let $|\mathscr{H}| = k$, and fix $m \in \mathbb{N}$ and $\delta \in (0,1)$. Then, with probability $1-\delta$,*

$$\varepsilon(\widehat{h}) \leq \underbrace{\min_{h\in\mathscr{H}} \varepsilon(h)}_{\varepsilon(h^*)} + 2\underbrace{\sqrt{\frac{1}{2m}\log\left(\frac{2k}{\delta}\right)}}_{\gamma}.$$

This leads to an understanding of bias versus variance: if the hypothesis class is too narrow, then the best (i.e. minimal) $\varepsilon(h)$ isn't too good, so the first term in on the right-hand side is the bias term. The second term is a variance term: there are so many hypotheses, it's not clear how to choose between them.

VC Dimension. One shortcoming of this approach is that it requires $\mathscr{H}$ to be a finite set. There is a notion called VC dimension can be used to handle this.

Definition. Suppose $\mathscr{H}$ is some infinite hypothesis class, and d points be given. If $\mathscr{H}$ can realize any of the 2^d labelings on the set S of these d points, then $\mathscr{H}$ is said to shatter S.

In other words, for any possible set of labelings, $\mathscr{H}$ can achieve zero training error.

Notice that if $\mathscr{H}$ is the set of linear classifiers on $\mathbb{R}^2$, then it can shatter some sets of 3 points, but no set of four or more points. Intuitively, this says something about its power.

Definition. The VC dimension of a hypothesis class $\mathscr{H}$ is the size of the largest set it can shatter.

As shown, the set of linear decision boundaries, as noted above, can shatter some sets of three points (but, significantly, not all!) but no sets of four points, so $\mathrm{VC}(\mathscr{H}) = 3$ in this case. (Logistic regression does better in higher dimensions.) This can be used as an alternative way of measuring the complexity of the hypothesis class rather than just the number of hypotheses; a higher VC dimension is more likely to lead to overfitting, and a lower VC dimension correlates to bias.

15. Convex Optimization II: 10/25/13

To review, a convex optimization problem is to minimize a convex function $f(x)$ subject to the constraitn that $\mathbf{x}$ lies within some convex set C. Typically, C is presented in terms of some inequalities and affine functions: $g_i(\mathbf{x}) \leq 0$ for $i = 1,\dots,m$, and $h_i(\mathbf{x}) = 0$ for $i = 1,\dots,p$.

One of the advantages of using a library such as CVX is that one can just as easily obtain the dual solution (and therefore a bound for the primal problem) from the problem.

Though one might have seen the Lagrangian before, it's a little more interesting with the inequality constraints (the g_i). The Lagrangian itself is

$$\mathscr{L}(\mathbf{x},\alpha,\beta) = f(\mathbf{x}) + \sum_{i=1}^{m} \alpha_i g_i(\mathbf{x}) + \sum_{i=1}^{p} \beta_i h_i(\mathbf{x}).$$

These α_i and β_i can be thought of as the costs for violating the constraints: a high value of α_i means that the constraint is particularly significant.

The primal problem then becomes

$$\min_{\mathbf{x}} \left(\max_{\alpha,\beta:\alpha\succeq 0} \mathscr{L}(\mathbf{x},\alpha,\beta) \right) = \min_{\mathbf{x}} \Theta_p(\mathbf{x}). \tag{8}$$

If these constraints are realizable, the problem is called primal feasible, and the solution is denoted $p^* = \Theta_p(\mathbf{x}^*)$. The dual problem is the same, but with the maxima and minima switched:

$$\max_{\alpha,\beta:\alpha\succeq 0} \left(\min_{\mathbf{x}} \mathscr{L}(\mathbf{x},\alpha,\beta) \right) = \max_{\alpha,\beta:\alpha_i\geq 0} \Theta_D(\alpha,\beta).$$

If α and β admit such a solution, then the problem is called dual-feasible, and the solution is written $d^* = \Theta_D(\alpha^*,\beta^*)$. Notice how this doesn't depend on $\mathbf{x}$.

If you stare at the constraints for a long enough time in the right way, it becomes clear that if the constraints can't be met, it's possible to make everything arbitrarily large in (8). Thus, one gets the fomula

$$\Theta_p(\mathbf{x}) = \begin{cases} 0, & \mathbf{x} \text{ is feasible,} \\ \infty, & \text{otherwise.} \end{cases}$$

This is the same problem as before, but with the contraints embedded in, in some sense.

The dual is actually a concave function in α and β, while the primal was still convex; this is because it involves taking the minimum of affine functions. But $\Theta_D(\alpha,\beta) \le p^*$ (i.e. it's always an underestimator of the primal solution), because

$$\begin{aligned}\Theta_D(\alpha,\beta) &= \min_{\mathbf{x}} \mathcal{L}(\mathbf{x},\alpha,\beta)\\ &\le \mathcal{L}(\mathbf{x}^*,\alpha,\beta)\\ &= f(\mathbf{x}^*) + \sum \alpha_i g_i(\mathbf{x}^*) + \sum \beta_i h_i(\mathbf{x}^*)\\ &\le f(\mathbf{x}^*) = p^*\end{aligned}$$

because $\mathbf{x}^*$ is feasible, so all of the g_i are nonpositive and the h_i are zero. This is a useful check: if the solution to the dual problem is too large, then something's gone wrong. One can think of maximizing Θ_D as trying to turn the inequalities in the above derivation into equalties, tightening the bound.

The notion of weak duality is that $d^* \le p^*$, which is still true whether or not f is convex (because the dual problem is still concave). Strong duality is when $d^* = p^*$, which holds when f is convex and a condition called Slater's condition holds. This requires that there exists some feasible point $\mathbf{x}$ such that $g_i(\mathbf{x}) < 0$ for all i (strict inequality, which is the important detail). In practice, this is always possible by jiggling the $\mathbf{x}$ around.

Complimentary Slackness. Assuming strong duality holds, so that $d^* = p^*$, then plug in d^*, α^*, and β^* into the above derivation; then, all of the inequalities become equalities:

$$d^* = \Theta_D(\alpha^*,\beta^*) = \mathcal{L}(\mathbf{x}^*,\alpha^*,\beta^*) = f(\mathbf{x}^*) = p^*.$$

Thus, this happens iff something known as complimentary slackness holds, i.e. that

$$\sum_{i=1}^{m} \alpha_i^* g_i(\mathbf{x}^*) + \sum_{i=1}^{p} \beta_i^* h_i(\mathbf{x}^*) = 0.$$

However, the second term is already zero for all feasible $\mathbf{x}$, so this simplifies to $\alpha_i^* g_i(\mathbf{x}^*) = 0$ for all $i = 1,\dots,m$.

Related to this is the KKT Theorem, which realizes $\mathbf{x}^* \in \mathbb{R}^n$, $\alpha^* \in \mathbb{R}^m$, and $\beta^* \in \mathbb{R}^p$. Then, suppose that the problem is primal-feasible, so that $g_i(\mathbf{x}^*) \le 0$ for all i; that it is dual-feasible, so that $\alpha_i^* \ge 0$ for all i; complimentary slackness holds, so that $\alpha_i^* g_i(\mathbf{x}^*) \ge 0$ for all i; and $\nabla_{\mathbf{x}} \mathcal{L}(\mathbf{x}^*,\alpha^*,\beta^*) = 0$. Then, the theorem says that the problem has strong duality.

Example 15.1. Suppose the goal is to minimize $x_1^2 + x_2$ (i.e. $\mathbf{x} \in \mathbb{R}^2$) such that $2x_1 + x_2 \ge 4$ and $x_2 \ge 1$. First, some amount of rearranging will put this problem in a familiar form: $4 - 2x_1 - x_2 \le 0$, and $1 - x_2 \le 0$. Thus, the Lagrangian is

$$\mathcal{L}(\mathbf{x},\alpha) = x_1^2 + x_2 + \alpha_1(4 - 2x_1 - x_2) + \alpha_2(1 - x_2).$$

Then, the goal will to find as a function of α the minimum of the Lagrangian: this is the dual problem. So, take some derivatives: $\frac{\partial \mathcal{L}}{\partial x_1} = 2x_1 - 2\alpha_1 = 0$, so that $x_1 = \alpha_1$. Thus, the dual problem is just minimizing across x_2:

$$\begin{aligned}\Theta_D(\alpha) &= \min_{\mathbf{x}} \mathcal{L}(\mathbf{x},\alpha) = \min_{x_2} \left(\alpha_1^2 + x_2 + \alpha_1(4 - 2\alpha_1 - x_2) + \alpha_2(1 - x_2)\right)\\ &= \min_{x_2} \left(-\alpha_1^2 + 4\alpha_1 - \alpha_2 + x_2(1 - \alpha_1 - \alpha_2)\right)\\ &= \begin{cases} -\alpha_1^2 + 4\alpha_1 + \alpha_2, & \text{if } 1 - \alpha_1 - \alpha_2 = 0,\\ -\infty, & \text{otherwise.}\end{cases}\end{aligned}$$

Now, to write this as a convex problem, rather than a concave one, the goal becomes to maximize $\Theta_D(\alpha)$ over $\alpha \in \mathbb{R}^2$, so that $\alpha_1, \alpha_2 \ge 0$. Since $-\infty$ is a very bad thing to get when one is trying to maximize, it corresponds to another constraint: we can just maximize $-\alpha_1^2 + 4\alpha_1 + \alpha_2$, with the above constraints and such that $1 - \alpha_1 - \alpha_2 = 0$.

Example 15.2. Now, consider the SVM with slack variables. Here, the goal is to find

$$\min_{\mathbf{w},b,\xi} \frac{1}{2}||\mathbf{w}|| + C\sum_{i=1}^{m} \xi_i,$$

such that $y^{(i)}(\mathbf{w}^{\mathrm{T}}\mathbf{x}^{(i)} + b) \ge 1 - \xi_i$ for $i = 1,\dots,m$, and $-x^{(i)} \le 0$.

It's necessary to rewrite the first constraint in standard form, which is easy. Then, the Lagrangian becomes

16. The VC Bound and Model Selection: 10/28/13

Using the training error and generalization error from last class, suppose $\mathcal{H}$ is finite and $|\mathcal{H}| = k$, $\widehat{h} = \arg\min_{h\in\mathcal{H}} \widehat{\varepsilon}(h)$, and $h^* = \arg\min_{h\in\mathcal{H}} \varepsilon(h)$. Then, with probability $1-\delta$,

$$\varepsilon(\widehat{h}) \le \underbrace{\min_{h\in\mathcal{H}} \varepsilon(h)}_{\varepsilon(h^*)} + 2\sqrt{\frac{1}{2m}\log\frac{2k}{\delta}}. \tag{9}$$

The first term should be thought of as relating to the bias, and the second akin to the variance.

For example, let $\mathcal{H}_1$ be the set of linear functions, and $\mathcal{H}_5$ be the set of fifth-order polynomials. Then, $\mathcal{H}_1 \subset \mathcal{H}_5$, so the former is likely to have a larger bias and smaller variance, and the latter will have less bias, but a larger variance. In general, for a fixed m, as the degree of a set of polynomial functions increases. the training error decreases (since there's less bias or underfitting). However, the generalization error eventually increases. There are algorithms to find the optimal degree, as will be demonstrated later.

It will be instructive to rewrite (9) to provide intuition for the case of infinite $\mathcal{H}$.

Corollary 16.1. *Let $|\mathcal{H}| = k$. Then, to guarantee that $\varepsilon(\widehat{h}) \le \varepsilon(h^*) + 2\gamma$ holds with probability $1-\delta$, then it suffices to prove that*

$$m \ge \frac{1}{2\gamma^2}\log\frac{k}{\delta} = O\left(\frac{1}{\gamma^2}\log\frac{k}{\delta}\right).$$

This is called a *sample complexity bound*, and is particularly useful to understand how much training data is necessary in order to obtain a certain guarantee. Its rearranged forms provide information about the probability, the error, etc.

The log function grows very slowly, so the following result, while obviously false, is still somewhat useful.

"Theorem" 16.2. *For all z, $\log(z) \le 30$.*

For example, a logistic regression with d parameters will have $64d$ bits, so $|\mathcal{H}| = 2^{64d}$. This is a broken way of thinking about it, but now there's a useful approximation for the infinite case with the finite one. And thanks to big-O notation, the bound simplifies away to $m \ge O((d/\gamma^2)\log(1/\delta)) = O_{\gamma,\delta}(d)$.[27] In other words, the training size needed is roughly linear in the number of parameters.

There is a better way to prove this, but it's very involved; thus, the result will be presented without proof.

Definition. A hypothesis class $\mathcal{H}$ shatters a set $S = \{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(d)}\}$ if for any $S' \subseteq S$, there is an $h \in H$ such that $\{\mathbf{x}^{(i)} \mid h(\mathbf{x}^{(i)}) = 1\} = S'$, i.e. such that every possible labeling of S is realized by some $h \in \mathcal{H}$.

For example, the set of linear classifiers over $\mathbb{R}^2$ can shatter sets of three points, but no set of four points.

Definition. The VC dimension of a hypothesis class $\mathcal{H}$, denoted $\mathrm{VC}(\mathcal{H})$, is the size of the largest set it can shatter.

Thus, the VC dimension of the set of linear classifiers over $\mathbb{R}^2$ is 3. More generally, if $\mathcal{H}$ is the set of linear classifiers over $\mathbb{R}^n$, then $\mathrm{VC}(\mathcal{H}) = n+1$.

With these definitions, the theorem can be stated. Notice that it applies to finite and infinite $\mathcal{H}$.

Theorem 16.3. *Given a hypothesis class $\mathcal{H}$, then with probability $1-\delta$,*

$$\varepsilon(\widehat{h}) \le \varepsilon(h^*) + O\left(\sqrt{\frac{\mathrm{VC}(\mathcal{H})}{m}\log\frac{m}{\mathrm{VC}(\mathcal{H})} + \frac{1}{m}\log\frac{1}{\delta}}\right).$$

This is an error bound, and thus in much the same way to (9) one has the following corollary:

Corollary 16.4. *To guarantee that $\varepsilon(\widehat{h}) \le \varepsilon(h^*) + 2\gamma$ with probability $1-\delta$, it suffices that $m \ge O_{\gamma,\delta}(d)$.*

Usually, the VC dimension is approximately linear in the number of parameters in the algorithm; in the linear class, this was exact. It's also exact for logistic regression: if $\mathbf{x} \in \mathbb{R}^n$, then $\mathrm{VC}(\mathcal{H}) = n+1$.

But then, what about the SVM? In an infinite-dimensional feature space, would the VC dimension be infinite? The key insight is that the SVM is not trying to minimize training error, but to find a hypothesis with a large margin. It turns out that large margins correspond to a small VC dimension. If $\mathcal{H}$ is the set of linear classifiers with margin at least γ, for example, and $\|\mathbf{x}\|_2 \le R$, then

$$\mathrm{VC}(\mathcal{H}) \le \left\lceil \frac{R^2}{4\gamma^2} \right\rceil + 1 = O\left(\frac{1}{\gamma^2}\right).$$

Notice that this doesn't depend on the location of $\mathbf{x}$; it could be in some infinite-dimensional space and the result still holds.

[27] Here, the subscripts mean that the indicated variables are treated as constants.

Model Selection. The motivation of model selection is to try and choose what sort of $\mathcal{H}$ to classify from. One might have this parameterized by the number of parameters, the bandwidth, the SVM quantity C, etc. One algorithm that doesn't work is to simply minimize the training error.

Instead, there's a technique called holdout cross-validation (also simple CV), in which one randomly splits the data into two sets S_{train} (with about 70% of the data) and S_{CV} (the remaining 30%). Then, train each model in S_{train} and test it on S_{CV}, and choose the model that performs best. Notice how this minimizes the issue of overfitting relative to the simpler algorithm first described. Finally, once the model is selected, one can either output the model and then train on the whole dataset, or just stop with the hypothesis given by S_{train}. Either of these will end up working, and in some of the less stable learning algorithms (none of the ones yet discussed in this class), this can help preserve stability. Not retraining can also save time or computation.

One criticism of this is that in some cases, data may be very expensive or very hard-won, and leaving out some of it is really bad. Similarly, one might want to get a more complicated hypothesis, which requires more data. Thus, one could use a technique called k-fold cross-validation, which uses more data but is more computationally expensive. The data is split into k pieces, and then, for $1 \leq i \leq k$, train on the $k-1$ pieces and test on the remaining piece, the i^{th} piece. Then, average the errors from the k steps. The choice of $k = 10$ is by far the most common. Thus, there are Mk different training runs to iterate through if M models are considered, which is a lot. But it uses all of the data.

The final idea that will be discussed is called leave-one-out cross-validation. This is useful when data is difficult to obtain and very important to use (e.g. in the medical field, where each data point is a quantifiable amount of human suffering). This is just m-fold cross-validation: each training example is tested after a model has been trained on the other $m-1$ training examples. This is even more computationally expensive, so it should be avoided on larger data sets (e.g. 150 or more examples). In general, increasing the size of k in k-fold cross-validation creates better, but increasingly diminishing, returns, yet is more and more computationally expensive.

One common subcase of model selection is that of feature selection. Take as an example some text classification problem with 10000 features (e.g. because it is over 10000 words); some of these words are obviously going to be more relevant than others (such as "the" and "of," which don't really indicate whether an email is spam or not). The naïve way to select features gives 2^{10000} models, since there are that many subsets of the feature set! But even halving the number of features reduces the VC dimension and the probability of overfitting, so this would be a good thing to accomplish.

An algorithm called forward search is a heuristic to avoid running through all of these examples; it can be considered an approximate search algorithm. This is done as follows: start with the empty feature set, and then pick the feature that reduces the cross-validation error the most and add it. This requires iterating through all of the examples, but this is linear, not exponential! Then, repeat, adding the next most useful feature, and then the next most, and so on. This is an example of a greedy algorithm, but it's not guaranteed to find the optimum set of features, just a (generally) pretty good one.

A related algorithm called backwards search starts with the full set of features, and then removes the worst ones one at a time. This is described more in the lecture notes, but is less commonly used than forward search.

The problem with forward search is that it's still pretty computationally expensive, even though it's a drastic improvement on the total solution. Thus, a yet faster algorithm can be used on large data sets, called filter feature selection. In this algorithm, for each i, calculate a score that indicates how useful x_i is for estimating y. Then, select th k best features under this score. One reasonable score is the correlation between x_i and y, though it's far more common to use $\text{MI}(x_i, y)$, the mutual information between x and y, defined as

$$\text{MI}(x_i, y) = \sum_{x_i \in \{0,1\}} \sum_{y \in \{0,1\}} p(x,y) \log \frac{p(x,y)}{p(x)p(y)} = \text{KL}(p(x,y) \| \, p(x)p(y)),$$

i.e. how different the joint distribution is from the product of the two individual distributions. This concept arises in information theory.

In essence, one computes the scores, and then picks the k best. But how many features do you want? One good strategy is to just use optimal cross-validation.

17. Bayesian Statistics: 10/30/13

The techniques we discussed last time — model selection, feature selection, and so on — were ways of simplifying a problem to avoid overfitting. However, there's an alternate approach to this that uses ideas from Bayesian statistics.

Thus far, the calculation of the MLE for the logistic regresion has proceeded in a frequentist manner: the goal is to find the parameters θ, but this is a fixed value and it definitely exists. The goal is to estimate it. Alternatively, one could imagine that the universe (or God) is choosing θ at random, so the actual value doesn't exist. The uncertainty in θ is captured by a prior distribution $P(\theta)$.

In the case of logistic regression, suppose $\theta \sim \mathsf{N}(0, \tau^2 I)$ is the prior. Then, the goal is to, given a training set $S = \{\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^m$, to calculate $P(\theta \mid S) = P(S \mid \theta)P(\theta)/P(S)$, and one might choose $\arg\max_\theta P(\theta \mid S)$. This might seem like meaningles semantics, but in the history of statistics, this debate was quite significant!

The argmax actually ends up being[28]

$$\arg\max_\theta P(\theta \mid S) = \arg\max_\theta (S \mid \theta) P(\theta) = \arg\max_\theta \left(\prod_{i=1}^m P(y^{(i)} \mid \mathbf{x}^{(i)}, \theta) \right) P(\theta),$$

so after taking logs, one obtains

$$\arg\max_\theta \sum_{i=1}^m \log P(y^{(i)} \mid \mathbf{x}^{(i)}, \theta) - \lambda ||\theta||^2, \tag{10}$$

where $\lambda = \tau/2$. Notice this equation is very similar to the MLE for θ, except for the $-\lambda||\theta||^2$ term. This equates to shrinking θ, which often makes the hypothesis less complicated.

In general, adding a term like $-\lambda||\theta||^2$ to an expression to prevent overfitting is called regularization. It turns out that for text classification with logistic regression, regularization is a useful way to get more out of the same data; it yields a pretty good text classification algorithm. Since regularization is easy to implement, it's a great choice for decreasing overfitting. However, one needs a good value of λ: if it is too big, the hypothesis is underfit, so try several values of λ with cross-validation to pick the best one.

(10) is aso known as the MAP estimate (maximum *a posteriori* estimate).

Online Learning. Though it's unlikely to come up in the course project, online learning is very useful in the real world.

Thus far, we have considered batch learning, in which the training data and testing data are kept separate. Online learning takes a different approach, where testing examples can also be used to train the algorithm. For example, web applications of machine learning (especially ad networks) uses feedback from what users do to sharpen their own algorithms.

For example, user $\mathbf{x}^{(1)}$ shows up, and the algorithm makes a prediction $\hat{y}^{(1)}$. Then, the user does something, giving an actual example $y^{(1)}$. Then, this data is fed back into the algorithm, and the user continues on, creating more data points $\mathbf{x}^{(2)}$, $\hat{y}^{(2)}$, $y^{(2)}$, etc.

Definition. The total online error of such an algorithm is $\sum_{i=1}^m 1\{\hat{y}^{(i)} \neq y^{(i)}\}$.

One example of an online algorithm is the perceptron algorithm, in which $\theta = 0$ at the beginning; then, for the i^{th} example, update $\theta_j := \theta_j + \alpha(y^{(i)} - h_\theta(\mathbf{x}^{(i)}))x_j^{(i)}$, where h is the sign function rather than a logistic function. Thus, it isn't in fact a gradient-descent algorithm, but it's close and it's very quick to compute.

One interesting theorem (deferred to the lecture notes) shows that as long as $||\theta||$ is small, then the perceptron won't overfit, even if $\mathbf{x}$ lies in some infinite-dimensional space.

Advice for Applying Machine Learning Algorithms. In the real world, machine learning leads to a huge number of choices, and it's very helpful to learn some strategies to get learning algorithms to work in different settings. This section, while not very mathematical, may be among the hardest conceptually. The key ideas are diagnostics for debugging learning algorithms, error analysis and ablative analysis, and how to approach starting an algorithm, and the idea of premature optimization.

For example, suppose one wants to make a spam classifier with a small set of 100 words, using Bayesian logistic regression as in (10). However, it's consistently getting 20% error, which is unacceptable. What next? Here are some ideas.

- You can always throw more data at the problem.
- Try a smaller set of features. 100 is still a large number of features.
- Sometimes, you need a *larger* set of features. Occasionally, this works.
- Try using different features, e.g. including the email header.
- Try running gradient descent for more iterations.
- Maybe Newton's method?
- Use a different value of λ.
- Try with an SVM.

The first, best thing to do is to consider multiple options. This already places you ahead of many teams.

To determine whether the classifier has high bias or high variance, calculate the training error and the test error: if the training error is significantly lower, the variance is the problem. This makes a significant difference in which approach one should take.

One useful diagnostic is the learning curve, where m is plotted against the test error. See Figure 5 for more information.

In the case of high variance, it is useful to add more training examples or try a smaller set of features. However, trying a larger set of features or switching out the features fixes high bias. Knowing which one to test will be very helpful and save a lot of time. But in the case of high bias, where the test curve rapidly flattens out, and thus throwing more data at the problem will not fix it.

[28]Here, θ is considered a random variable, so one can condition on it, so one writes $P(y^{(i)} \mid \mathbf{x}^{(i)}, \theta)$ rather than $P(y^{(i)} \mid \mathbf{x}^{(i)}; \theta)$; θ is not a parameter.

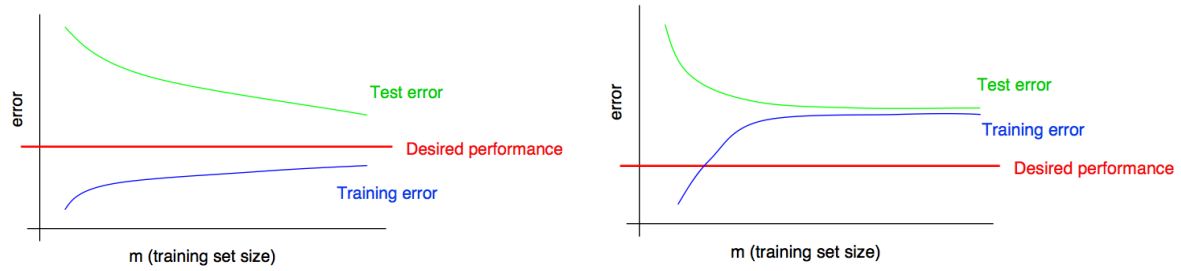


FIGURE 5. Left: a typical learning curve with high bias, in which there is a significant difference between the training error and testing error. Right: a typical learning curve with high bias, where the training error is about equal to the testing error. The plots might be noisier in practice, but the shape is fairly characteristic. Source: CS 229 Lecture Slides

Note that there are different diagnostics than just bias vs. variance, though it is most common. The next most common is the issue of convergence of the learning algorithm. For example, consider running a spam classifier that gets 2% error on both spam and non-spam; the former is OK, but we don't want to filter valid messages so frequently! Suppose also that an SVM works, but because of online learning it's better to go with the Naïve Bayes approach. Then, it might be worth checking how well the algorithm converges, graphing the number of iterations of the algorithm against the objective. However, it's generally hard to tell where the plot flattens.

Alternatively, are you even classifying the right function? One could adjust the tradeoffs by assigning weights to the different classifier values to promote misclassification of spam versus non-spam. For Bayesian logistic regression, one could also lower the norm by a different λ, for an SVM the constant C can be pushed around, etc.

If one has multiple models (e.g. Bayesian logistic regression and an SVM), and the goal is to maximize $a(\theta)$, one can look at the objective function $J(\theta)$ of the Bayesian algorithm. One can determine whether there is a good correlation between J and a: if optimizing one has no effect on the other, you have the wrong function!

In some sense, the accuracy function $a(\theta)$ is intractable: it's hard to optimize it, which is why J is used as a substitute. If J looks like a, then it's a good estimate, but the algorithm isn't working; correspondingly, it could be that J is being optimized well but is just totally off of the mark, in which case one wants a different J.

In this case, the specific diagnostic is whether $J(\theta_{\text{SVM}}) \leq J(\theta_{\text{BLR}})$. If this is true, then Bayesian logistic regression is not doing well (since $a(\theta_{\text{SVM}} > a(\theta_{\text{BLR}}))$, and the issue is with the algorithm. If instead the objective function is better for Bayesian logistic regression, then optimizing J is unrelated to optimizing a, and the problem lies with the objective function.

Once this is known, solutions exist: increasing the number of iterations or using Newton's method in place of gradient ascent fix the algorithm, but changing the parameters for λ or using an SVM allow one to fix the objective function J.

Part 3. Unsupervised Learning

18. CLUSTERING: 11/4/13

Before moving into the unsupervised learning material, there will be a bit more advice on applying machine learning. But first, a helicopter! An autonomous helicopter is simulated in some simulator, and uses reinforcement learning. There is some reward function, e.g. $R(\mathbf{s}) = -\|\mathbf{s} - \mathbf{s}_{\text{desired}}\|^2$, where $\mathbf{s}$ is the position of the helicopter, and the goal is to maximize $V^{\pi_{\text{RL}}} = E[R(\mathbf{s}_0) + \cdots + R(\mathbf{s}_T)]$, and obtain some reward policy π_{RL}.

Now, what happens if the resuting controller π_{RL} does much worse than a human pilot. How might one fix that?[29] Suppose that:

1. The helicopter simulation is accurate.
2. The reinforcement learning algorithm correctly controls the helicopter to maximize the expected payoff (i.e. minimizing the cost function).
3. Maximizing the expected payoff corresponds to correct autonomous flight.

These seem like they ought to force it to fly well, so which assumption (or assumptions) are wrong? Here are some diagnostics.

- If π_{RL} flies well in the simulator but not in the real world, then the simulator is not very accurate.
- If the human does better on the cost function than π_{RL} (in terms of minimizing sum-of-squares error), then the reinforcement learning algorithm isn't maximizing the reward function and a different algorithm (or a variant) should be used.

[29] This is not very straightforward; improving this specific algorithm made for at least two PhD theses.

- Correspondingly, if the reinforcement learning algorithm does better at maximising the cost function but is still less effective at flight, then the issue is with the reward function, which doesn't correspond to good autonomous flight.

Often, though, one has to invent one's own diagnostics. This is a good idea even when the algorithm is doing well, especially if your company's profit depends heavily on it. Having an intuitive understanding of how the problem works is quite helpful. It's also helpful in the case of a research paper, as it justifies why this is a good approach that can be used in future research.

Many applications have a pipeline of things that lead to the final classification. A proper error analysis should look at all aspects of the pipeline. For example, one could plug in manually preprocessed images to determine if image preprocessing is effective, or could manually classify faces to see how the rest of the algorithm works, etc. This shows where the upward potential is: for example, if manually preprocessing improves the classification accuracy from 85% to 85.1%, then maybe that's not the best place to work. This sort of error analysis could take half a day, and save six months (or more) of misplaced effort.

Error analysis tries to explain the difference between current performance and perfect performance. A different kind of analysis, called abalative analysis, compares the current performance with some simple baseline. For example, if one has a spam classifier with lots of fancy features, which ones are actually hepful? Thus, instead of adding components one at a time, one removes them one at a time. Features that aren't super useful can be elided out for faster performance.

Note that clasification accuracy has been used most of the time today, but sometimes this is a bad idea, especially when the classes are skewed. Thus, one might want to distinguish the importance of a false positive and a false negative, as in the case where a very rare, but very serious disease is detected (so that 99.9% accuracy can be gotten by saying everyone is negative). Thus, an alternate solution is to count the number of true/false negatives/positives. Then, determine which of a false positive or negative is worse? This means that the hypothesis function returns 1 if $\mathbf{w}^{\mathrm{T}}\mathbf{x}+b>\tau$, where τ is not necessarily zero. A larger value of τ errs on the side of predicting a false negative over a false positive.

19. k-MEANS: 11/6/13

20. FACTOR ANALYSIS: 11/11/13

Recall that for the EM algorithm, for some set of data $\mathbf{x}^{(1)},\dots,\mathbf{x}^{(m)}$, one alternates the E-step, where one calculates $Q_i(\mathbf{z}^{(i)})=P(\mathbf{z}^{(i)}\mid\mathbf{x}^{(i)};\theta)$, and the M-step, where one computes

$$\begin{aligned}\theta &:= \arg\max_\theta \sum_i \sum_{\mathbf{z}^{(i)}} Q_i(\mathbf{z}^{(i)}) \log \frac{P(\mathbf{z}^{(i)},\mathbf{x}^{(i)};\theta)}{Q_i(\mathbf{z}^{(i)})}\\ &= \arg\max_\theta \sum_i E_{\mathbf{z}^{(i)}\sim Q_i}\left[\log \frac{P(\mathbf{x}^{(i)},\mathbf{z}^{(i)};\theta)}{Q_i(\mathbf{z}^{(i)})}\right].\end{aligned}$$

In the continuous case, these sums can be replaced with integrals. But the second one, using the expected value, makes life easier in some cases. Where $Q_i\sim\mathsf{N}(\mu,\Sigma)$, $E_{\mathbf{z}^{(i)}\sim Q_i}[\mathbf{z}^{(i)}]=\mu$, so if $m\le n$, then this Gaussian is degenerate, and Σ is singular (which is bad for all of the algorithms that need Σ^{-1}). This does happen in the real world, e.g. if one takes data from a large number of sensors not all that many times.

In this case, there are a couple of things that one can do. For example, one could constrain Σ to be diagonal, with the diagonal entries

$$\sigma_j^2=\frac{1}{m}\sum_{i=1}^m (x_j^{(i)}-\mu_j)^2.$$

This forces the contours of the Gaussian to be axis-aligned, and reduces the number of parameters from n^2 to n. It can work well in practice, but makes the assumption that all of the features are independent (which is occasionally a bad assumption).

It's possible to make an even stronger assumption: that all of the σ_j^2 are the same. Geometrically, this means that the contours are circular.

In factor analysis, $\mathbf{x}$ is an observed random variable and $\mathbf{z}$ is a hidden (also called latent) random variable. It will be assumed that $\mathbf{z}\sim\mathsf{N}(0,I)$, where $\mathbf{z}\in\mathbb{R}^d$ for some $d<n$. It is also assumed that $\mathbf{x}=\mu+\Lambda\mathbf{z}+\varepsilon$, where $\varepsilon\sim\mathsf{N}(0,\Psi)$. The parameters to fit are $\mu\in\mathbb{R}^n$, $\Lambda\in\mathbb{R}^{n\times d}$, and $\Psi\in\mathbb{R}^{n\times n}$, where the covariance matrix Ψ is diagonal.

Once $\mathbf{z}$ is known, $\mathbf{x}\mid\mathbf{z}\sim\mathsf{N}(\mu+\Lambda\mathbf{z},\Psi)$.

For example, suppose $z\in\mathbb{R}$ and $\mathbf{x}\in\mathbb{R}^2$. Then, one will draw several examples $z^{(i)}$ from a Gaussian density. Then, if $\Lambda=\begin{bmatrix}2\\1\end{bmatrix}$ and $\mu=0$, then the $z^{(i)}$ are translated onto the line $y=2t$. The covariance matrix and ε add some noise above and below this line, and these are the $\mathbf{x}^{(i)}$.

Before discussing the EM algorithm for this model, it will be useful to discuss properties of the multivariate Gaussian. In particular, if $\mathbf{x}$ is a high-dimensional vector, then it may be useful to write $\mathbf{x}=\begin{bmatrix}\mathbf{x}_1\\\mathbf{x}_2\end{bmatrix}$, where $\mathbf{x}\in\mathbb{R}^n$, $\mathbf{x}_1\in\mathbb{R}^r$, and $\mathbf{x}_2\in\mathbb{R}^s$.

If $\mathbf{x} \sim \mathsf{N}(\mu, \Sigma)$, one can decompose μ in exactly the same way and $\Sigma = \begin{bmatrix} \Sigma_{11} & \Sigma_{12} \\ \Sigma_{21} & \Sigma_{22} \end{bmatrix}$, with each Σ_{ij} over the appropriate dimension (e.g. $\Sigma_{21} \in \mathbb{R}^{s \times r}$).

Then, $\mathbf{x}_i \sim \mathsf{N}(\mu_1, \Sigma_{11})$. The conditional distribution is a bit trickier, because it involves dividing one integral by another (as $P(\mathbf{x}_1, \mathbf{x}_2)/P(\mathbf{x}_2)$ for $P(\mathbf{x}_1 \mid \mathbf{x}_2)$). But it ends up being $\mathbf{x}_1 \mid \mathbf{x}_2 \sim \mathsf{N}(\mu_{1|2}, \Sigma_{1|2})$, where $\mu_{1|2} = \mu_1 + \Sigma_{12}\Sigma_{22}^{-1}(\mathbf{x}_2 - \mu_2)$ and $\Sigma_{1|2} = \Sigma_{11} - \Sigma_{12}\Sigma_{22}^{-1}\Sigma_{21}$.

Using this, one can write that $\begin{bmatrix} \mathbf{z} \\ \mathbf{x} \end{bmatrix} \sim \mathsf{N}(\mu_{\mathbf{x},\mathbf{z}}, \Sigma)$. But what are these values? Since $E[\mathbf{z}] = 0$ and $E[\mathbf{x}] = E[\mu + \Lambda\mathbf{z} + \varepsilon] = \mu$ (since the expectations of the other values go to zero), then $\mu_{\mathbf{x},\mathbf{z}} = E\begin{bmatrix} \mathbf{z} \\ \mathbf{x} \end{bmatrix} = \begin{bmatrix} 0 \\ \mu \end{bmatrix}$.

For Σ, one wants $A = E\left[\begin{bmatrix} \mathbf{z} \\ \mathbf{x} \end{bmatrix} - \mu_{\mathbf{x},\mathbf{z}}\right]$, and then $\Sigma = AA^{\mathrm{T}}$, using the formula from the covariance. Then, Σ is found componentwise, e.g.

$$\begin{aligned}
\Sigma_{22} &= E\left[(\mu + \Lambda\mathbf{z} - \mu)(\mu + \Lambda\mathbf{z} - \mu)^{\mathrm{T}}\right] \\
&= E[(\Lambda\mathbf{z} + \varepsilon)(\Lambda\mathbf{z} + \varepsilon)^{\mathrm{T}}] \\
&= E[\Lambda\mathbf{z}\mathbf{z}^{\mathrm{T}}\Lambda^{\mathrm{T}} + \varepsilon\varepsilon^{\mathrm{T}} + \Lambda\mathbf{z}\varepsilon^{\mathrm{T}} + \varepsilon\mathbf{z}^{\mathrm{T}}\Lambda] \\
&= \Lambda E[\mathbf{z}\mathbf{z}^{\mathrm{T}}]\Lambda^{\mathrm{T}} + E[\varepsilon\varepsilon^{\mathrm{T}}] \\
&= \Lambda\Lambda^{\mathrm{T}} + \Psi.
\end{aligned}$$

21. Principal Components Analysis and Independent Components Analysis: 11/13/13

Suppose one has a dataset with tens of thousands of features, but that is probably actually lower-dimensional. Principal components analysis (PCA) is a dimensionality reduction algorithm that can reduce memory usage and time taken to run other machine learning algorithms. More concretely, suppose $\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(n)} \in \mathbb{R}^n$ and one wants to reduce the data to be k-dimensional, where $k \ll n$. For example, suppose a set of features included the length in inches and the length in centimeters. But due to rounding error, it's not immediate how to get the data all on one plane. While this example is artificial, when there are large features, similarly correlated features tend to appear, though the relationships tend to be less obvious.

In order to use PCA, there's a little bit of preprocessing. Specifically, one needs to translate the mean to zero. Let $\mu = (1/m)\sum_i \mathbf{x}^{(i)}$ and $\mathbf{x}^{(i)} \leftarrow \mathbf{x}^{(i)} - \mu$. The variance is also standaridezed: let $\sigma_j^2 = (1/n)\sum_i (x_j^{(i)})^2$, and then setting $x_j^{(i)} \leftarrow x_j^{(i)} / \sqrt{\sigma_j}$. This is to ensure that larger features contribute equally to the smaller one.

Now, one can ask: what are the main directions in which the data varies? It would also be nice to have a mathematical characterization for this; the way to do this is to measure the variance of the data when it is projected onto the subspace corresponding to the reduced dimensions. The goal is to maximize this variance (corresponding to preserving as much of the data as possible while reducing the dimension).

Thus, the goal is to pick the unit vector $\mathbf{u}$ such that the variance of the lengths of the projections (i.e. $\mathbf{u}^{\mathrm{T}}\mathbf{x}^{(i)}$) is greatest. In other words, the problem is

$$\max_{\|\mathbf{u}\|=1} \frac{1}{m} \sum_{i=1}^{m} (\mathbf{x}^{(i)\mathrm{T}}\mathbf{u})^2.$$

Let $\Sigma = (1/m)\sum_i \mathbf{x}^{(i)}\mathbf{x}^{(i)\mathrm{T}}$ (i.e. the covariance matrix), then this is equivalent to maimizing $\mathbf{u}^{\mathrm{T}}\Sigma\mathbf{u}$. It turns out this happens when $\mathbf{u}$ is an eigenvector of Σ. This is because, when setting up the Lagrangian, $\mathscr{L}(\mathbf{u}, \lambda) = \mathbf{u}^{\mathrm{T}}\Sigma\mathbf{u} - \lambda(\mathbf{u}^{\mathrm{T}}\mathbf{u} - 1)$, so after the gradient, $\nabla_{\mathbf{u}}\mathscr{L} = \Sigma\mathbf{u} - \lambda\mathbf{u} = 0$. But fortunately, there are numerical and computational methods for handling this problem.

In order to project this data onto a k-dimensional subspace, one chooses the top k unit eigenvectors of Σ, $\mathbf{u}_1, \ldots, \mathbf{u}_k$. More precisely, if $\mathbf{x}^{(i)} \in \mathbb{R}^n$, then its projection is $\mathbf{y}^{(i)} = (\mathbf{u}_1^{\mathrm{T}}\mathbf{x}^{(i)}, \ldots, \mathbf{u}_k^{\mathrm{T}}\mathbf{x}^{(i)}) \in \mathbb{R}^k$ (i.e. $\mathbf{y}^{(i)}$ is a vector, and each of these entries is a scalar).

Another way of approaching this problem is to try to minimize the sum-of-squares if the distance from the data to the subspace. This is formally equivalent, the proof of which is left to the next problem set.

One application of PCA is visualization of high-dimensional data; for example, it's not possible to plot 50-dimensional data, but reducing it to two or three dimensional helps. PCA is also used to compress the data, which is useful not as much for memory reasons (since memory is cheap), but for increasing the efficiency of the algorithm. This is useful for the several useful algorithms that are quadratic (or worse; Newton's method is cubic) in the number of features.

However, sometimes people use PCA to reduce the number of features to reduce overfitting. This is usually unsuccessful; adding a Bayesian prior tends to be a better idea.

PCA can also be used for an application called matching, or data cleaning. If one has noisy data (e.g. image recognition), two images can be compated by checking the Euclidean difference in $\mathbb{R}^n$, but if PCA captures intrinsic features of the objects, then comparing their distances in the subspace is often cleaner, and can be used to determine if two pictures correspond to the second object.

This is all nice and wonderful, but before using PCA, one should check what happens with the original data. This is a great example of premature optimization. It's not that PCA is a bad algorithm; it's very useful, but is also somewhat overused.

Now, ther are four unsupervised learning algorithms to use; when is each best? It depends what one wants to do; these can be divided into models of $P(\mathbf{x})$ and non-probabilstic models. Additionally, the goal might be to find lower-dimensional subspaces in the data, or clusters. Specifically:

- If one wants to model $P(\mathbf{x})$ and use subspaces, use factor analysis.
- If one wants to use subspaces but non-probabilistically, use PCA.
- If one wants to model $P(\mathbf{x})$ and obtain clusters, use mixture of Gaussians.
- If one wants to use clusters but non-probabilistically, use k-means.

22. Independent Components Analysis: 11/18/13

To recall, suppose we have a cumulative distribution function $F(s) = P(S \leq s)$, so that $P_S(s) = F'(s)$. In the independent components analysis probem, one has n speakers (or microphones) to separate out, where $s_j^{(i)}$ is the emission of the speaker j at time i. Then, one has $\mathbf{s}^{(i)}, \mathbf{x}^{(i)} \in \mathbb{R}^n$, where $\mathbf{x}^{(i)} = A\mathbf{s}^{(i)}$ for some mixing matrix A. Then, the goal is to find its inverse, $W = A^{-1}$.

A good way to do this is to take $P(\mathbf{x}^{(i)}; W)$ and do some maximum likelihood estimation. One could try to write a density function $p_{\mathbf{s}}(\mathbf{s})$, so that $p_{\mathbf{x}}(\mathbf{x}) = p_{\mathbf{s}}(W\mathbf{s})$, but *this is wrong*. Consider $p_s(s) \sim \mathsf{Uni}(0,1)$, and let $x = 2s$. Then, $x \sin \mathsf{Uni}(0,2)$, so $p_x(x) = 1/2 p_s(Ws)$. Thus, the correct formula looks like this:

$$p_{\mathbf{x}}(\mathbf{x}) = p_{\mathbf{s}}(W\mathbf{s}) \det(W).$$

In the ICA model, the $\mathbf{s}^{(i)}$ are assumed to be independent, sp $p(\mathbf{s}) = \prod_{i=1}^n P_{\mathbf{s}}(\mathbf{s}^{(i)})$, so

$$p(\mathbf{x}) = \det(W) \left(\prod_{i=1}^n p_{\mathbf{s}}(\mathbf{w}_i^{\mathrm{T}} \mathbf{x}) \right),$$

where $\mathbf{w}_i$ is the i^{th} column of W.

Since one wants to write down a CDF that isn't for a Gaussian, the best choice is the logistic function $g(s) = 1/(1+e^{-s})$. This is nicer than the Gaussian, because it has "fatter tails:" it tends to zero less quickly than the Gaussian, which is useful because these sorts of applications sometimes produce very large values. Then, the log-likelihood is

$$\ell(W) = \sum_{i=1}^m \log \left(\prod_j P_{\mathbf{s}}(\mathbf{w}_j^{\mathrm{T}} \mathbf{x}^{(i)}) \right) \det(W).$$

Then, one uses gradient-ascent as normal: $W \coloneqq W + 2\nabla_W \ell(W)$. The actual calculation of the gradient is deferred to the lecture notes.

One interesting application of this is separating out EEG data, removing for example irregularities induced by heartbeats. This is actually the standard use in industry. Another interesting application that is a little more crazy futuristic is to use ICA to find independent components in image data... which correspond to edges in the image. This is a surprisingly effective way to find them, though a little more has to be done to make it work.

Moving on to an introduction to reinforcement learning, I'm not sure what it exactly is yet, but you use it to fly helicopters. The goal here is to not supply the right answer to the problem, but instead some sort of reward function, much like the Pavlovial training of a dog. For example, in a game of chess, there isn't generally a "right" move, but one can design a reward function that favors good moves.

Definition. A Markov decision process (MDP) is a 5-tuple $(S, A, \{P_{sa}\}, \gamma, R)$, where:

- S is some set of states,
- A is a set of actions,
- P_{sa} is a state transition probability, so that the whole set sums to 1,
- γ is a discount factor, and
- R is a reward function.

As an example, suppose one has a 3×4 grid with a robot[30] that can move in any cardinal direction: $A = \{N, S, E, W\}$. At grid location $(3,1)$, it has a probability 0.8 of moving north and 0.1 of moving each of east or west; this is written

[30] Unfortunately, this isn't Karel.

$P_{(3,1),N}((3,2)) = 0.8$, $P_{(3,1),N}((4,1)) = 0.1$, etc. Then, each square has a reward associated with it, e.g. $R(4,3) = 1$, $R(4,2) = -1$, and so on. Generally, one might set $R(S) = -0.02$ for all other states, so that it doesn't move around too much.

Part 4. Reinforcement Learning and Control

23. Introduction to Reinforcement Learning: 11/20/13

Recall that a Markov decision process is a tuple $(S, A < \{P_{sa}\}, \gamma, R)$ as defined before, and the goal is to find a policy $\pi : S \to A$ that maximizes the reward. The value of a state is the expected value of the immediate reward plus γ times the future reward:

$$V^\pi(s) = E[R(s_0) + \gamma R(s_1) + \cdots \mid \pi, s_0 = s].$$

Note that since π isn't a random variable, it's an abuse of notation to condition on it. One good way to rewrite it is called Bellman's equation:

$$V^\pi(s) = R(s) + \gamma \sum_{s'} P_{s,\pi(s)}(s') V^\pi(s').$$

The optimal value function is $V^*(s) = \max_\pi V^\pi(s)$. Basically, one plans to keep acting optimally, so both instances of V^π in the previous equation can be replaced with V^*. In other words,

$$V^*(s) = R(s) + \max_a \sum_{s'} P_{sa}(s') V^*(s').$$

Then, the optimal policy is

$$\pi^*(s) = \arg\max_a \gamma \sum_{s'} P_{sa}(s') V^*(s).$$

Of course, this is exponential in $|A|$, and is not pleasant to compute. A better idea is an algorithm called value iteration:

- Initialize $V(s) = 0$ for all s.
- For every s, update

$$V(s) \coloneqq R(s) + \max_a \gamma \sum_{s'} P_{sa}(s') V(s').$$

There's actually a slight ambiguity in the above algorithm. One could update $V(s)$ for all states at once, which is called synchronous updating, or update each one for the computation of the next, which is called asynchronous. Both of these converge to the optimal value, though for the synchronous update it's much easier to show (and correspondingly, the asynchronous version is easier to implement). Typically, it can be thought of as $V \coloneqq B(V)$ for some linear operator B. Then, π^* can be found by substituting into one of the previous equations.

This is all fine if all of the values are known, but sometimes this isn't the case. What if P_{sa} and/or R are unknown? Typically, this means the best idea is to estimate these by collecting some data. Some cases may not appear in the testing data, so rather than dividing by zero, initialize them to $1/|S|$. In summary, repeat the following:

- Take actions using π to get experience in an MDP.
- Update the estimates for P_{sa}.
- Use value iteration to estimate V^*.
- Update $\pi(s) = \arg\max_a \sum_{s'} P_{sa}(s') V^*(s)$.

Okay, excellent. But often, the state space is continuous, which is a wrinkle that must be dealt with in many real-world applications such as autonomous driving, where there is a six-dimensional space: x, y, z, $\dot{x}$, $\dot{y}$, and $\dot{z}$. For helicopters one also needs three axes of rotation and their derivatives. One tends to write $s \in \mathbb{R}^n$, though technically you would want to use the rotation group instead of some of the components. Generally, one discretizes the problem, by taking a large finite number of continuous states and using them as the set S.

Discretization isn't just for reinforcement learning: for a regression model, one could discretize at the data points, obtaining a piecewise constant function that is fairly close to the original one.

The advantage of discretization is that it's easy to implement and greatly simplifies the problem. However, as the number of dimensional increases, it takes more and more points, increasing exponentially. This is known as the "curse of dimensionality." It's more pervasive than one might think: if the state space is even six-dimensional, one must start to be careful how the dimensions are discretized, e.g. picking more points in one direction. Anything past $\mathbb{R}^8$ is out of the question.

In these cases, it is better to approximate $V^*(s)$ as a function of s. Here, it's necessary to assume there is a model or simulator of the MDP, i.e. a black box that inputs a state s and an action a and outputs an $s' \sim P_{sa}$. Models could come out of physics simulators Alternatively, one can just use machine learning to get a model.

24. State Action Models and Linear Dynamical Systems: 12/2/13

"If you close your eyes and drive your car… don't do this… you can kind of stay in your lane for a few seconds. When I was younger, to impress my date… don't ever do this… I was driving with the lights off at night. Well, it turns out that driving with your headlights off is a bad idea."

Recall that we talked about the value iteration algorithm, were $V(S)$ gets iteratively updated as

$$V(s) \coloneqq R(s) + \max_a \gamma \sum_{s'} P_{sa}(s')V(s').$$

Then, $V \to V^*$ asymptotically. This uses $\pi^*(s) = \arg\max_a \sum_{s'} P_{sa}(s')V^*(s')$ to calculate more reasonably.

In a state-action reward model, one has $R : S \times A \to \mathbb{R}$, rather than just $R : S \to \mathbb{R}$, i.e. the actions are factored into the reward too. This could incentivize a robot to stay in place if it costs too much power to move, etc. Then, one has the followijng form of Bellman's equation:

$$V^*(s) = \max_a R(s,a) + \gamma \sum_{s'} P_{sa}(s')V^*(s'),$$

in other words a sum of the immediate reward and the anticipated future payoff. The factor γ controls how these are balanced.

Definition. A finite horizon MDP is a 5-tuple $(S, A, \{P_{sa}\}, T, R)$, where most of the quantities are the same as in a standard MPD, but where T is a horizon time, corresponding to some finite amount of time the model is allowed to run (e.g. a plane with a finite amount of fuel).

This means that the optimal action isn't always just a function of the state. For example, if a reward of $+1$ is close by and a reward of $+100$ is far away, then which one to go for depends on how much time is left. One says that the optimal policy is *non-stationary*, i.e. time-dependent. Thus, the non-stationary transition probabilities are $s_{t+1} \sim P^{(t)}_{s_t a_t}$, and the reward function is also non-stationary, as $R^{(t)}(s_t, a_t)$. So now

$$V^*_t(s) = E\left[\sum_{i=t}^{T} R^{(i)}(s_i, a_i) \mid \pi^*, s_t = s\right].$$

Value iteration now becomes a dynamic programming algorithm:

$$V^*_t(s) = \max_a R^{(t)}(s,a) + \sum_{s'} P^{(t)}_{sa}(s')V^*_{t+1}(s'),$$

where the last term is at time $t+1$ because time advances for calculating the future payoff. The general lack of discounting is because for finite horizon MDPs, convention ignores it. The above equation is the inductive step; the base case is $V^*_T(s) = \max_a R^{(T)}(s,a)$; then, one computes V^*_t for decreasing values of t. Then, one can compute π^* in the usual way:

$$\pi^*_t = \arg\max_a R^{(t)}(s,a) + \sum_{s'} P^{(t)}_{sa}(s')V^*_{t+1}(s).$$

This means that it's time-dependent, finessing the problem raised earlier.

This finite horizon model really opens up the range of problems that can be handled by reinforcement learning. It leads into an algorithm called linear quadratic regulation (LQR), which computes the optimal value exactly. It only works for some problems, though. Let the state space be $S = \mathbb{R}^n$, and the set of actions be $A = \mathbb{R}^d$, which is completely reasonable for a lot of cases. It then makes a stricter assumption that $P_{sa} : s_{t+1} = A^{(t)}s_t + B^{(t)}a_t + \omega_t$, where (leaving off the non-stationaryness to simplify the derivation) $\omega_t \sim \mathsf{N}(0, \Sigma_\omega)$, $s_{t+1} \mid s_t, a_t \sim \mathsf{N}(As_t + Ba_t, \Sigma_\omega)$, $A \in \mathbb{R}^{n\times n}$, and $B \in \mathbb{R}^{n\times d}$. Then, the reward function is $R(s_t, a_t) = -(s_t^{\mathrm{T}} U s_t + a_t^{\mathrm{T}} V a_t)$, where $U \in \mathbb{R}^{n\times n}$, $V \in \mathbb{R}^{d\times d}$, and both are positive semidefinite, so the reward function is negative (and the goal is to keep it close to zero). This could make one want $s_t \approx 0$, so if $U = I$ and $V = I$, then $R(s_t, a_t) = -\|s_t\|^2 - \|a_t\|^2$.

Thus, to calculate this model one could find

$$\min_{A,B} \sum_{i=1}^{m} \sum_{t=0}^{T-1} \left\| s^{(i)}_{t+1} - (As^{(i)}_t + Ba^{(i)}_t) \right\|^2.$$

Then, one must linearize the function found.