CS229 Lecture notes

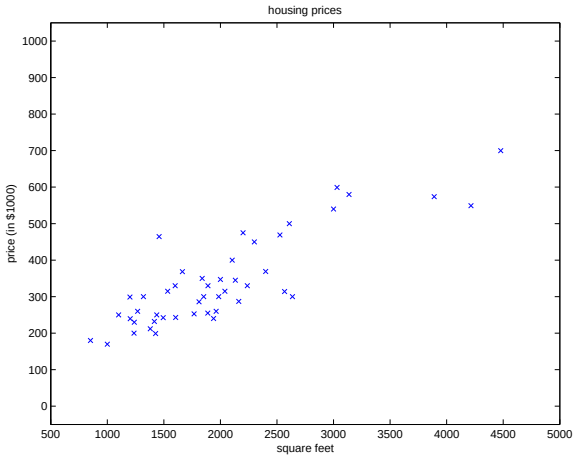
Andrew Ng

Supervised learning

Lets start by talking about a few examples of supervised learning problems. Suppose we have a dataset giving the living areas and prices of 47 houses from Portland, Oregon:

Living area (feet^2)	Price (1000$s)
2104	400
1600	330
2400	369
1416	232
3000	540
⋮	⋮

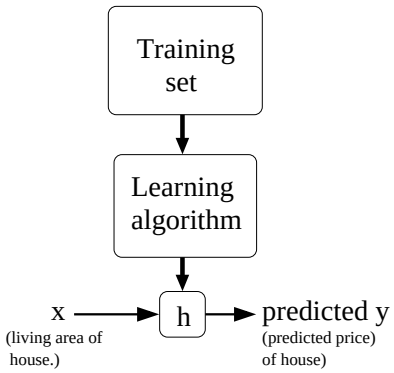
We can plot this data:



Given data like this, how can we learn to predict the prices of other houses in Portland, as a function of the size of their living areas?

To establish notation for future use, we'll use $x^{(i)}$ to denote the "input" variables (living area in this example), also called input **features**, and $y^{(i)}$ to denote the "output" or **target** variable that we are trying to predict (price). A pair $(x^{(i)}, y^{(i)})$ is called a **training example**, and the dataset that we'll be using to learn—a list of m training examples $\{(x^{(i)}, y^{(i)}); i = 1, \ldots, m\}$—is called a **training set**. Note that the superscript "(i)" in the notation is simply an index into the training set, and has nothing to do with exponentiation. We will also use $\mathcal{X}$ denote the space of input values, and $\mathcal{Y}$ the space of output values. In this example, $\mathcal{X} = \mathcal{Y} = \mathbb{R}$.

To describe the supervised learning problem slightly more formally, our goal is, given a training set, to learn a function $h : \mathcal{X} \mapsto \mathcal{Y}$ so that $h(x)$ is a "good" predictor for the corresponding value of y. For historical reasons, this function h is called a **hypothesis**. Seen pictorially, the process is therefore like this:



When the target variable that we're trying to predict is continuous, such as in our housing example, we call the learning problem a **regression** problem. When y can take on only a small number of discrete values (such as if, given the living area, we wanted to predict if a dwelling is a house or an apartment, say), we call it a **classification** problem.

Part I
Linear Regression

To make our housing example more interesting, lets consider a slightly richer dataset in which we also know the number of bedrooms in each house:

Living area (feet^2)	#bedrooms	Price (1000$s)
2104	3	400
1600	3	330
2400	3	369
1416	2	232
3000	4	540
$\vdots$	$\vdots$	$\vdots$

Here, the x's are two-dimensional vectors in $\mathbb{R}^2$. For instance, $x_1^{(i)}$ is the living area of the i-th house in the training set, and $x_2^{(i)}$ is its number of bedrooms. (In general, when designing a learning problem, it will be up to you to decide what features to choose, so if you are out in Portland gathering housing data, you might also decide to include other features such as whether each house has a fireplace, the number of bathrooms, and so on. We'll say more about feature selection later, but for now lets take the features as given.)

To perform supervised learning, we must decide how we're going to represent functions/hypotheses h in a computer. As an initial choice, lets say we decide to approximate y as a linear function of x:

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2$$

Here, the θ_i's are the **parameters** (also called **weights**) parameterizing the space of linear functions mapping from $\mathcal{X}$ to $\mathcal{Y}$. When there is no risk of confusion, we will drop the θ subscript in $h_\theta(x)$, and write it more simply as $h(x)$. To simplify our notation, we also introduce the convention of letting $x_0 = 1$ (this is the **intercept term**), so that

$$h(x) = \sum_{i=0}^{n} \theta_i x_i = \theta^T x,$$

where on the right-hand side above we are viewing θ and x both as vectors, and here n is the number of input variables (not counting x_0).

Now, given a training set, how do we pick, or learn, the parameters θ? One reasonable method seems to be to make $h(x)$ close to y, at least for

the training examples we have. To formalize this, we will define a function that measures, for each value of the θ's, how close the $h(x^{(i)})$'s are to the corresponding $y^{(i)}$'s. We define the **cost function**:

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

If you've seen linear regression before, you may recognize this as the familiar least-squares cost function that gives rise to the **ordinary least squares** regression model. Whether or not you have seen it previously, lets keep going, and we'll eventually show this to be a special case of a much broader family of algorithms.

1 LMS algorithm

We want to choose θ so as to minimize $J(\theta)$. To do so, lets use a search algorithm that starts with some "initial guess" for θ, and that repeatedly changes θ to make $J(\theta)$ smaller, until hopefully we converge to a value of θ that minimizes $J(\theta)$. Specifically, lets consider the **gradient descent** algorithm, which starts with some initial θ, and repeatedly performs the update:

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

(This update is simultaneously performed for all values of $j = 0, \ldots, n$.) Here, α is called the **learning rate**. This is a very natural algorithm that repeatedly takes a step in the direction of steepest decrease of J.

In order to implement this algorithm, we have to work out what is the partial derivative term on the right hand side. Lets first work it out for the case of if we have only one training example (x, y), so that we can neglect the sum in the definition of J. We have:

$$\begin{aligned}
\frac{\partial}{\partial\theta_j}J(\theta) &= \frac{\partial}{\partial\theta_j}\frac{1}{2}\left(h_\theta(x) - y\right)^2 \\
&= 2\cdot\frac{1}{2}\left(h_\theta(x) - y\right)\cdot\frac{\partial}{\partial\theta_j}(h_\theta(x) - y) \\
&= \left(h_\theta(x) - y\right)\cdot\frac{\partial}{\partial\theta_j}\left(\sum_{i=0}^{n}\theta_i x_i - y\right) \\
&= \left(h_\theta(x) - y\right)x_j
\end{aligned}$$

For a single training example, this gives the update rule:[1]

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

The rule is called the **LMS** update rule (LMS stands for "least mean squares"), and is also known as the **Widrow-Hoff** learning rule. This rule has several properties that seem natural and intuitive. For instance, the magnitude of the update is proportional to the **error** term $(y^{(i)} - h_\theta(x^{(i)}))$; thus, for instance, if we are encountering a training example on which our prediction nearly matches the actual value of $y^{(i)}$, then we find that there is little need to change the parameters; in contrast, a larger change to the parameters will be made if our prediction $h_\theta(x^{(i)})$ has a large error (i.e., if it is very far from $y^{(i)}$).

We'd derived the LMS rule for when there was only a single training example. There are two ways to modify this method for a training set of more than one example. The first is replace it with the following algorithm:

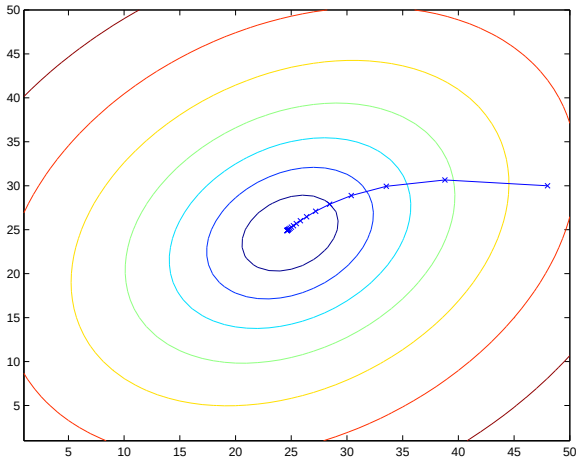
Repeat until convergence {

$$\theta_j := \theta_j + \alpha \sum_{i=1}^{m} \left(y^{(i)} - h_\theta(x^{(i)}) \right) x_j^{(i)} \qquad \text{(for every } j\text{)}.$$

}

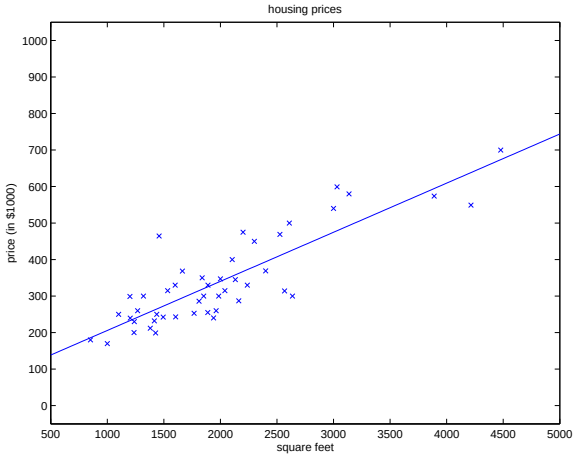
The reader can easily verify that the quantity in the summation in the update rule above is just $\partial J(\theta)/\partial \theta_j$ (for the original definition of J). So, this is simply gradient descent on the original cost function J. This method looks at every example in the entire training set on every step, and is called **batch gradient descent**. Note that, while gradient descent can be susceptible to local minima in general, the optimization problem we have posed here for linear regression has only one global, and no other local, optima; thus gradient descent always converges (assuming the learning rate α is not too large) to the global minimum. Indeed, J is a convex quadratic function. Here is an example of gradient descent as it is run to minimize a quadratic function.

[1] We use the notation "$a := b$" to denote an operation (in a computer program) in which we *set* the value of a variable a to be equal to the value of b. In other words, this operation overwrites a with the value of b. In contrast, we will write "$a = b$" when we are asserting a statement of fact, that the value of a is equal to the value of b.



The ellipses shown above are the contours of a quadratic function. Also shown is the trajectory taken by gradient descent, with was initialized at (48,30). The x's in the figure (joined by straight lines) mark the successive values of θ that gradient descent went through.

When we run batch gradient descent to fit θ on our previous dataset, to learn to predict housing price as a function of living area, we obtain $\theta_0 = 71.27$, $\theta_1 = 0.1345$. If we plot $h_\theta(x)$ as a function of x (area), along with the training data, we obtain the following figure:



If the number of bedrooms were included as one of the input features as well, we get $\theta_0 = 89.60, \theta_1 = 0.1392$, $\theta_2 = -8.738$.

The above results were obtained with batch gradient descent. There is an alternative to batch gradient descent that also works very well. Consider the following algorithm:

Loop {

for i=1 to m, {

$\theta_j := \theta_j + \alpha \left(y^{(i)} - h_\theta(x^{(i)})\right) x_j^{(i)}$ (for every j).

}

}

In this algorithm, we repeatedly run through the training set, and each time we encounter a training example, we update the parameters according to the gradient of the error with respect to that single training example only. This algorithm is called **stochastic gradient descent** (also **incremental gradient descent**). Whereas batch gradient descent has to scan through the entire training set before taking a single step—a costly operation if m is large—stochastic gradient descent can start making progress right away, and continues to make progress with each example it looks at. Often, stochastic gradient descent gets θ "close" to the minimum much faster than batch gradient descent. (Note however that it may never "converge" to the minimum, and the parameters θ will keep oscillating around the minimum of $J(\theta)$; but in practice most of the values near the minimum will be reasonably good approximations to the true minimum.[2]) For these reasons, particularly when the training set is large, stochastic gradient descent is often preferred over batch gradient descent.

2 The normal equations

Gradient descent gives one way of minimizing J. Lets discuss a second way of doing so, this time performing the minimization explicitly and without resorting to an iterative algorithm. In this method, we will minimize J by explicitly taking its derivatives with respect to the θ_j's, and setting them to zero. To enable us to do this without having to write reams of algebra and pages full of matrices of derivatives, lets introduce some notation for doing calculus with matrices.

[2]While it is more common to run stochastic gradient descent as we have described it and with a fixed learning rate α, by slowly letting the learning rate α decrease to zero as the algorithm runs, it is also possible to ensure that the parameters will converge to the global minimum rather then merely oscillate around the minimum.

2.1 Matrix derivatives

For a function $f : \mathbb{R}^{m\times n} \mapsto \mathbb{R}$ mapping from m-by-n matrices to the real numbers, we define the derivative of f with respect to A to be:

$$\nabla_A f(A) = \begin{bmatrix} \frac{\partial f}{\partial A_{11}} & \cdots & \frac{\partial f}{\partial A_{1n}} \\ \vdots & \ddots & \vdots \\ \frac{\partial f}{\partial A_{m1}} & \cdots & \frac{\partial f}{\partial A_{mn}} \end{bmatrix}$$

Thus, the gradient $\nabla_A f(A)$ is itself an m-by-n matrix, whose (i,j)-element is $\partial f/\partial A_{ij}$. For example, suppose $A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$ is a 2-by-2 matrix, and the function $f : \mathbb{R}^{2\times 2} \mapsto \mathbb{R}$ is given by

$$f(A) = \frac{3}{2}A_{11} + 5A_{12}^2 + A_{21}A_{22}.$$

Here, A_{ij} denotes the (i,j) entry of the matrix A. We then have

$$\nabla_A f(A) = \begin{bmatrix} \frac{3}{2} & 10A_{12} \\ A_{22} & A_{21} \end{bmatrix}.$$

We also introduce the **trace** operator, written "tr." For an n-by-n (square) matrix A, the trace of A is defined to be the sum of its diagonal entries:

$$\mathrm{tr}A = \sum_{i=1}^{n} A_{ii}$$

If a is a real number (i.e., a 1-by-1 matrix), then $\mathrm{tr}\, a = a$. (If you haven't seen this "operator notation" before, you should think of the trace of A as $\mathrm{tr}(A)$, or as application of the "trace" function to the matrix A. It's more commonly written without the parentheses, however.)

The trace operator has the property that for two matrices A and B such that AB is square, we have that $\mathrm{tr}AB = \mathrm{tr}BA$. (Check this yourself!) As corollaries of this, we also have, e.g.,

$$\mathrm{tr}ABC = \mathrm{tr}CAB = \mathrm{tr}BCA,$$
$$\mathrm{tr}ABCD = \mathrm{tr}DABC = \mathrm{tr}CDAB = \mathrm{tr}BCDA.$$

The following properties of the trace operator are also easily verified. Here, A and B are square matrices, and a is a real number:

$$\mathrm{tr}A = \mathrm{tr}A^T$$
$$\mathrm{tr}(A+B) = \mathrm{tr}A + \mathrm{tr}B$$
$$\mathrm{tr}\, aA = a\mathrm{tr}A$$

We now state without proof some facts of matrix derivatives (we won't need some of these until later this quarter). Equation (4) applies only to non-singular square matrices A, where $|A|$ denotes the determinant of A. We have:

$$\begin{aligned} \nabla_A \mathrm{tr} AB &= B^T && (1) \\ \nabla_{A^T} f(A) &= (\nabla_A f(A))^T && (2) \\ \nabla_A \mathrm{tr} ABA^T C &= CAB + C^T AB^T && (3) \\ \nabla_A |A| &= |A|(A^{-1})^T. && (4) \end{aligned}$$

To make our matrix notation more concrete, let us now explain in detail the meaning of the first of these equations. Suppose we have some fixed matrix $B \in \mathbb{R}^{n \times m}$. We can then define a function $f : \mathbb{R}^{m \times n} \mapsto \mathbb{R}$ according to $f(A) = \mathrm{tr} AB$. Note that this definition makes sense, because if $A \in \mathbb{R}^{m \times n}$, then AB is a square matrix, and we can apply the trace operator to it; thus, f does indeed map from $\mathbb{R}^{m \times n}$ to $\mathbb{R}$. We can then apply our definition of matrix derivatives to find $\nabla_A f(A)$, which will itself by an m-by-n matrix. Equation (1) above states that the (i, j) entry of this matrix will be given by the (i, j)-entry of B^T, or equivalently, by B_{ji}.

The proofs of Equations (1-3) are reasonably simple, and are left as an exercise to the reader. Equations (4) can be derived using the adjoint representation of the inverse of a matrix.[3]

2.2 Least squares revisited

Armed with the tools of matrix derivatives, let us now proceed to find in closed-form the value of θ that minimizes $J(\theta)$. We begin by re-writing J in matrix-vectorial notation.

Giving a training set, define the **design matrix** X to be the m-by-n matrix (actually m-by-$n + 1$, if we include the intercept term) that contains

[3] If we define A' to be the matrix whose (i, j) element is $(-1)^{i+j}$ times the determinant of the square matrix resulting from deleting row i and column j from A, then it can be proved that $A^{-1} = (A')^T/|A|$. (You can check that this is consistent with the standard way of finding A^{-1} when A is a 2-by-2 matrix. If you want to see a proof of this more general result, see an intermediate or advanced linear algebra text, such as Charles Curtis, 1991, *Linear Algebra*, Springer.) This shows that $A' = |A|(A^{-1})^T$. Also, the determinant of a matrix can be written $|A| = \sum_j A_{ij} A'_{ij}$. Since $(A')_{ij}$ does not depend on A_{ij} (as can be seen from its definition), this implies that $(\partial/\partial A_{ij})|A| = A'_{ij}$. Putting all this together shows the result.

the training examples' input values in its rows:

$$X = \begin{bmatrix} — (x^{(1)})^T — \\ — (x^{(2)})^T — \\ \vdots \\ — (x^{(m)})^T — \end{bmatrix}.$$

Also, let $\vec{y}$ be the m-dimensional vector containing all the target values from the training set:

$$\vec{y} = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{bmatrix}.$$

Now, since $h_\theta(x^{(i)}) = (x^{(i)})^T\theta$, we can easily verify that

$$\begin{aligned} X\theta - \vec{y} &= \begin{bmatrix} (x^{(1)})^T\theta \\ \vdots \\ (x^{(m)})^T\theta \end{bmatrix} - \begin{bmatrix} y^{(1)} \\ \vdots \\ y^{(m)} \end{bmatrix} \\ &= \begin{bmatrix} h_\theta(x^{(1)}) - y^{(1)} \\ \vdots \\ h_\theta(x^{(m)}) - y^{(m)} \end{bmatrix}. \end{aligned}$$

Thus, using the fact that for a vector z, we have that $z^T z = \sum_i z_i^2$:

$$\begin{aligned} \frac{1}{2}(X\theta - \vec{y})^T(X\theta - \vec{y}) &= \frac{1}{2}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2 \\ &= J(\theta) \end{aligned}$$

Finally, to minimize J, lets find its derivatives with respect to θ. Combining Equations (2) and (3), we find that

$$\nabla_{A^T} \mathrm{tr} ABA^TC = B^TA^TC^T + BA^TC \tag{5}$$

Hence,

$$\begin{aligned}
\nabla_\theta J(\theta) &= \nabla_\theta \frac{1}{2}(X\theta - \vec{y})^T(X\theta - \vec{y}) \\
&= \frac{1}{2}\nabla_\theta \left(\theta^T X^T X\theta - \theta^T X^T \vec{y} - \vec{y}^T X\theta + \vec{y}^T\vec{y}\right) \\
&= \frac{1}{2}\nabla_\theta \operatorname{tr}\left(\theta^T X^T X\theta - \theta^T X^T \vec{y} - \vec{y}^T X\theta + \vec{y}^T\vec{y}\right) \\
&= \frac{1}{2}\nabla_\theta \left(\operatorname{tr}\theta^T X^T X\theta - 2\operatorname{tr}\vec{y}^T X\theta\right) \\
&= \frac{1}{2}\left(X^T X\theta + X^T X\theta - 2X^T\vec{y}\right) \\
&= X^T X\theta - X^T\vec{y}
\end{aligned}$$

In the third step, we used the fact that the trace of a real number is just the real number; the fourth step used the fact that $\operatorname{tr}A = \operatorname{tr}A^T$, and the fifth step used Equation (5) with $A^T = \theta$, $B = B^T = X^T X$, and $C = I$, and Equation (1). To minimize J, we set its derivatives to zero, and obtain the **normal equations**:

$$X^T X\theta = X^T\vec{y}$$

Thus, the value of θ that minimizes $J(\theta)$ is given in closed form by the equation

$$\theta = (X^T X)^{-1} X^T\vec{y}.$$

3 Probabilistic interpretation

When faced with a regression problem, why might linear regression, and specifically why might the least-squares cost function J, be a reasonable choice? In this section, we will give a set of probabilistic assumptions, under which least-squares regression is derived as a very natural algorithm.

Let us assume that the target variables and the inputs are related via the equation

$$y^{(i)} = \theta^T x^{(i)} + \epsilon^{(i)},$$

where $\epsilon^{(i)}$ is an error term that captures either unmodeled effects (such as if there are some features very pertinent to predicting housing price, but that we'd left out of the regression), or random noise. Let us further assume that the $\epsilon^{(i)}$ are distributed IID (independently and identically distributed) according to a Gaussian distribution (also called a Normal distribution) with

mean zero and some variance σ^2. We can write this assumption as "$\epsilon^{(i)} \sim \mathcal{N}(0, \sigma^2)$." I.e., the density of $\epsilon^{(i)}$ is given by

$$p(\epsilon^{(i)}) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(\epsilon^{(i)})^2}{2\sigma^2}\right).$$

This implies that

$$p(y^{(i)}|x^{(i)};\theta) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(y^{(i)} - \theta^T x^{(i)})^2}{2\sigma^2}\right).$$

The notation "$p(y^{(i)}|x^{(i)};\theta)$" indicates that this is the distribution of $y^{(i)}$ given $x^{(i)}$ and parameterized by θ. Note that we should not condition on θ ("$p(y^{(i)}|x^{(i)},\theta)$"), since θ is not a random variable. We can also write the distribution of $y^{(i)}$ as as $y^{(i)} \mid x^{(i)};\theta \sim \mathcal{N}(\theta^T x^{(i)}, \sigma^2)$.

Given X (the design matrix, which contains all the $x^{(i)}$'s) and θ, what is the distribution of the $y^{(i)}$'s? The probability of the data is given by $p(\vec{y}|X;\theta)$. This quantity is typically viewed a function of $\vec{y}$ (and perhaps X), for a fixed value of θ. When we wish to explicitly view this as a function of θ, we will instead call it the **likelihood** function:

$$L(\theta) = L(\theta; X, \vec{y}) = p(\vec{y}|X;\theta).$$

Note that by the independence assumption on the $\epsilon^{(i)}$'s (and hence also the $y^{(i)}$'s given the $x^{(i)}$'s), this can also be written

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

Now, given this probabilistic model relating the $y^{(i)}$'s and the $x^{(i)}$'s, what is a reasonable way of choosing our best guess of the parameters θ? The principal of **maximum likelihood** says that we should should choose θ so as to make the data as high probability as possible. I.e., we should choose θ to maximize $L(\theta)$.

Instead of maximizing $L(\theta)$, we can also maximize any strictly increasing function of $L(\theta)$. In particular, the derivations will be a bit simpler if we

instead maximize the **log likelihood** $\ell(\theta)$:

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

Hence, maximizing $\ell(\theta)$ gives the same answer as minimizing

$$\frac{1}{2} \sum_{i=1}^{m} (y^{(i)} - \theta^T x^{(i)})^2,$$

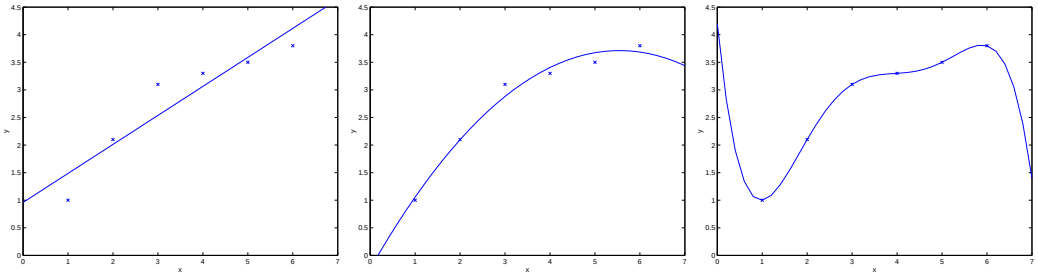
which we recognize to be $J(\theta)$, our original least-squares cost function.

To summarize: Under the previous probabilistic assumptions on the data, least-squares regression corresponds to finding the maximum likelihood estimate of θ. This is thus one set of assumptions under which least-squares regression can be justified as a very natural method that's just doing maximum likelihood estimation. (Note however that the probabilistic assumptions are by no means *necessary* for least-squares to be a perfectly good and rational procedure, and there may—and indeed there are—other natural assumptions that can also be used to justify it.)

Note also that, in our previous discussion, our final choice of θ did not depend on what was σ^2, and indeed we'd have arrived at the same result even if σ^2 were unknown. We will use this fact again later, when we talk about the exponential family and generalized linear models.

4 Locally weighted linear regression

Consider the problem of predicting y from $x \in \mathbb{R}$. The leftmost figure below shows the result of fitting a $y = \theta_0 + \theta_1 x$ to a dataset. We see that the data doesn't really lie on straight line, and so the fit is not very good.



Instead, if we had added an extra feature x^2, and fit $y = \theta_0 + \theta_1 x + \theta_2 x^2$, then we obtain a slightly better fit to the data. (See middle figure) Naively, it might seem that the more features we add, the better. However, there is also a danger in adding too many features: The rightmost figure is the result of fitting a 5-th order polynomial $y = \sum_{j=0}^{5} \theta_j x^j$. We see that even though the fitted curve passes through the data perfectly, we would not expect this to be a very good predictor of, say, housing prices (y) for different living areas (x). Without formally defining what these terms mean, we'll say the figure on the left shows an instance of **underfitting**—in which the data clearly shows structure not captured by the model—and the figure on the right is an example of **overfitting**. (Later in this class, when we talk about learning theory we'll formalize some of these notions, and also define more carefully just what it means for a hypothesis to be good or bad.)

As discussed previously, and as shown in the example above, the choice of features is important to ensuring good performance of a learning algorithm. (When we talk about model selection, we'll also see algorithms for automatically choosing a good set of features.) In this section, let us talk briefly talk about the locally weighted linear regression (LWR) algorithm which, assuming there is sufficient training data, makes the choice of features less critical. This treatment will be brief, since you'll get a chance to explore some of the properties of the LWR algorithm yourself in the homework.

In the original linear regression algorithm, to make a prediction at a query point x (i.e., to evaluate $h(x)$), we would:

1. Fit θ to minimize $\sum_i (y^{(i)} - \theta^T x^{(i)})^2$.
2. Output $\theta^T x$.

In contrast, the locally weighted linear regression algorithm does the following:

1. Fit θ to minimize $\sum_i w^{(i)} (y^{(i)} - \theta^T x^{(i)})^2$.
2. Output $\theta^T x$.

Here, the $w^{(i)}$'s are non-negative valued **weights**. Intuitively, if $w^{(i)}$ is large for a particular value of i, then in picking θ, we'll try hard to make $(y^{(i)} - \theta^T x^{(i)})^2$ small. If $w^{(i)}$ is small, then the $(y^{(i)} - \theta^T x^{(i)})^2$ error term will be pretty much ignored in the fit.

A fairly standard choice for the weights is[4]

$$w^{(i)} = \exp\left(-\frac{(x^{(i)} - x)^2}{2\tau^2}\right)$$

Note that the weights depend on the particular point x at which we're trying to evaluate x. Moreover, if $|x^{(i)} - x|$ is small, then $w^{(i)}$ is close to 1; and if $|x^{(i)} - x|$ is large, then $w^{(i)}$ is small. Hence, θ is chosen giving a much higher "weight" to the (errors on) training examples close to the query point x. (Note also that while the formula for the weights takes a form that is cosmetically similar to the density of a Gaussian distribution, the $w^{(i)}$'s do not directly have anything to do with Gaussians, and in particular the $w^{(i)}$ are not random variables, normally distributed or otherwise.) The parameter τ controls how quickly the weight of a training example falls off with distance of its $x^{(i)}$ from the query point x; τ is called the **bandwidth** parameter, and is also something that you'll get to experiment with in your homework.

Locally weighted linear regression is the first example we're seeing of a **non-parametric** algorithm. The (unweighted) linear regression algorithm that we saw earlier is known as a **parametric** learning algorithm, because it has a fixed, finite number of parameters (the θ_i's), which are fit to the data. Once we've fit the θ_i's and stored them away, we no longer need to keep the training data around to make future predictions. In contrast, to make predictions using locally weighted linear regression, we need to keep the entire training set around. The term "non-parametric" (roughly) refers to the fact that the amount of stuff we need to keep in order to represent the hypothesis h grows linearly with the size of the training set.

[4] If x is vector-valued, this is generalized to be $w^{(i)} = \exp(-(x^{(i)} - x)^T(x^{(i)} - x)/(2\tau^2))$, or $w^{(i)} = \exp(-(x^{(i)} - x)^T\Sigma^{-1}(x^{(i)} - x)/2)$, for an appropriate choice of τ or Σ.

Part II
Classification and logistic regression

Lets now talk about the classification problem. This is just like the regression problem, except that the values y we now want to predict take on only a small number of discrete values. For now, we will focus on the **binary classification** problem in which y can take on only two values, 0 and 1. (Most of what we say here will also generalize to the multiple-class case.) For instance, if we are trying to build a spam classifier for email, then $x^{(i)}$ may be some features of a piece of email, and y may be 1 if it is a piece of spam mail, and 0 otherwise. 0 is also called the **negative class**, and 1 the **positive class**, and they are sometimes also denoted by the symbols "-" and "+." Given $x^{(i)}$, the corresponding $y^{(i)}$ is also called the **label** for the training example.

5 Logistic regression

We could approach the classification problem ignoring the fact that y is discrete-valued, and use our old linear regression algorithm to try to predict y given x. However, it is easy to construct examples where this method performs very poorly. Intuitively, it also doesn't make sense for $h_\theta(x)$ to take values larger than 1 or smaller than 0 when we know that $y \in \{0, 1\}$.

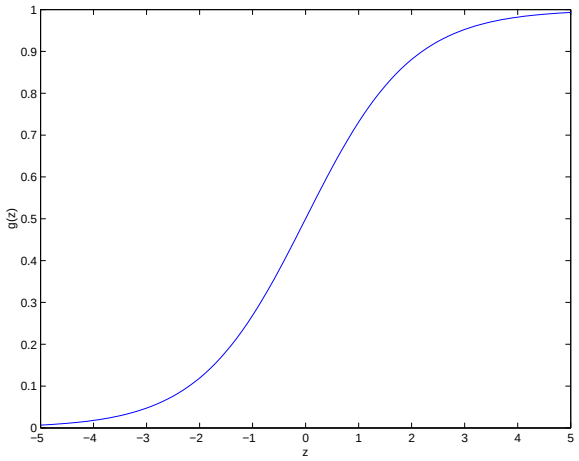
To fix this, lets change the form for our hypotheses $h_\theta(x)$. We will choose

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

where

$$g(z) = \frac{1}{1 + e^{-z}}$$

is called the **logistic function** or the **sigmoid function**. Here is a plot showing $g(z)$:



Notice that $g(z)$ tends towards 1 as $z \to \infty$, and $g(z)$ tends towards 0 as $z \to -\infty$. Moreover, g(z), and hence also $h(x)$, is always bounded between 0 and 1. As before, we are keeping the convention of letting $x_0 = 1$, so that $\theta^T x = \theta_0 + \sum_{j=1}^{n} \theta_j x_j$.

For now, lets take the choice of g as given. Other functions that smoothly increase from 0 to 1 can also be used, but for a couple of reasons that we'll see later (when we talk about GLMs, and when we talk about generative learning algorithms), the choice of the logistic function is a fairly natural one. Before moving on, here's a useful property of the derivative of the sigmoid function, which we write a g':

$$\begin{aligned} g'(z) &= \frac{d}{dz} \frac{1}{1+e^{-z}} \\ &= \frac{1}{(1+e^{-z})^2} \left(e^{-z}\right) \\ &= \frac{1}{(1+e^{-z})} \cdot \left(1 - \frac{1}{(1+e^{-z})}\right) \\ &= g(z)(1-g(z)). \end{aligned}$$

So, given the logistic regression model, how do we fit θ for it? Following how we saw least squares regression could be derived as the maximum likelihood estimator under a set of assumptions, lets endow our classification model with a set of probabilistic assumptions, and then fit the parameters via maximum likelihood.

Let us assume that

$$\begin{aligned} P(y = 1 \mid x; \theta) &= h_\theta(x) \\ P(y = 0 \mid x; \theta) &= 1 - h_\theta(x) \end{aligned}$$

Note that this can be written more compactly as

$$p(y \mid x; \theta) = (h_\theta(x))^y \, (1 - h_\theta(x))^{1-y}$$

Assuming that the m training examples were generated independently, we can then write down the likelihood of the parameters as

$$\begin{aligned} L(\theta) &= p(\vec{y} \mid X; \theta) \\ &= \prod_{i=1}^{m} p(y^{(i)} \mid x^{(i)}; \theta) \\ &= \prod_{i=1}^{m} \left(h_\theta(x^{(i)})\right)^{y^{(i)}} \left(1 - h_\theta(x^{(i)})\right)^{1-y^{(i)}} \end{aligned}$$

As before, it will be easier to maximize the log likelihood:

$$\begin{aligned} \ell(\theta) &= \log L(\theta) \\ &= \sum_{i=1}^{m} y^{(i)} \log h(x^{(i)}) + (1 - y^{(i)}) \log(1 - h(x^{(i)})) \end{aligned}$$

How do we maximize the likelihood? Similar to our derivation in the case of linear regression, we can use gradient ascent. Written in vectorial notation, our updates will therefore be given by $\theta := \theta + \alpha \nabla_\theta \ell(\theta)$. (Note the positive rather than negative sign in the update formula, since we're maximizing, rather than minimizing, a function now.) Lets start by working with just one training example (x, y), and take derivatives to derive the stochastic gradient ascent rule:

$$\begin{aligned} \frac{\partial}{\partial \theta_j} \ell(\theta) &= \left(y \frac{1}{g(\theta^T x)} - (1 - y) \frac{1}{1 - g(\theta^T x)} \right) \frac{\partial}{\partial \theta_j} g(\theta^T x) \\ &= \left(y \frac{1}{g(\theta^T x)} - (1 - y) \frac{1}{1 - g(\theta^T x)} \right) g(\theta^T x)(1 - g(\theta^T x) \frac{\partial}{\partial \theta_j} \theta^T x \\ &= \left(y(1 - g(\theta^T x)) - (1 - y) g(\theta^T x) \right) x_j \\ &= (y - h_\theta(x)) \, x_j \end{aligned}$$

Above, we used the fact that $g'(z) = g(z)(1 - g(z))$. This therefore gives us the stochastic gradient ascent rule

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

If we compare this to the LMS update rule, we see that it looks identical; but this is *not* the same algorithm, because $h_\theta(x^{(i)})$ is now defined as a non-linear function of $\theta^T x^{(i)}$. Nonetheless, it's a little surprising that we end up with the same update rule for a rather different algorithm and learning problem. Is this coincidence, or is there a deeper reason behind this? We'll answer this when get get to GLM models. (See also the extra credit problem on Q3 of problem set 1.)

6 Digression: The perceptron learning algorithm

We now digress to talk briefly about an algorithm that's of some historical interest, and that we will also return to later when we talk about learning theory. Consider modifying the logistic regression method to "force" it to output values that are either 0 or 1 or exactly. To do so, it seems natural to change the definition of g to be the threshold function:

$$g(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases}$$

If we then let $h_\theta(x) = g(\theta^T x)$ as before but using this modified definition of g, and if we use the update rule

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

then we have the **perceptron learning algorithm**.

In the 1960s, this "perceptron" was argued to be a rough model for how individual neurons in the brain work. Given how simple the algorithm is, it will also provide a starting point for our analysis when we talk about learning theory later in this class. Note however that even though the perceptron may be cosmetically similar to the other algorithms we talked about, it is actually a very different type of algorithm than logistic regression and least squares linear regression; in particular, it is difficult to endow the perceptron's predictions with meaningful probabilistic interpretations, or derive the perceptron as a maximum likelihood estimation algorithm.

7 Another algorithm for maximizing $\ell(\theta)$

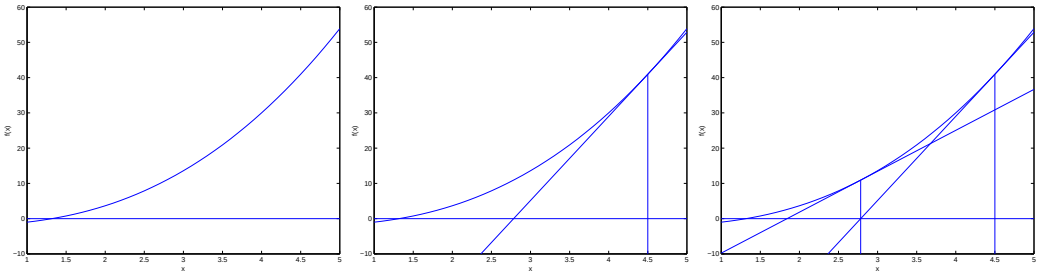
Returning to logistic regression with $g(z)$ being the sigmoid function, lets now talk about a different algorithm for minimizing $\ell(\theta)$.

To get us started, lets consider Newton's method for finding a zero of a function. Specifically, suppose we have some function $f : \mathbb{R} \mapsto \mathbb{R}$, and we wish to find a value of θ so that $f(\theta) = 0$. Here, $\theta \in \mathbb{R}$ is a real number. Newton's method performs the following update:

$$\theta := \theta - \frac{f(\theta)}{f'(\theta)}.$$

This method has a natural interpretation in which we can think of it as approximating the function f via a linear function that is tangent to f at the current guess θ, solving for where that linear function equals to zero, and letting the next guess for θ be where that linear function is zero.

Here's a picture of the Newton's method in action:



In the leftmost figure, we see the function f plotted along with the line $y = 0$. We're trying to find θ so that $f(\theta) = 0$; the value of θ that achieves this is about 1.3. Suppose we initialized the algorithm with $\theta = 4.5$. Newton's method then fits a straight line tangent to f at $\theta = 4.5$, and solves for the where that line evaluates to 0. (Middle figure.) This give us the next guess for θ, which is about 2.8. The rightmost figure shows the result of running one more iteration, which the updates θ to about 1.8. After a few more iterations, we rapidly approach $\theta = 1.3$.

Newton's method gives a way of getting to $f(\theta) = 0$. What if we want to use it to maximize some function ℓ? The maxima of ℓ correspond to points where its first derivative $\ell'(\theta)$ is zero. So, by letting $f(\theta) = \ell'(\theta)$, we can use the same algorithm to maximize ℓ, and we obtain update rule:

$$\theta := \theta - \frac{\ell'(\theta)}{\ell''(\theta)}.$$

(Something to think about: How would this change if we wanted to use Newton's method to minimize rather than maximize a function?)

Lastly, in our logistic regression setting, θ is vector-valued, so we need to generalize Newton's method to this setting. The generalization of Newton's method to this multidimensional setting (also called the Newton-Raphson method) is given by

$$\theta := \theta - H^{-1}\nabla_\theta \ell(\theta).$$

Here, $\nabla_\theta \ell(\theta)$ is, as usual, the vector of partial derivatives of $\ell(\theta)$ with respect to the θ_i's; and H is an n-by-n matrix (actually, $n+1$-by-$n+1$, assuming that we include the intercept term) called the **Hessian**, whose entries are given by

$$H_{ij} = \frac{\partial^2 \ell(\theta)}{\partial \theta_i \partial \theta_j}.$$

Newton's method typically enjoys faster convergence than (batch) gradient descent, and requires many fewer iterations to get very close to the minimum. One iteration of Newton's can, however, be more expensive than one iteration of gradient descent, since it requires finding and inverting an n-by-n Hessian; but so long as n is not too large, it is usually much faster overall. When Newton's method is applied to maximize the logistic regression log likelihood function $\ell(\theta)$, the resulting method is also called **Fisher scoring**.

Part III
Generalized Linear Models[5]

So far, we've seen a regression example, and a classification example. In the regression example, we had $y|x;\theta \sim \mathcal{N}(\mu, \sigma^2)$, and in the classification one, $y|x;\theta \sim \text{Bernoulli}(\phi)$, where for some appropriate definitions of μ and ϕ as functions of x and θ. In this section, we will show that both of these methods are special cases of a broader family of models, called Generalized Linear Models (GLMs). We will also show how other models in the GLM family can be derived and applied to other classification and regression problems.

8 The exponential family

To work our way up to GLMs, we will begin by defining exponential family distributions. We say that a class of distributions is in the exponential family if it can be written in the form

$$p(y;\eta) = b(y)\exp(\eta^T T(y) - a(\eta)) \tag{6}$$

Here, η is called the **natural parameter** (also called the **canonical parameter**) of the distribution; $T(y)$ is the **sufficient statistic** (for the distributions we consider, it will often be the case that $T(y) = y$); and $a(\eta)$ is the **log partition function**. The quantity $e^{-a(\eta)}$ essentially plays the role of a normalization constant, that makes sure the distribution $p(y;\eta)$ sums/integrates over y to 1.

A fixed choice of T, a and b defines a *family* (or set) of distributions that is parameterized by η; as we vary η, we then get different distributions within this family.

We now show that the Bernoulli and the Gaussian distributions are examples of exponential family distributions. The Bernoulli distribution with mean ϕ, written $\text{Bernoulli}(\phi)$, specifies a distribution over $y \in \{0, 1\}$, so that $p(y = 1;\phi) = \phi$; $p(y = 0;\phi) = 1 - \phi$. As we varying ϕ, we obtain Bernoulli distributions with different means. We now show that this class of Bernoulli distributions, ones obtained by varying ϕ, is in the exponential family; i.e., that there is a choice of T, a and b so that Equation (6) becomes exactly the class of Bernoulli distributions.

[5]The presentation of the material in this section takes inspiration from Michael I. Jordan, *Learning in graphical models* (unpublished book draft), and also McCullagh and Nelder, *Generalized Linear Models (2nd ed.)*.

We write the Bernoulli distribution as:

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

Thus, the natural parameter is given by $\eta = \log(\phi/(1-\phi))$. Interestingly, if we invert this definition for η by solving for ϕ in terms of η, we obtain $\phi = 1/(1+e^{-\eta})$. This is the familiar sigmoid function! This will come up again when we derive logistic regression as a GLM. To complete the formulation of the Bernoulli distribution as an exponential family distribution, we also have

$$\begin{aligned} T(y) &= y \\ a(\eta) &= -\log(1-\phi) \\ &= \log(1+e^{\eta}) \\ b(y) &= 1 \end{aligned}$$

This shows that the Bernoulli distribution can be written in the form of Equation (6), using an appropriate choice of T, a and b.

Lets now move on to consider the Gaussian distribution. Recall that, when deriving linear regression, the value of σ^2 had no effect on our final choice of θ and $h_\theta(x)$. Thus, we can choose an arbitrary value for σ^2 without changing anything. To simplify the derivation below, lets set $\sigma^2 = 1$.[6] We then have:

$$\begin{aligned} p(y;\mu) &= \frac{1}{\sqrt{2\pi}}\exp\left(-\frac{1}{2}(y-\mu)^2\right) \\ &= \frac{1}{\sqrt{2\pi}}\exp\left(-\frac{1}{2}y^2\right)\cdot\exp\left(\mu y - \frac{1}{2}\mu^2\right) \end{aligned}$$

[6] If we leave σ^2 as a variable, the Gaussian distribution can also be shown to be in the exponential family, where $\eta \in \mathbb{R}^2$ is now a 2-dimension vector that depends on both μ and σ. For the purposes of GLMs, however, the σ^2 parameter can also be treated by considering a more general definition of the exponential family: $p(y;\eta,\tau) = b(a,\tau)\exp((\eta^T T(y) - a(\eta))/c(\tau))$. Here, τ is called the **dispersion parameter**, and for the Gaussian, $c(\tau) = \sigma^2$; but given our simplification above, we won't need the more general definition for the examples we will consider here.

Thus, we see that the Gaussian is in the exponential family, with

$$\begin{aligned} \eta &= \mu \\ T(y) &= y \\ a(\eta) &= \mu^2/2 \\ &= \eta^2/2 \\ b(y) &= (1/\sqrt{2\pi}) \exp(-y^2/2). \end{aligned}$$

There're many other distributions that are members of the exponential family: The multinomial (which we'll see later), the Poisson (for modelling count-data; also see the problem set); the gamma and the exponential (for modelling continuous, non-negative random variables, such as time-intervals); the beta and the Dirichlet (for distributions over probabilities); and many more. In the next section, we will describe a general "recipe" for constructing models in which y (given x and θ) comes from any of these distributions.

9 Constructing GLMs

Suppose you would like to build a model to estimate the number y of customers arriving in your store (or number of page-views on your website) in any given hour, based on certain features x such as store promotions, recent advertising, weather, day-of-week, etc. We know that the Poisson distribution usually gives a good model for numbers of visitors. Knowing this, how can we come up with a model for our problem? Fortunately, the Poisson is an exponential family distribution, so we can apply a Generalized Linear Model (GLM). In this section, we will we will describe a method for constructing GLM models for problems such as these.

More generally, consider a classification or regression problem where we would like to predict the value of some random variable y as a function of x. To derive a GLM for this problem, we will make the following three assumptions about the conditional distribution of y given x and about our model:

1. $y \mid x; \theta \sim \text{ExponentialFamily}(\eta)$. I.e., given x and θ, the distribution of y follows some exponential family distribution, with parameter η.

2. Given x, our goal is to predict the expected value of $T(y)$ given x. In most of our examples, we will have $T(y) = y$, so this means we would like the prediction $h(x)$ output by our learned hypothesis h to

satisfy $h(x) = \mathrm{E}[y|x]$. (Note that this assumption is satisfied in the choices for $h_\theta(x)$ for both logistic regression and linear regression. For instance, in logistic regression, we had $h_\theta(x) = p(y = 1|x; \theta) = 0 \cdot p(y = 0|x; \theta) + 1 \cdot p(y = 1|x; \theta) = \mathrm{E}[y|x; \theta]$.)

3. The natural parameter η and the inputs x are related linearly: $\eta = \theta^T x$. (Or, if η is vector-valued, then $\eta_i = \theta_i^T x$.)

The third of these assumptions might seem the least well justified of the above, and it might be better thought of as a "design choice" in our recipe for designing GLMs, rather than as an assumption per se. These three assumptions/design choices will allow us to derive a very elegant class of learning algorithms, namely GLMs, that have many desirable properties such as ease of learning. Furthermore, the resulting models are often very effective for modelling different types of distributions over y; for example, we will shortly show that both logistic regression and ordinary least squares can both be derived as GLMs.

9.1 Ordinary Least Squares

To show that ordinary least squares is a special case of the GLM family of models, consider the setting where the target variable y (also called the **response variable** in GLM terminology) is continuous, and we model the conditional distribution of y given x as as a Gaussian $\mathcal{N}(\mu, \sigma^2)$. (Here, μ may depend x.) So, we let the $ExponentialFamily(\eta)$ distribution above be the Gaussian distribution. As we saw previously, in the formulation of the Gaussian as an exponential family distribution, we had $\mu = \eta$. So, we have

$$\begin{aligned} h_\theta(x) &= E[y|x; \theta] \\ &= \mu \\ &= \eta \\ &= \theta^T x. \end{aligned}$$

The first equality follows from Assumption 2, above; the second equality follows from the fact that $y|x; \theta \sim \mathcal{N}(\mu, \sigma^2)$, and so its expected value is given by μ; the third equality follows from Assumption 1 (and our earlier derivation showing that $\mu = \eta$ in the formulation of the Gaussian as an exponential family distribution); and the last equality follows from Assumption 3.

9.2 Logistic Regression

We now consider logistic regression. Here we are interested in binary classification, so $y \in \{0, 1\}$. Given that y is binary-valued, it therefore seems natural to choose the Bernoulli family of distributions to model the conditional distribution of y given x. In our formulation of the Bernoulli distribution as an exponential family distribution, we had $\phi = 1/(1 + e^{-\eta})$. Furthermore, note that if $y|x; \theta \sim \text{Bernoulli}(\phi)$, then $\text{E}[y|x; \theta] = \phi$. So, following a similar derivation as the one for ordinary least squares, we get:

$$\begin{aligned} h_\theta(x) &= E[y|x; \theta] \\ &= \phi \\ &= 1/(1 + e^{-\eta}) \\ &= 1/(1 + e^{-\theta^T x}) \end{aligned}$$

So, this gives us hypothesis functions of the form $h_\theta(x) = 1/(1 + e^{-\theta^T x})$. If you are previously wondering how we came up with the form of the logistic function $1/(1 + e^{-z})$, this gives one answer: Once we assume that y conditioned on x is Bernoulli, it arises as a consequence of the definition of GLMs and exponential family distributions.

To introduce a little more terminology, the function g giving the distribution's mean as a function of the natural parameter ($g(\eta) = \text{E}[T(y); \eta]$) is called the **canonical response function**. Its inverse, g^{-1}, is called the **canonical link function**. Thus, the canonical response function for the Gaussian family is just the identify function; and the canonical response function for the Bernoulli is the logistic function.[7]

9.3 Softmax Regression

Lets look at one more example of a GLM. Consider a classification problem in which the response variable y can take on any one of k values, so $y \in \{1\,2, \ldots, k\}$. For example, rather than classifying email into the two classes spam or not-spam—which would have been a binary classification problem—we might want to classify it into three classes, such as spam, personal mail, and work-related mail. The response variable is still discrete, but can now take on more than two values. We will thus model it as distributed according to a multinomial distribution.

[7]Many texts use g to denote the link function, and g^{-1} to denote the response function; but the notation we're using here, inherited from the early machine learning literature, will be more consistent with the notation used in the rest of the class.

Lets derive a GLM for modelling this type of multinomial data. To do so, we will begin by expressing the multinomial as an exponential family distribution.

To parameterize a multinomial over k possible outcomes, one could use k parameters $\phi_1, \ldots, \phi_k$ specifying the probability of each of the outcomes. However, these parameters would be redundant, or more formally, they would not be independent (since knowing any $k-1$ of the ϕ_i's uniquely determines the last one, as they must satisfy $\sum_{i=1}^{k} \phi_i = 1$). So, we will instead parameterize the multinomial with only $k-1$ parameters, $\phi_1, \ldots, \phi_{k-1}$, where $\phi_i = p(y = i; \phi)$, and $p(y = k; \phi) = 1 - \sum_{i=1}^{k-1} \phi_i$. For notational convenience, we will also let $\phi_k = 1 - \sum_{i=1}^{k-1} \phi_i$, but we should keep in mind that this is not a parameter, and that it is fully specified by $\phi_1, \ldots, \phi_{k-1}$.

To express the multinomial as an exponential family distribution, we will define $T(y) \in \mathbb{R}^{k-1}$ as follows:

$$T(1) = \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \; T(2) = \begin{bmatrix} 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \; T(3) = \begin{bmatrix} 0 \\ 0 \\ 1 \\ \vdots \\ 0 \end{bmatrix}, \; \cdots, T(k-1) = \begin{bmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 1 \end{bmatrix}, \; T(k) = \begin{bmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix},$$

Unlike our previous examples, here we do *not* have $T(y) = y$; also, $T(y)$ is now a $k-1$ dimensional vector, rather than a real number. We will write $(T(y))_i$ to denote the i-th element of the vector $T(y)$.

We introduce one more very useful piece of notation. An indicator function $1\{\cdot\}$ takes on a value of 1 if its argument is true, and 0 otherwise ($1\{\text{True}\} = 1$, $1\{\text{False}\} = 0$). For example, $1\{2 = 3\} = 0$, and $1\{3 = 5 - 2\} = 1$. So, we can also write the relationship between $T(y)$ and y as $(T(y))_i = 1\{y = i\}$. (Before you continue reading, please make sure you understand why this is true!) Further, we have that $\mathrm{E}[(T(y))_i] = P(y = i) = \phi_i$.

We are now ready to show that the multinomial is a member of the

exponential family. We have:

$$\begin{aligned}
p(y;\phi) &= \phi_1^{1\{y=1\}}\phi_2^{1\{y=2\}}\cdots\phi_k^{1\{y=k\}} \\
&= \phi_1^{1\{y=1\}}\phi_2^{1\{y=2\}}\cdots\phi_k^{1-\sum_{i=1}^{k-1}1\{y=i\}} \\
&= \phi_1^{(T(y))_1}\phi_2^{(T(y))_2}\cdots\phi_k^{1-\sum_{i=1}^{k-1}(T(y))_i} \\
&= \exp((T(y))_1\log(\phi_1) + (T(y))_2\log(\phi_2) + \\
&\qquad \cdots + \left(1-\textstyle\sum_{i=1}^{k-1}(T(y))_i\right)\log(\phi_k)) \\
&= \exp((T(y))_1\log(\phi_1/\phi_k) + (T(y))_2\log(\phi_2/\phi_k) + \\
&\qquad \cdots + (T(y))_{k-1}\log(\phi_{k-1}/\phi_k) + \log(\phi_k)) \\
&= b(y)\exp(\eta^T T(y) - a(\eta))
\end{aligned}$$

where

$$\begin{aligned}
\eta &= \begin{bmatrix} \log(\phi_1/\phi_k) \\ \log(\phi_2/\phi_k) \\ \vdots \\ \log(\phi_{k-1}/\phi_k) \end{bmatrix}, \\
a(\eta) &= -\log(\phi_k) \\
b(y) &= 1.
\end{aligned}$$

This completes our formulation of the multinomial as an exponential family distribution.

The link function is given (for $i = 1,\ldots,k$) by

$$\eta_i = \log\frac{\phi_i}{\phi_k}.$$

For convenience, we have also defined $\eta_k = \log(\phi_k/\phi_k) = 0$. To invert the link function and derive the response function, we therefore have that

$$\begin{aligned}
e^{\eta_i} &= \frac{\phi_i}{\phi_k} \\
\phi_k e^{\eta_i} &= \phi_i \\
\phi_k\sum_{i=1}^{k} e^{\eta_i} &= \sum_{i=1}^{k}\phi_i = 1
\end{aligned} \tag{7}$$

This implies that $\phi_k = 1/\sum_{i=1}^{k} e^{\eta_i}$, which can be substituted back into Equation (7) to give the response function

$$\phi_i = \frac{e^{\eta_i}}{\sum_{j=1}^{k} e^{\eta_j}}$$

This function mapping from the η's to the ϕ's is called the **softmax** function.

To complete our model, we use Assumption 3, given earlier, that the η_i's are linearly related to the x's. So, have $\eta_i = \theta_i^T x$ (for $i = 1, \ldots, k-1$), where $\theta_1, \ldots, \theta_{k-1} \in \mathbb{R}^{n+1}$ are the parameters of our model. For notational convenience, we can also define $\theta_k = 0$, so that $\eta_k = \theta_k^T x = 0$, as given previously. Hence, our model assumes that the conditional distribution of y given x is given by

$$\begin{aligned} p(y = i|x;\theta) &= \phi_i \\ &= \frac{e^{\eta_i}}{\sum_{j=1}^{k} e^{\eta_j}} \\ &= \frac{e^{\theta_i^T x}}{\sum_{j=1}^{k} e^{\theta_j^T x}} \end{aligned} \tag{8}$$

This model, which applies to classification problems where $y \in \{1, \ldots, k\}$, is called **softmax regression**. It is a generalization of logistic regression.

Our hypothesis will output

$$\begin{aligned} h_\theta(x) &= \mathrm{E}[T(y)|x;\theta] \\ &= \mathrm{E}\left[\left.\begin{array}{c} 1\{y=1\} \\ 1\{y=2\} \\ \vdots \\ 1\{y=k-1\} \end{array}\right| x;\theta\right] \\ &= \begin{bmatrix} \phi_1 \\ \phi_2 \\ \vdots \\ \phi_{k-1} \end{bmatrix} \\ &= \begin{bmatrix} \frac{\exp(\theta_1^T x)}{\sum_{j=1}^{k} \exp(\theta_j^T x)} \\ \frac{\exp(\theta_2^T x)}{\sum_{j=1}^{k} \exp(\theta_j^T x)} \\ \vdots \\ \frac{\exp(\theta_{k-1}^T x)}{\sum_{j=1}^{k} \exp(\theta_j^T x)} \end{bmatrix}. \end{aligned}$$

In other words, our hypothesis will output the estimated probability that $p(y = i|x;\theta)$, for every value of $i = 1, \ldots, k$. (Even though $h_\theta(x)$ as defined above is only $k-1$ dimensional, clearly $p(y = k|x;\theta)$ can be obtained as $1 - \sum_{i=1}^{k-1} \phi_i$.)

Lastly, lets discuss parameter fitting. Similar to our original derivation of ordinary least squares and logistic regression, if we have a training set of m examples $\{(x^{(i)}, y^{(i)}); i = 1, \ldots, m\}$ and would like to learn the parameters θ_i of this model, we would begin by writing down the log-likelihood

$$\begin{aligned}
\ell(\theta) &= \sum_{i=1}^{m} \log p(y^{(i)}|x^{(i)}; \theta) \\
&= \sum_{i=1}^{m} \log \prod_{l=1}^{k} \left(\frac{e^{\theta_l^T x^{(i)}}}{\sum_{j=1}^{k} e^{\theta_j^T x^{(i)}}} \right)^{1\{y^{(i)}=l\}}
\end{aligned}$$

To obtain the second line above, we used the definition for $p(y|x; \theta)$ given in Equation (8). We can now obtain the maximum likelihood estimate of the parameters by maximizing $\ell(\theta)$ in terms of θ, using a method such as gradient ascent or Newton's method.

CS229 Lecture notes

Andrew Ng

Part IV

Generative Learning algorithms

So far, we've mainly been talking about learning algorithms that model $p(y|x;\theta)$, the conditional distribution of y given x. For instance, logistic regression modeled $p(y|x;\theta)$ as $h_\theta(x) = g(\theta^T x)$ where g is the sigmoid function. In these notes, we'll talk about a different type of learning algorithm.

Consider a classification problem in which we want to learn to distinguish between elephants ($y = 1$) and dogs ($y = 0$), based on some features of an animal. Given a training set, an algorithm like logistic regression or the perceptron algorithm (basically) tries to find a straight line—that is, a decision boundary—that separates the elephants and dogs. Then, to classify a new animal as either an elephant or a dog, it checks on which side of the decision boundary it falls, and makes its prediction accordingly.

Here's a different approach. First, looking at elephants, we can build a model of what elephants look like. Then, looking at dogs, we can build a separate model of what dogs look like. Finally, to classify a new animal, we can match the new animal against the elephant model, and match it against the dog model, to see whether the new animal looks more like the elephants or more like the dogs we had seen in the training set.

Algorithms that try to learn $p(y|x)$ directly (such as logistic regression), or algorithms that try to learn mappings directly from the space of inputs $\mathcal{X}$ to the labels $\{0, 1\}$, (such as the perceptron algorithm) are called **discriminative** learning algorithms. Here, we'll talk about algorithms that instead try to model $p(x|y)$ (and $p(y)$). These algorithms are called **generative** learning algorithms. For instance, if y indicates whether a example is a dog (0) or an elephant (1), then $p(x|y = 0)$ models the distribution of dogs' features, and $p(x|y = 1)$ models the distribution of elephants' features.

After modeling $p(y)$ (called the **class priors**) and $p(x|y)$, our algorithm

can then use Bayes rule to derive the posterior distribution on y given x:

$$p(y|x) = \frac{p(x|y)p(y)}{p(x)}.$$

Here, the denominator is given by $p(x) = p(x|y = 1)p(y = 1) + p(x|y = 0)p(y = 0)$ (you should be able to verify that this is true from the standard properties of probabilities), and thus can also be expressed in terms of the quantities $p(x|y)$ and $p(y)$ that we've learned. Actually, if were calculating $p(y|x)$ in order to make a prediction, then we don't actually need to calculate the denominator, since

$$\begin{aligned} \arg\max_y p(y|x) &= \arg\max_y \frac{p(x|y)p(y)}{p(x)} \\ &= \arg\max_y p(x|y)p(y). \end{aligned}$$

1 Gaussian discriminant analysis

The first generative learning algorithm that we'll look at is Gaussian discriminant analysis (GDA). In this model, we'll assume that $p(x|y)$ is distributed according to a multivariate normal distribution. Lets talk briefly about the properties of multivariate normal distributions before moving on to the GDA model itself.

1.1 The multivariate normal distribution

The multivariate normal distribution in n-dimensions, also called the multivariate Gaussian distribution, is parameterized by a **mean vector** $\mu \in \mathbb{R}^n$ and a **covariance matrix** $\Sigma \in \mathbb{R}^{n\times n}$, where $\Sigma \geq 0$ is symmetric and positive semi-definite. Also written "$\mathcal{N}(\mu, \Sigma)$", its density is given by:

$$p(x; \mu, \Sigma) = \frac{1}{(2\pi)^{n/2}|\Sigma|^{1/2}} \exp\left(-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)\right).$$

In the equation above, "$|\Sigma|$" denotes the determinant of the matrix Σ.

For a random variable X distributed $\mathcal{N}(\mu, \Sigma)$, the mean is (unsurprisingly,) given by μ:

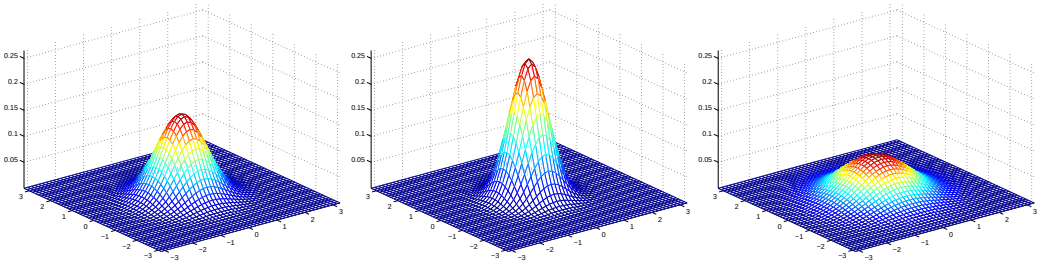
$$\mathrm{E}[X] = \int_x x\, p(x; \mu, \Sigma)dx = \mu$$

The **covariance** of a vector-valued random variable Z is defined as $\mathrm{Cov}(Z) = \mathrm{E}[(Z - \mathrm{E}[Z])(Z - \mathrm{E}[Z])^T]$. This generalizes the notion of the variance of a

real-valued random variable. The covariance can also be defined as $\mathrm{Cov}(Z) = \mathrm{E}[ZZ^T] - (\mathrm{E}[Z])(\mathrm{E}[Z])^T$. (You should be able to prove to yourself that these two definitions are equivalent.) If $X \sim \mathcal{N}(\mu, \Sigma)$, then

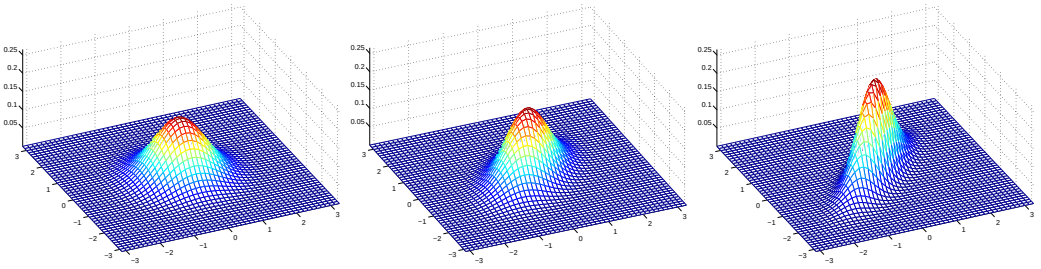
$$\mathrm{Cov}(X) = \Sigma.$$

Here're some examples of what the density of a Gaussian distribution look like:



The left-most figure shows a Gaussian with mean zero (that is, the 2x1 zero-vector) and covariance matrix $\Sigma = I$ (the 2x2 identity matrix). A Gaussian with zero mean and identity covariance is also called the **standard normal distribution**. The middle figure shows the density of a Gaussian with zero mean and $\Sigma = 0.6I$; and in the rightmost figure shows one with , $\Sigma = 2I$. We see that as Σ becomes larger, the Gaussian becomes more "spread-out," and as it becomes smaller, the distribution becomes more "compressed."

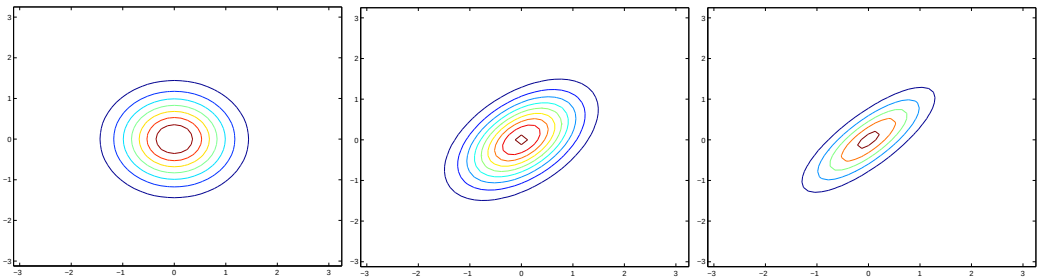
Lets look at some more examples.



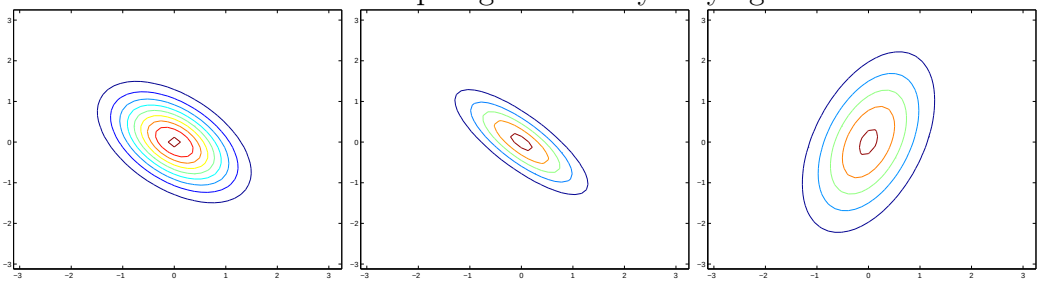
The figures above show Gaussians with mean 0, and with covariance matrices respectively

$$\Sigma = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}; \quad \Sigma = \begin{bmatrix} 1 & 0.5 \\ 0.5 & 1 \end{bmatrix}; \quad .\Sigma = \begin{bmatrix} 1 & 0.8 \\ 0.8 & 1 \end{bmatrix}.$$

The leftmost figure shows the familiar standard normal distribution, and we see that as we increase the off-diagonal entry in Σ, the density becomes more "compressed" towards the 45° line (given by $x_1 = x_2$). We can see this more clearly when we look at the contours of the same three densities:



Here's one last set of examples generated by varying Σ:

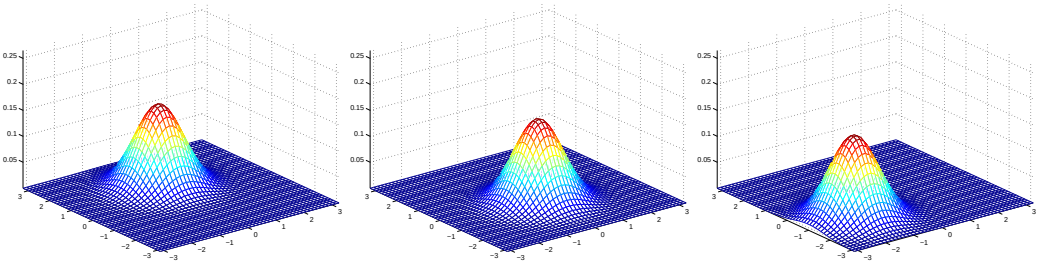


The plots above used, respectively,

$$\Sigma = \begin{bmatrix} 1 & -0.5 \\ -0.5 & 1 \end{bmatrix}; \quad \Sigma = \begin{bmatrix} 1 & -0.8 \\ -0.8 & 1 \end{bmatrix}; \quad .\Sigma = \begin{bmatrix} 3 & 0.8 \\ 0.8 & 1 \end{bmatrix}.$$

From the leftmost and middle figures, we see that by decreasing the diagonal elements of the covariance matrix, the density now becomes "compressed" again, but in the opposite direction. Lastly, as we vary the parameters, more generally the contours will form ellipses (the rightmost figure showing an example).

As our last set of examples, fixing $\Sigma = I$, by varying μ, we can also move the mean of the density around.



The figures above were generated using $\Sigma = I$, and respectively

$$\mu = \begin{bmatrix} 1 \\ 0 \end{bmatrix}; \quad \mu = \begin{bmatrix} -0.5 \\ 0 \end{bmatrix}; \quad \mu = \begin{bmatrix} -1 \\ -1.5 \end{bmatrix}.$$

1.2 The Gaussian Discriminant Analysis model

When we have a classification problem in which the input features x are continuous-valued random variables, we can then use the Gaussian Discriminant Analysis (GDA) model, which models $p(x|y)$ using a multivariate normal distribution. The model is:

$$\begin{aligned} y &\sim \text{Bernoulli}(\phi) \\ x|y=0 &\sim \mathcal{N}(\mu_0, \Sigma) \\ x|y=1 &\sim \mathcal{N}(\mu_1, \Sigma) \end{aligned}$$

Writing out the distributions, this is:

$$\begin{aligned} p(y) &= \phi^y(1-\phi)^{1-y} \\ p(x|y=0) &= \frac{1}{(2\pi)^{n/2}|\Sigma|^{1/2}} \exp\left(-\frac{1}{2}(x-\mu_0)^T\Sigma^{-1}(x-\mu_0)\right) \\ p(x|y=1) &= \frac{1}{(2\pi)^{n/2}|\Sigma|^{1/2}} \exp\left(-\frac{1}{2}(x-\mu_1)^T\Sigma^{-1}(x-\mu_1)\right) \end{aligned}$$

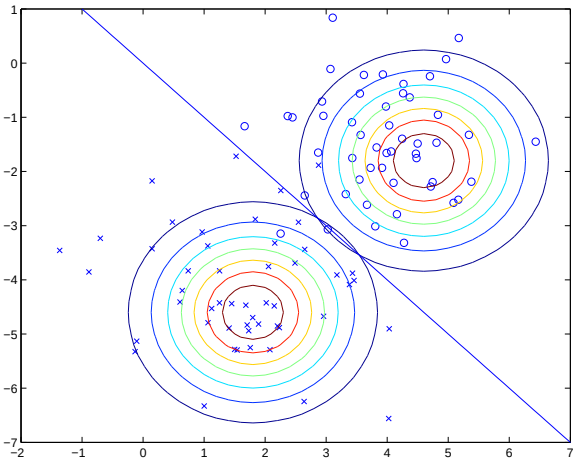
Here, the parameters of our model are ϕ, Σ, μ_0 and μ_1. (Note that while there're two different mean vectors μ_0 and μ_1, this model is usually applied using only one covariance matrix Σ.) The log-likelihood of the data is given by

$$\begin{aligned} \ell(\phi, \mu_0, \mu_1, \Sigma) &= \log \prod_{i=1}^{m} p(x^{(i)}, y^{(i)}; \phi, \mu_0, \mu_1, \Sigma) \\ &= \log \prod_{i=1}^{m} p(x^{(i)}|y^{(i)}; \mu_0, \mu_1, \Sigma) p(y^{(i)}; \phi). \end{aligned}$$

By maximizing ℓ with respect to the parameters, we find the maximum likelihood estimate of the parameters (see problem set 1) to be:

$$\begin{aligned}
\phi &= \frac{1}{m}\sum_{i=1}^{m} 1\{y^{(i)} = 1\} \\
\mu_0 &= \frac{\sum_{i=1}^{m} 1\{y^{(i)} = 0\} x^{(i)}}{\sum_{i=1}^{m} 1\{y^{(i)} = 0\}} \\
\mu_1 &= \frac{\sum_{i=1}^{m} 1\{y^{(i)} = 1\} x^{(i)}}{\sum_{i=1}^{m} 1\{y^{(i)} = 1\}} \\
\Sigma &= \frac{1}{m}\sum_{i=1}^{m} (x^{(i)} - \mu_{y^{(i)}})(x^{(i)} - \mu_{y^{(i)}})^T.
\end{aligned}$$

Pictorially, what the algorithm is doing can be seen in as follows:



Shown in the figure are the training set, as well as the contours of the two Gaussian distributions that have been fit to the data in each of the two classes. Note that the two Gaussians have contours that are the same shape and orientation, since they share a covariance matrix Σ, but they have different means μ_0 and μ_1. Also shown in the figure is the straight line giving the decision boundary at which $p(y = 1|x) = 0.5$. On one side of the boundary, we'll predict $y = 1$ to be the most likely outcome, and on the other side, we'll predict $y = 0$.

1.3 Discussion: GDA and logistic regression

The GDA model has an interesting relationship to logistic regression. If we view the quantity $p(y = 1|x; \phi, \mu_0, \mu_1, \Sigma)$ as a function of x, we'll find that it

can be expressed in the form

$$p(y = 1|x; \phi, \Sigma, \mu_0, \mu_1) = \frac{1}{1 + \exp(-\theta^T x)},$$

where θ is some appropriate function of $\phi, \Sigma, \mu_0, \mu_1$.[1] This is exactly the form that logistic regression—a discriminative algorithm—used to model $p(y = 1|x)$.

When would we prefer one model over another? GDA and logistic regression will, in general, give different decision boundaries when trained on the same dataset. Which is better?

We just argued that if $p(x|y)$ is multivariate gaussian (with shared Σ), then $p(y|x)$ necessarily follows a logistic function. The converse, however, is not true; i.e., $p(y|x)$ being a logistic function does not imply $p(x|y)$ is multivariate gaussian. This shows that GDA makes *stronger* modeling assumptions about the data than does logistic regression. It turns out that when these modeling assumptions are correct, then GDA will find better fits to the data, and is a better model. Specifically, when $p(x|y)$ is indeed gaussian (with shared Σ), then GDA is **asymptotically efficient**. Informally, this means that in the limit of very large training sets (large m), there is no algorithm that is strictly better than GDA (in terms of, say, how accurately they estimate $p(y|x)$). In particular, it can be shown that in this setting, GDA will be a better algorithm than logistic regression; and more generally, even for small training set sizes, we would generally expect GDA to better.

In contrast, by making significantly weaker assumptions, logistic regression is also more *robust* and less sensitive to incorrect modeling assumptions. There are many different sets of assumptions that would lead to $p(y|x)$ taking the form of a logistic function. For example, if $x|y = 0 \sim \text{Poisson}(\lambda_0)$, and $x|y = 1 \sim \text{Poisson}(\lambda_1)$, then $p(y|x)$ will be logistic. Logistic regression will also work well on Poisson data like this. But if we were to use GDA on such data—and fit Gaussian distributions to such non-Gaussian data—then the results will be less predictable, and GDA may (or may not) do well.

To summarize: GDA makes stronger modeling assumptions, and is more data efficient (i.e., requires less training data to learn "well") when the modeling assumptions are correct or at least approximately correct. Logistic regression makes weaker assumptions, and is significantly more robust to deviations from modeling assumptions. Specifically, when the data is indeed non-Gaussian, then in the limit of large datasets, logistic regression will

[1] This uses the convention of redefining the $x^{(i)}$'s on the right-hand-side to be $n + 1$-dimensional vectors by adding the extra coordinate $x_0^{(i)} = 1$; see problem set 1.

almost always do better than GDA. For this reason, in practice logistic regression is used more often than GDA. (Some related considerations about discriminative vs. generative models also apply for the Naive Bayes algorithm that we discuss next, but the Naive Bayes algorithm is still considered a very good, and is certainly also a very popular, classification algorithm.)

2 Naive Bayes

In GDA, the feature vectors x were continuous, real-valued vectors. Lets now talk about a different learning algorithm in which the x_i's are discrete-valued.

For our motivating example, consider building an email spam filter using machine learning. Here, we wish to classify messages according to whether they are unsolicited commercial (spam) email, or non-spam email. After learning to do this, we can then have our mail reader automatically filter out the spam messages and perhaps place them in a separate mail folder. Classifying emails is one example of a broader set of problems called **text classification**.

Lets say we have a training set (a set of emails labeled as spam or non-spam). We'll begin our construction of our spam filter by specifying the features x_i used to represent an email.

We will represent an email via a feature vector whose length is equal to the number of words in the dictionary. Specifically, if an email contains the i-th word of the dictionary, then we will set $x_i = 1$; otherwise, we let $x_i = 0$. For instance, the vector

$$x = \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{bmatrix} \begin{matrix} \text{a} \\ \text{aardvark} \\ \text{aardwolf} \\ \vdots \\ \text{buy} \\ \vdots \\ \text{zygmurgy} \end{matrix}$$

is used to represent an email that contains the words "a" and "buy," but not "aardvark," "aardwolf" or "zygmurgy."[2] The set of words encoded into the

[2]Actually, rather than looking through an english dictionary for the list of all english words, in practice it is more common to look through our training set and encode in our feature vector only the words that occur at least once there. Apart from reducing the number of words modeled and hence reducing our computational and space requirements,

feature vector is called the **vocabulary**, so the dimension of x is equal to the size of the vocabulary.

Having chosen our feature vector, we now want to build a discriminative model. So, we have to model $p(x|y)$. But if we have, say, a vocabulary of 50000 words, then $x \in \{0,1\}^{50000}$ (x is a 50000-dimensional vector of 0's and 1's), and if we were to model x explicitly with a multinomial distribution over the 2^{50000} possible outcomes, then we'd end up with a $(2^{50000}-1)$-dimensional parameter vector. This is clearly too many parameters.

To model $p(x|y)$, we will therefore make a very strong assumption. We will assume that the x_i's are conditionally independent given y. This assumption is called the **Naive Bayes (NB) assumption**, and the resulting algorithm is called the **Naive Bayes classifier**. For instance, if $y = 1$ means spam email; "buy" is word 2087 and "price" is word 39831; then we are assuming that if I tell you $y = 1$ (that a particular piece of email is spam), then knowledge of x_{2087} (knowledge of whether "buy" appears in the message) will have no effect on your beliefs about the value of x_{39831} (whether "price" appears). More formally, this can be written $p(x_{2087}|y) = p(x_{2087}|y, x_{39831})$. (Note that this is *not* the same as saying that x_{2087} and x_{39831} are independent, which would have been written "$p(x_{2087}) = p(x_{2087}|x_{39831})$"; rather, we are only assuming that x_{2087} and x_{39831} are conditionally independent *given* y.)

We now have:

$$\begin{aligned}
&p(x_1, \ldots, x_{50000}|y) \\
&\quad = p(x_1|y)p(x_2|y, x_1)p(x_3|y, x_1, x_2)\cdots p(x_{50000}|y, x_1, \ldots, x_{49999}) \\
&\quad = p(x_1|y)p(x_2|y)p(x_3|y)\cdots p(x_{50000}|y) \\
&\quad = \prod_{i=1}^{n} p(x_i|y)
\end{aligned}$$

The first equality simply follows from the usual properties of probabilities, and the second equality used the NB assumption. We note that even though the Naive Bayes assumption is an extremely strong assumptions, the resulting algorithm works well on many problems.

Our model is parameterized by $\phi_{i|y=1} = p(x_i = 1|y = 1)$, $\phi_{i|y=0} = p(x_i = 1|y = 0)$, and $\phi_y = p(y = 1)$. As usual, given a training set $\{(x^{(i)}, y^{(i)}); i =$

this also has the advantage of allowing us to model/include as a feature many words that may appear in your email (such as "cs229") but that you won't find in a dictionary. Sometimes (as in the homework), we also exclude the very high frequency words (which will be words like "the," "of," "and,"; these high frequency, "content free" words are called **stop words**) since they occur in so many documents and do little to indicate whether an email is spam or non-spam.

$1, \ldots, m\}$, we can write down the joint likelihood of the data:

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

Maximizing this with respect to $\phi_y, \phi_{i|y=0}$ and $\phi_{i|y=1}$ gives the maximum likelihood estimates:

$$\begin{aligned}
\phi_{j|y=1} &= \frac{\sum_{i=1}^{m} 1\{x_j^{(i)} = 1 \wedge y^{(i)} = 1\}}{\sum_{i=1}^{m} 1\{y^{(i)} = 1\}} \\
\phi_{j|y=0} &= \frac{\sum_{i=1}^{m} 1\{x_j^{(i)} = 1 \wedge y^{(i)} = 0\}}{\sum_{i=1}^{m} 1\{y^{(i)} = 0\}} \\
\phi_y &= \frac{\sum_{i=1}^{m} 1\{y^{(i)} = 1\}}{m}
\end{aligned}$$

In the equations above, the "$\wedge$" symbol means "and." The parameters have a very natural interpretation. For instance, $\phi_{j|y=1}$ is just the fraction of the spam ($y = 1$) emails in which word j does appear.

Having fit all these parameters, to make a prediction on a new example with features x, we then simply calculate

$$\begin{aligned}
p(y = 1|x) &= \frac{p(x|y = 1)p(y = 1)}{p(x)} \\
&= \frac{(\prod_{i=1}^{n} p(x_i|y = 1))\, p(y = 1)}{(\prod_{i=1}^{n} p(x_i|y = 1))\, p(y = 1) + (\prod_{i=1}^{n} p(x_i|y = 0))\, p(y = 0)},
\end{aligned}$$

and pick whichever class has the higher posterior probability.

Lastly, we note that while we have developed the Naive Bayes algorithm mainly for the case of problems where the features x_i are binary-valued, the generalization to where x_i can take values in $\{1, 2, \ldots, k_i\}$ is straightforward. Here, we would simply model $p(x_i|y)$ as multinomial rather than as Bernoulli. Indeed, even if some original input attribute (say, the living area of a house, as in our earlier example) were continuous valued, it is quite common to **discretize** it—that is, turn it into a small set of discrete values—and apply Naive Bayes. For instance, if we use some feature x_i to represent living area, we might discretize the continuous values as follows:

Living area (sq. feet)	< 400	400-800	800-1200	1200-1600	>1600
x_i	1	2	3	4	5

Thus, for a house with living area 890 square feet, we would set the value of the corresponding feature x_i to 3. We can then apply the Naive Bayes

algorithm, and model $p(x_i|y)$ with a multinomial distribution, as described previously. When the original, continuous-valued attributes are not well-modeled by a multivariate normal distribution, discretizing the features and using Naive Bayes (instead of GDA) will often result in a better classifier.

2.1 Laplace smoothing

The Naive Bayes algorithm as we have described it will work fairly well for many problems, but there is a simple change that makes it work much better, especially for text classification. Lets briefly discuss a problem with the algorithm in its current form, and then talk about how we can fix it.

Consider spam/email classification, and lets suppose that, after completing CS229 and having done excellent work on the project, you decide around June 2003 to submit the work you did to the NIPS conference for publication. (NIPS is one of the top machine learning conferences, and the deadline for submitting a paper is typically in late June or early July.) Because you end up discussing the conference in your emails, you also start getting messages with the word "nips" in it. But this is your first NIPS paper, and until this time, you had not previously seen any emails containing the word "nips"; in particular "nips" did not ever appear in your training set of spam/non-spam emails. Assuming that "nips" was the 35000th word in the dictionary, your Naive Bayes spam filter therefore had picked its maximum likelihood estimates of the parameters $\phi_{35000|y}$ to be

$$\begin{aligned}
\phi_{35000|y=1} &= \frac{\sum_{i=1}^{m} 1\{x_{35000}^{(i)} = 1 \wedge y^{(i)} = 1\}}{\sum_{i=1}^{m} 1\{y^{(i)} = 1\}} = 0 \\
\phi_{35000|y=0} &= \frac{\sum_{i=1}^{m} 1\{x_{35000}^{(i)} = 1 \wedge y^{(i)} = 0\}}{\sum_{i=1}^{m} 1\{y^{(i)} = 0\}} = 0
\end{aligned}$$

I.e., because it has never seen "nips" before in either spam or non-spam training examples, it thinks the probability of seeing it in either type of email is zero. Hence, when trying to decide if one of these messages containing "nips" is spam, it calculates the class posterior probabilities, and obtains

$$\begin{aligned}
p(y = 1|x) &= \frac{\prod_{i=1}^{n} p(x_i|y = 1)p(y = 1)}{\prod_{i=1}^{n} p(x_i|y = 1)p(y = 1) + \prod_{i=1}^{n} p(x_i|y = 0)p(y = 0)} \\
&= \frac{0}{0}.
\end{aligned}$$

This is because each of the terms "$\prod_{i=1}^{n} p(x_i|y)$" includes a term $p(x_{35000}|y) = 0$ that is multiplied into it. Hence, our algorithm obtains 0/0, and doesn't know how to make a prediction.

Stating the problem more broadly, it is statistically a bad idea to estimate the probability of some event to be zero just because you haven't seen it before in your finite training set. Take the problem of estimating the mean of a multinomial random variable z taking values in $\{1, \ldots, k\}$. We can parameterize our multinomial with $\phi_i = p(z = i)$. Given a set of m independent observations $\{z^{(1)}, \ldots, z^{(m)}\}$, the maximum likelihood estimates are given by

$$\phi_j = \frac{\sum_{i=1}^{m} 1\{z^{(i)} = j\}}{m}.$$

As we saw previously, if we were to use these maximum likelihood estimates, then some of the ϕ_j's might end up as zero, which was a problem. To avoid this, we can use **Laplace smoothing**, which replaces the above estimate with

$$\phi_j = \frac{\sum_{i=1}^{m} 1\{z^{(i)} = j\} + 1}{m + k}.$$

Here, we've added 1 to the numerator, and k to the denominator. Note that $\sum_{j=1}^{k} \phi_j = 1$ still holds (check this yourself!), which is a desirable property since the ϕ_j's are estimates for probabilities that we know must sum to 1. Also, $\phi_j \neq 0$ for all values of j, solving our problem of probabilities being estimated as zero. Under certain (arguably quite strong) conditions, it can be shown that the Laplace smoothing actually gives the optimal estimator of the ϕ_j's.

Returning to our Naive Bayes classifier, with Laplace smoothing, we therefore obtain the following estimates of the parameters:

$$\begin{aligned}
\phi_{j|y=1} &= \frac{\sum_{i=1}^{m} 1\{x_j^{(i)} = 1 \wedge y^{(i)} = 1\} + 1}{\sum_{i=1}^{m} 1\{y^{(i)} = 1\} + 2} \\
\phi_{j|y=0} &= \frac{\sum_{i=1}^{m} 1\{x_j^{(i)} = 1 \wedge y^{(i)} = 0\} + 1}{\sum_{i=1}^{m} 1\{y^{(i)} = 0\} + 2}
\end{aligned}$$

(In practice, it usually doesn't matter much whether we apply Laplace smoothing to ϕ_y or not, since we will typically have a fair fraction each of spam and non-spam messages, so ϕ_y will be a reasonable estimate of $p(y = 1)$ and will be quite far from 0 anyway.)

2.2 Event models for text classification

To close off our discussion of generative learning algorithms, lets talk about one more model that is specifically for text classification. While Naive Bayes

as we've presented it will work well for many classification problems, for text classification, there is a related model that does even better.

In the specific context of text classification, Naive Bayes as presented uses the what's called the **multi-variate Bernoulli event model**. In this model, we assumed that the way an email is generated is that first it is randomly determined (according to the class priors $p(y)$) whether a spammer or non-spammer will send you your next message. Then, the person sending the email runs through the dictionary, deciding whether to include each word i in that email independently and according to the probabilities $p(x_i = 1|y) = \phi_{i|y}$. Thus, the probability of a message was given by $p(y) \prod_{i=1}^{n} p(x_i|y)$.

Here's a different model, called the **multinomial event model**. To describe this model, we will use a different notation and set of features for representing emails. We let x_i denote the identity of the i-th word in the email. Thus, x_i is now an integer taking values in $\{1, \ldots, |V|\}$, where $|V|$ is the size of our vocabulary (dictionary). An email of n words is now represented by a vector $(x_1, x_2, \ldots, x_n)$ of length n; note that n can vary for different documents. For instance, if an email starts with "A NIPS ...," then $x_1 = 1$ ("a" is the first word in the dictionary), and $x_2 = 35000$ (if "nips" is the 35000th word in the dictionary).

In the multinomial event model, we assume that the way an email is generated is via a random process in which spam/non-spam is first determined (according to $p(y)$) as before. Then, the sender of the email writes the email by first generating x_1 from some multinomial distribution over words ($p(x_1|y)$). Next, the second word x_2 is chosen independently of x_1 but from the same multinomial distribution, and similarly for x_3, x_4, and so on, until all n words of the email have been generated. Thus, the overall probability of a message is given by $p(y) \prod_{i=1}^{n} p(x_i|y)$. Note that this formula looks like the one we had earlier for the probability of a message under the multi-variate Bernoulli event model, but that the terms in the formula now mean very different things. In particular $x_i|y$ is now a multinomial, rather than a Bernoulli distribution.

The parameters for our new model are $\phi_y = p(y)$ as before, $\phi_{i|y=1} = p(x_j = i|y = 1)$ (for any j) and $\phi_{i|y=0} = p(x_j = i|y = 0)$. Note that we have assumed that $p(x_j|y)$ is the same for all values of j (i.e., that the distribution according to which a word is generated does not depend on its position j within the email).

If we are given a training set $\{(x^{(i)}, y^{(i)}); i = 1, \ldots, m\}$ where $x^{(i)} = (x_1^{(i)}, x_2^{(i)}, \ldots, x_{n_i}^{(i)})$ (here, n_i is the number of words in the i-training example),

the likelihood of the data is given by

$$\begin{aligned}\mathcal{L}(\phi, \phi_{i|y=0}, \phi_{i|y=1}) &= \prod_{i=1}^{m} p(x^{(i)}, y^{(i)}) \\ &= \prod_{i=1}^{m} \left(\prod_{j=1}^{n_i} p(x_j^{(i)}|y; \phi_{i|y=0}, \phi_{i|y=1}) \right) p(y^{(i)}; \phi_y).\end{aligned}$$

Maximizing this yields the maximum likelihood estimates of the parameters:

$$\begin{aligned}\phi_{k|y=1} &= \frac{\sum_{i=1}^{m} \sum_{j=1}^{n_i} 1\{x_j^{(i)} = k \wedge y^{(i)} = 1\}}{\sum_{i=1}^{m} 1\{y^{(i)} = 1\} n_i} \\ \phi_{k|y=0} &= \frac{\sum_{i=1}^{m} \sum_{j=1}^{n_i} 1\{x_j^{(i)} = k \wedge y^{(i)} = 0\}}{\sum_{i=1}^{m} 1\{y^{(i)} = 0\} n_i} \\ \phi_y &= \frac{\sum_{i=1}^{m} 1\{y^{(i)} = 1\}}{m}.\end{aligned}$$

If we were to apply Laplace smoothing (which needed in practice for good performance) when estimating $\phi_{k|y=0}$ and $\phi_{k|y=1}$, we add 1 to the numerators and $|V|$ to the denominators, and obtain:

$$\begin{aligned}\phi_{k|y=1} &= \frac{\sum_{i=1}^{m} \sum_{j=1}^{n_i} 1\{x_j^{(i)} = k \wedge y^{(i)} = 1\} + 1}{\sum_{i=1}^{m} 1\{y^{(i)} = 1\} n_i + |V|} \\ \phi_{k|y=0} &= \frac{\sum_{i=1}^{m} \sum_{j=1}^{n_i} 1\{x_j^{(i)} = k \wedge y^{(i)} = 0\} + 1}{\sum_{i=1}^{m} 1\{y^{(i)} = 0\} n_i + |V|}.\end{aligned}$$

While not necessarily the very best classification algorithm, the Naive Bayes classifier often works surprisingly well. It is often also a very good "first thing to try," given its simplicity and ease of implementation.

CS229 Lecture notes

Andrew Ng

Part V
Support Vector Machines

This set of notes presents the Support Vector Machine (SVM) learning algorithm. SVMs are among the best (and many believe is indeed the best) "off-the-shelf" supervised learning algorithm. To tell the SVM story, we'll need to first talk about margins and the idea of separating data with a large "gap." Next, we'll talk about the optimal margin classifier, which will lead us into a digression on Lagrange duality. We'll also see kernels, which give a way to apply SVMs efficiently in very high dimensional (such as infinite-dimensional) feature spaces, and finally, we'll close off the story with the SMO algorithm, which gives an efficient implementation of SVMs.

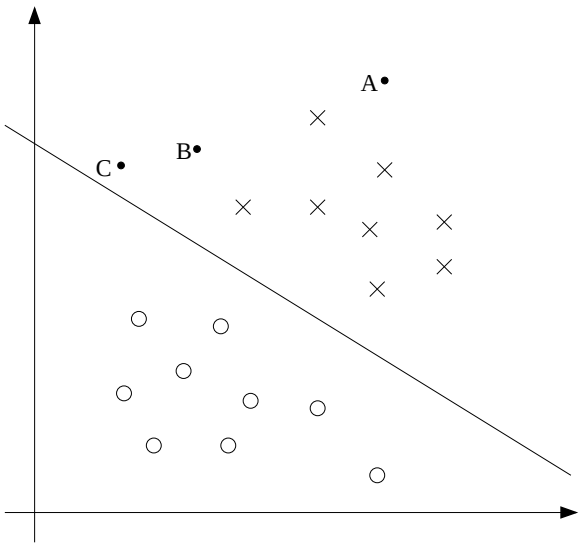
1 Margins: Intuition

We'll start our story on SVMs by talking about margins. This section will give the intuitions about margins and about the "confidence" of our predictions; these ideas will be made formal in Section 3.

Consider logistic regression, where the probability $p(y = 1|x; \theta)$ is modeled by $h_\theta(x) = g(\theta^T x)$. We would then predict "1" on an input x if and only if $h_\theta(x) \geq 0.5$, or equivalently, if and only if $\theta^T x \geq 0$. Consider a positive training example ($y = 1$). The larger $\theta^T x$ is, the larger also is $h_\theta(x) = p(y = 1|x; w, b)$, and thus also the higher our degree of "confidence" that the label is 1. Thus, informally we can think of our prediction as being a very confident one that $y = 1$ if $\theta^T x \gg 0$. Similarly, we think of logistic regression as making a very confident prediction of $y = 0$, if $\theta^T x \ll 0$. Given a training set, again informally it seems that we'd have found a good fit to the training data if we can find θ so that $\theta^T x^{(i)} \gg 0$ whenever $y^{(i)} = 1$, and

$\theta^T x^{(i)} \ll 0$ whenever $y^{(i)} = 0$, since this would reflect a very confident (and correct) set of classifications for all the training examples. This seems to be a nice goal to aim for, and we'll soon formalize this idea using the notion of functional margins.

For a different type of intuition, consider the following figure, in which x's represent positive training examples, o's denote negative training examples, a decision boundary (this is the line given by the equation $\theta^T x = 0$, and is also called the **separating hyperplane**) is also shown, and three points have also been labeled A, B and C.



Notice that the point A is very far from the decision boundary. If we are asked to make a prediction for the value of y at at A, it seems we should be quite confident that $y = 1$ there. Conversely, the point C is very close to the decision boundary, and while it's on the side of the decision boundary on which we would predict $y = 1$, it seems likely that just a small change to the decision boundary could easily have caused out prediction to be $y = 0$. Hence, we're much more confident about our prediction at A than at C. The point B lies in-between these two cases, and more broadly, we see that if a point is far from the separating hyperplane, then we may be significantly more confident in our predictions. Again, informally we think it'd be nice if, given a training set, we manage to find a decision boundary that allows us to make all correct and confident (meaning far from the decision boundary) predictions on the training examples. We'll formalize this later using the notion of geometric margins.

2 Notation

To make our discussion of SVMs easier, we'll first need to introduce a new notation for talking about classification. We will be considering a linear classifier for a binary classification problem with labels y and features x. From now, we'll use $y \in \{-1, 1\}$ (instead of $\{0, 1\}$) to denote the class labels. Also, rather than parameterizing our linear classifier with the vector θ, we will use parameters w, b, and write our classifier as

$$h_{w,b}(x) = g(w^T x + b).$$

Here, $g(z) = 1$ if $z \geq 0$, and $g(z) = -1$ otherwise. This "w, b" notation allows us to explicitly treat the intercept term b separately from the other parameters. (We also drop the convention we had previously of letting $x_0 = 1$ be an extra coordinate in the input feature vector.) Thus, b takes the role of what was previously θ_0, and w takes the role of $[\theta_1 \ldots \theta_n]^T$.

Note also that, from our definition of g above, our classifier will directly predict either 1 or -1 (cf. the perceptron algorithm), without first going through the intermediate step of estimating the probability of y being 1 (which was what logistic regression did).

3 Functional and geometric margins

Lets formalize the notions of the functional and geometric margins. Given a training example $(x^{(i)}, y^{(i)})$, we define the **functional margin** of (w, b) with respect to the training example

$$\hat{\gamma}^{(i)} = y^{(i)}(w^T x + b).$$

Note that if $y^{(i)} = 1$, then for the functional margin to be large (i.e., for our prediction to be confident and correct), then we need $w^T x + b$ to be a large positive number. Conversely, if $y^{(i)} = -1$, then for the functional margin to be large, then we need $w^T x + b$ to be a large negative number. Moreover, if $y^{(i)}(w^T x + b) > 0$, then our prediction on this example is correct. (Check this yourself.) Hence, a large functional margin represents a confident and a correct prediction.

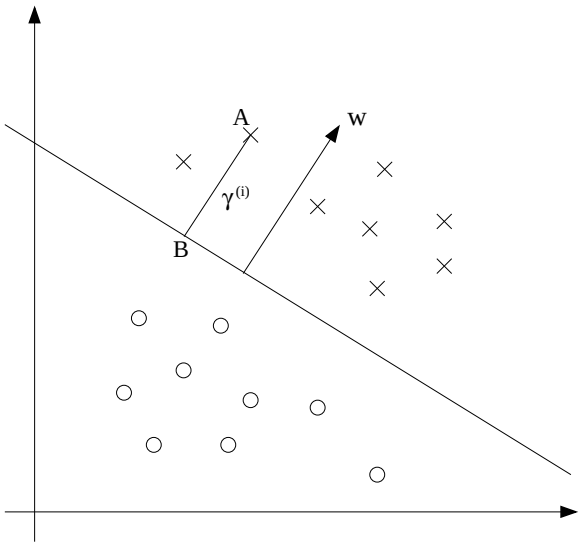
For a linear classifier with the choice of g given above (taking values in $\{-1, 1\}$), there's one property of the functional margin that makes it not a very good measure of confidence, however. Given our choice of g, we note that if we replace w with $2w$ and b with $2b$, then since $g(w^T x + b) = g(2w^T x + 2b)$,

this would not change $h_{w,b}(x)$ at all. I.e., g, and hence also $h_{w,b}(x)$, depends only on the sign, but not on the magnitude, of $w^T x + b$. However, replacing (w, b) with $(2w, 2b)$ also results in multiplying our functional margin by a factor of 2. Thus, it seems that by exploiting our freedom to scale w and b, we can make the functional margin arbitrarily large without really changing anything meaningful. Intuitively, it might therefore make sense to impose some sort of normalization condition such as that $||w||_2 = 1$; i.e., we might replace (w, b) with $(w/||w||_2, b/||w||_2)$, and instead consider the functional margin of $(w/||w||_2, b/||w||_2)$. We'll come back to this later.

Given a training set $S = \{(x^{(i)}, y^{(i)}); i = 1, \ldots, m\}$, we also define the function margin of (w, b) with respect to S as the smallest of the functional margins of the individual training examples. Denoted by $\hat{\gamma}$, this can therefore be written:

$$\hat{\gamma} = \min_{i=1,\ldots,m} \hat{\gamma}^{(i)}.$$

Next, lets talk about **geometric margins**. Consider the picture below:



The decision boundary corresponding to (w, b) is shown, along with the vector w. Note that w is orthogonal (at 90°) to the separating hyperplane. (You should convince yourself that this must be the case.) Consider the point at A, which represents the input $x^{(i)}$ of some training example with label $y^{(i)} = 1$. Its distance to the decision boundary, $\gamma^{(i)}$, is given by the line segment AB.

How can we find the value of $\gamma^{(i)}$? Well, $w/||w||$ is a unit-length vector pointing in the same direction as w. Since A represents $x^{(i)}$, we therefore

find that the point B is given by $x^{(i)} - \gamma^{(i)} \cdot w/||w||$. But this point lies on the decision boundary, and all points x on the decision boundary satisfy the equation $w^T x + b = 0$. Hence,

$$w^T \left(x^{(i)} - \gamma^{(i)} \frac{w}{||w||} \right) + b = 0.$$

Solving for $\gamma^{(i)}$ yields

$$\gamma^{(i)} = \frac{w^T x^{(i)} + b}{||w||} = \left(\frac{w}{||w||} \right)^T x^{(i)} + \frac{b}{||w||}.$$

This was worked out for the case of a positive training example at A in the figure, where being on the "positive" side of the decision boundary is good. More generally, we define the geometric margin of (w, b) with respect to a training example $(x^{(i)}, y^{(i)})$ to be

$$\gamma^{(i)} = y^{(i)} \left(\left(\frac{w}{||w||} \right)^T x^{(i)} + \frac{b}{||w||} \right).$$

Note that if $||w|| = 1$, then the functional margin equals the geometric margin—this thus gives us a way of relating these two different notions of margin. Also, the geometric margin is invariant to rescaling of the parameters; i.e., if we replace w with $2w$ and b with $2b$, then the geometric margin does not change. This will in fact come in handy later. Specifically, because of this invariance to the scaling of the parameters, when trying to fit w and b to training data, we can impose an arbitrary scaling constraint on w without changing anything important; for instance, we can demand that $||w|| = 1$, or $|w_1| = 5$, or $|w_1 + b| + |w_2| = 2$, and any of these can be satisfied simply by rescaling w and b.

Finally, given a training set $S = \{(x^{(i)}, y^{(i)}); i = 1, \ldots, m\}$, we also define the geometric margin of (w, b) with respect to S to be the smallest of the geometric margins on the individual training examples:

$$\gamma = \min_{i=1,\ldots,m} \gamma^{(i)}.$$

4 The optimal margin classifier

Given a training set, it seems from our previous discussion that a natural desideratum is to try to find a decision boundary that maximizes the (geometric) margin, since this would reflect a very confident set of predictions

on the training set and a good "fit" to the training data. Specifically, this will result in a classifier that separates the positive and the negative training examples with a "gap" (geometric margin).

For now, we will assume that we are given a training set that is linearly separable; i.e., that it is possible to separate the positive and negative examples using some separating hyperplane. How we we find the one that achieves the maximum geometric margin? We can pose the following optimization problem:

$$\begin{aligned} \max_{\gamma,w,b} \quad & \gamma \\ \text{s.t.} \quad & y^{(i)}(w^T x^{(i)} + b) \geq \gamma, \quad i = 1, \ldots, m \\ & ||w|| = 1. \end{aligned}$$

I.e., we want to maximize γ, subject to each training example having functional margin at least γ. The $||w|| = 1$ constraint moreover ensures that the functional margin equals to the geometric margin, so we are also guaranteed that all the geometric margins are at least γ. Thus, solving this problem will result in (w, b) with the largest possible geometric margin with respect to the training set.

If we could solve the optimization problem above, we'd be done. But the "$||w|| = 1$" constraint is a nasty (non-convex) one, and this problem certainly isn't in any format that we can plug into standard optimization software to solve. So, lets try transforming the problem into a nicer one. Consider:

$$\begin{aligned} \max_{\gamma,w,b} \quad & \frac{\hat{\gamma}}{||w||} \\ \text{s.t.} \quad & y^{(i)}(w^T x^{(i)} + b) \geq \hat{\gamma}, \quad i = 1, \ldots, m \end{aligned}$$

Here, we're going to maximize $\hat{\gamma}/||w||$, subject to the functional margins all being at least $\hat{\gamma}$. Since the geometric and functional margins are related by $\gamma = \hat{\gamma}/||w|$, this will give us the answer we want. Moreover, we've gotten rid of the constraint $||w|| = 1$ that we didn't like. The downside is that we now have a nasty (again, non-convex) objective $\frac{\hat{\gamma}}{||w||}$ function; and, we still don't have any off-the-shelf software that can solve this form of an optimization problem.

Lets keep going. Recall our earlier discussion that we can add an arbitrary scaling constraint on w and b without changing anything. This is the key idea we'll use now. We will introduce the scaling constraint that the functional margin of w, b with respect to the training set must be 1:

$$\hat{\gamma} = 1.$$

Since multiplying w and b by some constant results in the functional margin being multiplied by that same constant, this is indeed a scaling constraint, and can be satisfied by rescaling w, b. Plugging this into our problem above, and noting that maximizing $\hat{\gamma}/||w|| = 1/||w||$ is the same thing as minimizing $||w||^2$, we now have the following optimization problem:

$$\begin{aligned} \min_{\gamma,w,b} \quad & \frac{1}{2}||w||^2 \\ \text{s.t.} \quad & y^{(i)}(w^T x^{(i)} + b) \geq 1, \quad i = 1, \ldots, m \end{aligned}$$

We've now transformed the problem into a form that can be efficiently solved. The above is an optimization problem with a convex quadratic objective and only linear constraints. Its solution gives us the **optimal margin classifier**. This optimization problem can be solved using commercial quadratic programming (QP) code.[1]

While we could call the problem solved here, what we will instead do is make a digression to talk about Lagrange duality. This will lead us to our optimization problem's dual form, which will play a key role in allowing us to use kernels to get optimal margin classifiers to work efficiently in very high dimensional spaces. The dual form will also allow us to derive an efficient algorithm for solving the above optimization problem that will typically do much better than generic QP software.

5 Lagrange duality

Lets temporarily put aside SVMs and maximum margin classifiers, and talk about solving constrained optimization problems.

Consider a problem of the following form:

$$\begin{aligned} \min_{w} \quad & f(w) \\ \text{s.t.} \quad & h_i(w) = 0, \quad i = 1, \ldots, l. \end{aligned}$$

Some of you may recall how the method of Lagrange multipliers can be used to solve it. (Don't worry if you haven't seen it before.) In this method, we define the **Lagrangian** to be

$$\mathcal{L}(w, \beta) = f(w) + \sum_{i=1}^{l} \beta_i h_i(w)$$

[1] You may be familiar with linear programming, which solves optimization problems that have linear objectives and linear constraints. QP software is also widely available, which allows convex quadratic objectives and linear constraints.

Here, the β_i's are called the **Lagrange multipliers**. We would then find and set $\mathcal{L}$'s partial derivatives to zero:

$$\frac{\partial \mathcal{L}}{\partial w_i} = 0; \quad \frac{\partial \mathcal{L}}{\partial \beta_i} = 0,$$

and solve for w and β.

In this section, we will generalize this to constrained optimization problems in which we may have inequality as well as equality constraints. Due to time constraints, we won't really be able to do the theory of Lagrange duality justice in this class,[2] but we will give the main ideas and results, which we will then apply to our optimal margin classifier's optimization problem.

Consider the following, which we'll call the **primal** optimization problem:

$$\begin{aligned} \min_w \quad & f(w) \\ \text{s.t.} \quad & g_i(w) \leq 0, \quad i = 1, \ldots, k \\ & h_i(w) = 0, \quad i = 1, \ldots, l. \end{aligned}$$

To solve it, we start by defining the **generalized Lagrangian**

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

Here, the α_i's and β_i's are the Lagrange multipliers. Consider the quantity

$$\theta_{\mathcal{P}}(w) = \max_{\alpha, \beta \,:\, \alpha_i \geq 0} \mathcal{L}(w, \alpha, \beta).$$

Here, the "$\mathcal{P}$" subscript stands for "primal." Let some w be given. If w violates any of the primal constraints (i.e., if either $g_i(w) > 0$ or $h_i(w) \neq 0$ for some i), then you should be able to verify that

$$\begin{aligned} \theta_{\mathcal{P}}(w) &= \max_{\alpha, \beta \,:\, \alpha_i \geq 0} f(w) + \sum_{i=1}^{k} \alpha_i g_i(w) + \sum_{i=1}^{l} \beta_i h_i(w) \qquad (1) \\ &= \infty. \qquad (2) \end{aligned}$$

Conversely, if the constraints are indeed satisfied for a particular value of w, then $\theta_{\mathcal{P}}(w) = f(w)$. Hence,

$$\theta_{\mathcal{P}}(w) = \begin{cases} f(w) & \text{if } w \text{ satisfies primal constraints} \\ \infty & \text{otherwise.} \end{cases}$$

[2]Readers interested in learning more about this topic are encouraged to read, e.g., R. T. Rockarfeller (1970), Convex Analysis, Princeton University Press.

Thus, $\theta_{\mathcal{P}}$ takes the same value as the objective in our problem for all values of w that satisfies the primal constraints, and is positive infinity if the constraints are violated. Hence, if we consider the minimization problem

$$\min_w \theta_{\mathcal{P}}(w) = \min_w \max_{\alpha,\beta\,:\,\alpha_i \geq 0} \mathcal{L}(w, \alpha, \beta),$$

we see that it is the same problem (i.e., and has the same solutions as) our original, primal problem. For later use, we also define the optimal value of the objective to be $p^* = \min_w \theta_{\mathcal{P}}(w)$; we call this the **value** of the primal problem.

Now, lets look at a slightly different problem. We define

$$\theta_{\mathcal{D}}(\alpha, \beta) = \min_w \mathcal{L}(w, \alpha, \beta).$$

Here, the "$\mathcal{D}$" subscript stands for "dual." Note also that whereas in the definition of $\theta_{\mathcal{P}}$ we were optimizing (maximizing) with respect to α, β, here are are minimizing with respect to w.

We can now pose the **dual** optimization problem:

$$\max_{\alpha,\beta\,:\,\alpha_i \geq 0} \theta_{\mathcal{D}}(\alpha, \beta) = \max_{\alpha,\beta\,:\,\alpha_i \geq 0} \min_w \mathcal{L}(w, \alpha, \beta).$$

This is exactly the same as our primal problem shown above, except that the order of the "max" and the "min" are now exchanged. We also define the optimal value of the dual problem's objective to be $d^* = \max_{\alpha,\beta\,:\,\alpha_i \geq 0} \theta_{\mathcal{D}}(w)$.

How are the primal and the dual problems related? It can easily be shown that

$$d^* = \max_{\alpha,\beta\,:\,\alpha_i \geq 0} \min_w \mathcal{L}(w, \alpha, \beta) \leq \min_w \max_{\alpha,\beta\,:\,\alpha_i \geq 0} \mathcal{L}(w, \alpha, \beta) = p^*.$$

(You should convince yourself of this; this follows from the "max min" of a function always being less than or equal to the "min max.") However, under certain conditions, we will have

$$d^* = p^*,$$

so that we can solve the dual problem in lieu of the primal problem. Lets see what these conditions are.

Suppose f and the g_i's are convex,[3] and the h_i's are affine.[4] Suppose further that the constraints g_i are (strictly) feasible; this means that there exists some w so that $g_i(w) < 0$ for all i.

[3]When f has a Hessian, then it is convex if and only if the hessian is positive semi-definite. For instance, $f(w) = w^T w$ is convex; similarly, all linear (and affine) functions are also convex. (A function f can also be convex without being differentiable, but we won't need those more general definitions of convexity here.)

[4]I.e., there exists a_i, b_i, so that $h_i(w) = a_i^T w + b_i$. "Affine" means the same thing as linear, except that we also allow the extra intercept term b_i.

Under our above assumptions, there must exist w^*, α^*, β^* so that w^* is the solution to the primal problem, α^*, β^* are the solution to the dual problem, and moreover $p^* = d^* = \mathcal{L}(w^*, \alpha^*, \beta^*)$. Moreover, w^*, α^* and β^* satisfy the **Karush-Kuhn-Tucker (KKT) conditions**, which are as follows:

$$\begin{aligned}
\frac{\partial}{\partial w_i}\mathcal{L}(w^*, \alpha^*, \beta^*) &= 0, \quad i = 1, \ldots, n && (3)\\
\frac{\partial}{\partial \beta_i}\mathcal{L}(w^*, \alpha^*, \beta^*) &= 0, \quad i = 1, \ldots, l && (4)\\
\alpha_i^* g_i(w^*) &= 0, \quad i = 1, \ldots, k && (5)\\
g_i(w^*) &\leq 0, \quad i = 1, \ldots, k && (6)\\
\alpha^* &\geq 0, \quad i = 1, \ldots, k && (7)
\end{aligned}$$

Moreover, if some w^*, α^*, β^* satisfy the KKT conditions, then it is also a solution to the primal and dual problems.

We draw attention to Equation (5), which is called the KKT **dual complementarity** condition. Specifically, it implies that if $\alpha_i^* > 0$, then $g_i(w^*) = 0$. (I.e., the "$g_i(w) \leq 0$" constraint is **active**, meaning it holds with equality rather than with inequality.) Later on, this will be key for showing that the SVM has only a small number of "support vectors"; the KKT dual complementarity condition will also give us our convergence test when we talk about the SMO algorithm.

6 Optimal margin classifiers

Previously, we posed the following (primal) optimization problem for finding the optimal margin classifier:

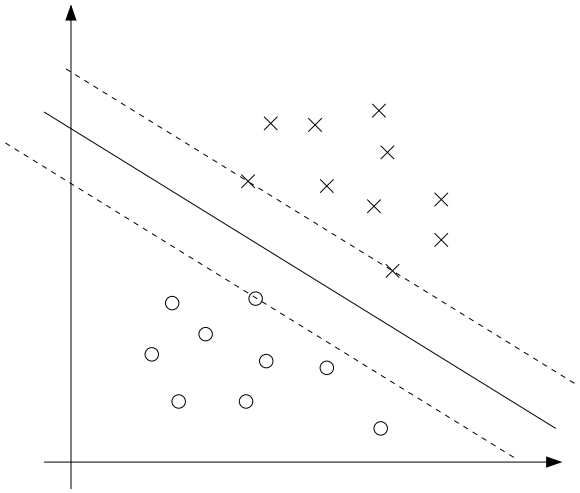
$$\begin{aligned}
\min_{\gamma,w,b} \quad & \frac{1}{2}||w||^2 \\
\text{s.t.} \quad & y^{(i)}(w^T x^{(i)} + b) \geq 1, \quad i = 1, \ldots, m
\end{aligned}$$

We can write the constraints as

$$g_i(w) = -y^{(i)}(w^T x^{(i)} + b) + 1 \leq 0.$$

We have one such constraint for each training example. Note that from the KKT dual complementarity condition, we will have $\alpha_i > 0$ only for the training examples that have functional margin exactly equal to one (i.e., the ones

corresponding to constraints that hold with equality, $g_i(w) = 0$). Consider the figure below, in which a maximum margin separating hyperplane is shown by the solid line.



The points with the smallest margins are exactly the ones closest to the decision boundary; here, these are the three points (one negative and two positive examples) that lie on the dashed lines parallel to the decision boundary. Thus, only three of the α_i's—namely, the ones corresponding to these three training examples—will be non-zero at the optimal solution to our optimization problem. These three points are called the **support vectors** in this problem. The fact that the number of support vectors can be much smaller than the size the training set will be useful later.

Lets move on. Looking ahead, as we develop the dual form of the problem, one key idea to watch out for is that we'll try to write our algorithm in terms of only the inner product $\langle x^{(i)}, x^{(j)} \rangle$ (think of this as $(x^{(i)})^T x^{(j)}$) between points in the input feature space. The fact that we can express our algorithm in terms of these inner products will be key when we apply the kernel trick.

When we construct the Lagrangian for our optimization problem we have:

$$\mathcal{L}(w, b, \alpha) = \frac{1}{2}||w||^2 - \sum_{i=1}^{m} \alpha_i \left[y^{(i)}(w^T x^{(i)} + b) - 1 \right]. \tag{8}$$

Note that there're only "α_i" but no "β_i" Lagrange multipliers, since the problem has only inequality constraints.

Lets find the dual form of the problem. To do so, we need to first minimize $\mathcal{L}(w, b, \alpha)$ with respect to w and b (for fixed α), to get $\theta_{\mathcal{D}}$, which we'll do by

setting the derivatives of $\mathcal{L}$ with respect to w and b to zero. We have:

$$\nabla_w \mathcal{L}(w, b, \alpha) = w - \sum_{i=1}^{m} \alpha_i y^{(i)} x^{(i)} = 0$$

This implies that

$$w = \sum_{i=1}^{m} \alpha_i y^{(i)} x^{(i)}. \tag{9}$$

As for the derivative with respect to b, we obtain

$$\frac{\partial}{\partial b} \mathcal{L}(w, b, \alpha) = \sum_{i=1}^{m} \alpha_i y^{(i)} = 0. \tag{10}$$

If we take the definition of w in Equation (9) and plug that back into the Lagrangian (Equation 8), and simplify, we get

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

But from Equation (10), the last term must be zero, so we obtain

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

Recall that we got to the equation above by minimizing $\mathcal{L}$ with respect to w and b. Putting this together with the constraints $\alpha_i \geq 0$ (that we always had) and the constraint (10), we obtain the following dual optimization problem:

$$\begin{aligned} \max_\alpha \quad & W(\alpha) = \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{m} y^{(i)} y^{(j)} \alpha_i \alpha_j \langle x^{(i)}, x^{(j)} \rangle. \\ \text{s.t.} \quad & \alpha_i \geq 0, \quad i = 1, \ldots, m \\ & \sum_{i=1}^{m} \alpha_i y^{(i)} = 0, \end{aligned}$$

You should also be able to verify that the conditions required for $p^* = d^*$ and the KKT conditions (Equations 3–7) to hold are indeed satisfied in our optimization problem. Hence, we can solve the dual in lieu of solving the primal problem. Specifically, in the dual problem above, we have a maximization problem in which the parameters are the α_i's. We'll talk later

about the specific algorithm that we're going to use to solve the dual problem, but if we are indeed able to solve it (i.e., find the α's that maximize $W(\alpha)$ subject to the constraints), then we can use Equation (9) to go back and find the optimal w's as a function of the α's. Having found w^*, by considering the primal problem, it is also straightforward to find the optimal value for the intercept term b as

$$b^* = -\frac{\max_{i:y^{(i)}=-1} w^{*T} x^{(i)} + \min_{i:y^{(i)}=1} w^{*T} x^{(i)}}{2}. \tag{11}$$

(Check for yourself that this is correct.)

Before moving on, lets also take a more careful look at Equation (9), which gives the optimal value of w in terms of (the optimal value of) α. Suppose we've fit our model's parameters to a training set, and now wish to make a prediction at a new point input x. We would then calculate $w^T x + b$, and predict $y = 1$ if and only if this quantity is bigger than zero. But using (9), this quantity can also be written:

$$\begin{aligned} w^T x + b &= \left(\sum_{i=1}^{m} \alpha_i y^{(i)} x^{(i)} \right)^T x + b && (12) \\ &= \sum_{i=1}^{m} \alpha_i y^{(i)} \langle x^{(i)}, x \rangle + b. && (13) \end{aligned}$$

Hence, if we've found the α_i's, in order to make a prediction, we have to calculate a quantity that depends only on the inner product between x and the points in the training set. Moreover, we saw earlier that the α_i's will all be zero except for the support vectors. Thus, many of the terms in the sum above will be zero, and we really need to find only the inner products between x and the support vectors (of which there is often only a small number) in order calculate (13) and make our prediction.

By examining the dual form of the optimization problem, we gained significant insight into the structure of the problem, and were also able to write the entire algorithm in terms of only inner products between input feature vectors. In the next section, we will exploit this property to apply the kernels to our classification problem. The resulting algorithm, **support vector machines**, will be able to efficiently learn in very high dimensional spaces.

7 Kernels

Back in our discussion of linear regression, we had a problem in which the input x was the living area of a house, and we considered performing regres-

sion using the features x, x^2 and x^3 (say) to obtain a cubic function. To distinguish between these two sets of variables, we'll call the "original" input value the input **attributes** of a problem (in this case, x, the living area). When that is mapped to some new set of quantities that are then passed to the learning algorithm, we'll call those new quantities the input **features**. (Unfortunately, different authors use different terms to describe these two things, but we'll try to use this terminology consistently in these notes.) We will also let ϕ denote the **feature mapping**, which maps from the attributes to the features. For instance, in our example, we had

$$\phi(x) = \begin{bmatrix} x \\ x^2 \\ x^3 \end{bmatrix}.$$

Rather than applying SVMs using the original input attributes x, we may instead want to learn using some features $\phi(x)$. To do so, we simply need to go over our previous algorithm, and replace x everywhere in it with $\phi(x)$.

Since the algorithm can be written entirely in terms of the inner products $\langle x, z\rangle$, this means that we would replace all those inner products with $\langle \phi(x), \phi(z)\rangle$. Specificically, given a feature mapping ϕ, we define the corresponding **Kernel** to be

$$K(x, z) = \phi(x)^T \phi(z).$$

Then, everywhere we previously had $\langle x, z\rangle$ in our algorithm, we could simply replace it with $K(x, z)$, and our algorithm would now be learning using the features ϕ.

Now, given ϕ, we could easily compute $K(x, z)$ by finding $\phi(x)$ and $\phi(z)$ and taking their inner product. But what's more interesting is that often, $K(x, z)$ may be very inexpensive to calculate, even though $\phi(x)$ itself may be very expensive to calculate (perhaps because it is an extremely high dimensional vector). In such settings, by using in our algorithm an efficient way to calculate $K(x, z)$, we can get SVMs to learn in the high dimensional feature space space given by ϕ, but without ever having to explicitly find or represent vectors $\phi(x)$.

Lets see an example. Suppose $x, z \in \mathbb{R}^n$, and consider

$$K(x, z) = (x^T z)^2.$$

We can also write this as

$$\begin{aligned} K(x,z) &= \left(\sum_{i=1}^{n} x_i z_i\right)\left(\sum_{j=1}^{n} x_i z_i\right) \\ &= \sum_{i=1}^{n}\sum_{j=1}^{n} x_i x_j z_i z_j \\ &= \sum_{i,j=1}^{n} (x_i x_j)(z_i z_j) \end{aligned}$$

Thus, we see that $K(x,z) = \phi(x)^T \phi(z)$, where the feature mapping ϕ is given (shown here for the case of $n = 3$) by

$$\phi(x) = \begin{bmatrix} x_1 x_1 \\ x_1 x_2 \\ x_1 x_3 \\ x_2 x_1 \\ x_2 x_2 \\ x_2 x_3 \\ x_3 x_1 \\ x_3 x_2 \\ x_3 x_3 \end{bmatrix}.$$

Note that whereas calculating the high-dimensional $\phi(x)$ requires $O(n^2)$ time, finding $K(x,z)$ takes only $O(n)$ time—linear in the dimension of the input attributes.

For a related kernel, also consider

$$\begin{aligned} K(x,z) &= (x^T z + c)^2 \\ &= \sum_{i,j=1}^{n} (x_i x_j)(z_i z_j) + \sum_{i=1}^{n} (\sqrt{2c}x_i)(\sqrt{2c}z_i) + c^2. \end{aligned}$$

(Check this yourself.) This corresponds to the feature mapping (again shown

for $n = 3$)

$$\phi(x) = \begin{bmatrix} x_1x_1 \\ x_1x_2 \\ x_1x_3 \\ x_2x_1 \\ x_2x_2 \\ x_2x_3 \\ x_3x_1 \\ x_3x_2 \\ x_3x_3 \\ \sqrt{2c}x_1 \\ \sqrt{2c}x_2 \\ \sqrt{2c}x_3 \\ c \end{bmatrix},$$

and the parameter c controls the relative weighting between the x_i (first order) and the x_ix_j (second order) terms.

More broadly, the kernel $K(x, z) = (x^T z + c)^d$ corresponds to a feature mapping to an $\binom{n+d}{d}$ feature space, corresponding of all monomials of the form $x_{i_1}x_{i_2}\ldots x_{i_k}$ that are up to order d. However, despite working in this $O(n^d)$-dimensional space, computing $K(x, z)$ still takes only $O(n)$ time, and hence we never need to explicitly represent feature vectors in this very high dimensional feature space.

Now, lets talk about a slightly different view of kernels. Intuitively, (and there are things wrong with this intuition, but nevermind), if $\phi(x)$ and $\phi(z)$ are close together, then we might expect $K(x, z) = \phi(x)^T\phi(z)$ to be large. Conversely, if $\phi(x)$ and $\phi(z)$ are far apart—say nearly orthogonal to each other—then $K(x, z) = \phi(x)^T\phi(z)$ will be small. So, we can think of $K(x, z)$ as some measurement of how similar are $\phi(x)$ and $\phi(z)$, or of how similar are x and z.

Given this intuition, suppose that for some learning problem that you're working on, you've come up with some function $K(x, z)$ that you think might be a reasonable measure of how similar x and z are. For instance, perhaps you chose

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

This is a resonable measure of x and z's similarity, and is close to 1 when x and z are close, and near 0 when x and z are far apart. Can we use this definition of K as the kernel in an SVM? In this particular example, the answer is yes. (This kernel is called the **Gaussian kernel**, and corresponds

to an infinite dimensional feature mapping ϕ.) But more broadly, given some function K, how can we tell if it's a valid kernel; i.e., can we tell if there is some feature mapping ϕ so that $K(x, z) = \phi(x)^T \phi(z)$ for all x, z?

Suppose for now that K is indeed a valid kernel corresponding to some feature mapping ϕ. Now, consider some finite set of m points (not necessarily the training set) $\{x^{(1)}, \ldots, x^{(m)}\}$, and let a square, m-by-m matrix K be defined so that its (i, j)-entry is given by $K_{ij} = K(x^{(i)}, x^{(j)})$. This matrix is called the **Kernel matrix**. Note that we've overloaded the notation and used K to denote both the kernel function $K(x, z)$ and the kernel matrix K, due to their obvious close relationship.

Now, if K is a valid Kernel, then $K_{ij} = K(x^{(i)}, x^{(j)}) = \phi(x^{(i)})^T \phi(x^{(j)}) = \phi(x^{(j)})^T \phi(x^{(i)}) = K(x^{(j)}, x^{(i)}) = K_{ji}$, and hence K must be symmetric. Moreover, letting $\phi_k(x)$ denote the k-th coordinate of the vector $\phi(x)$, we find that for any vector z, we have

$$\begin{aligned}
z^T K z &= \sum_i \sum_j z_i K_{ij} z_j \\
&= \sum_i \sum_j z_i \phi(x^{(i)})^T \phi(x^{(j)}) z_j \\
&= \sum_i \sum_j z_i \sum_k \phi_k(x^{(i)}) \phi_k(x^{(j)}) z_j \\
&= \sum_k \sum_i \sum_j z_i \phi_k(x^{(i)}) \phi_k(x^{(j)}) z_j \\
&= \sum_k \left(\sum_i z_i \phi_k(x^{(i)}) \right)^2 \\
&\geq 0.
\end{aligned}$$

The second-to-last step above used the same trick as you saw in Problem set 1 Q1. Since z was arbitrary, this shows that K is positive semi-definite ($K \geq 0$).

Hence, we've shown that if K is a valid kernel (i.e., if it corresponds to some feature mapping ϕ), then the corresponding Kernel matrix $K \in \mathbb{R}^{m \times m}$ is symmetric positive semidefinite. More generally, this turns out to be not only a necessary, but also a sufficient, condition for K to be a valid kernel (also called a Mercer kernel). The following result is due to Mercer.[5]

[5]Many texts present Mercer's theorem in a slightly more complicated form involving L^2 functions, but when the input attributes take values in $\mathbb{R}^n$, the version given here is equivalent.

Theorem (Mercer). Let $K : \mathbb{R}^n \times \mathbb{R}^n \mapsto \mathbb{R}$ be given. Then for K to be a valid (Mercer) kernel, it is necessary and sufficient that for any $\{x^{(1)}, \ldots, x^{(m)}\}$, $(m < \infty)$, the corresponding kernel matrix is symmetric positive semi-definite.

Given a function K, apart from trying to find a feature mapping ϕ that corresponds to it, this theorem therefore gives another way of testing if it is a valid kernel. You'll also have a chance to play with these ideas more in problem set 2.

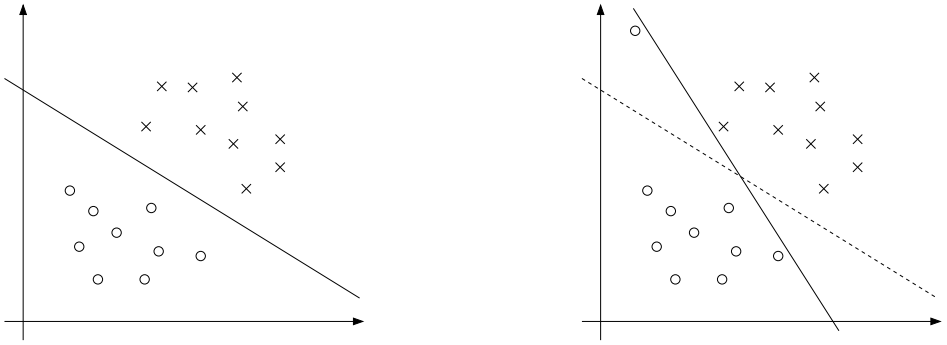
In class, we also briefly talked about a couple of other examples of kernels. For instance, consider the digit recognition problem, in which given an image (16x16 pixels) of a handwritten digit (0-9), we have to figure out which digit it was. Using either a simple polynomial kernel $K(x, z) = (x^T z)^d$ or the Gaussian kernel, SVMs were able to obtain extremely good performance on this problem. This was particularly surprising since the input attributes x were just a 256-dimensional vector of the image pixel intensity values, and the system had no prior knowledge about vision, or even about which pixels are adjacent to which other ones. Another example that we briefly talked about in lecture was that if the objects x that we are trying to classify are strings (say, x is a list of amino acids, which strung together form a protein), then it seems hard to construct a reasonable, "small" set of features for most learning algorithms, especially if different strings have different lengths. However, consider letting $\phi(x)$ be a feature vector that counts the number of occurrences of each length-k substring in x. If we're considering strings of english alphabets, then there're 26^k such strings. Hence, $\phi(x)$ is a 26^k dimensional vector; even for moderate values of k, this is probably too big for us to efficiently work with. (e.g., $26^4 \approx 460000$.) However, using (dynamic programming-ish) string matching algorithms, it is possible to efficiently compute $K(x, z) = \phi(x)^T \phi(z)$, so that we can now implicitly work in this 26^k-dimensional feature space, but without ever explicitly computing feature vectors in this space.

The application of kernels to support vector machines should already be clear and so we won't dwell too much longer on it here. Keep in mind however that the idea of kernels has significantly broader applicability than SVMs. Specifically, if you have any learning algorithm that you can write in terms of only inner products $\langle x, z \rangle$ between input attribute vectors, then by replacing this with $K(x, z)$ where K is a kernel, you can "magically" allow your algorithm to work efficiently in the high dimensional feature space corresponding to K. For instance, this kernel trick can be applied with the perceptron to to derive a kernel perceptron algorithm. Many of the

algorithms that we'll see later in this class will also be amenable to this method, which has come to be known as the "kernel trick."

8 Regularization and the non-separable case

The derivation of the SVM as presented so far assumed that the data is linearly separable. While mapping data to a high dimensional feature space via ϕ does generally increase the likelihood that the data is separable, we can't guarantee that it always will be so. Also, in some cases it is not clear that finding a separating hyperplane is exactly what we'd want to do, since that might be susceptible to outliers. For instance, the left figure below shows an optimal margin classifier, and when a single outlier is added in the upper-left region (right figure), it causes the decision boundary to make a dramatic swing, and the resulting classifier has a much smaller margin.



To make the algorithm work for non-linearly separable datasets as well as be less sensitive to outliers, we reformulate our optimization (using ℓ_1 **regularization**) as follows:

$$\begin{aligned} \min_{\gamma,w,b} \quad & \frac{1}{2}||w||^2 + C\sum_{i=1}^{m}\xi_i \\ \text{s.t.} \quad & y^{(i)}(w^T x^{(i)} + b) \geq 1 - \xi_i, \quad i = 1,\ldots,m \\ & \xi_i \geq 0, \quad i = 1,\ldots,m. \end{aligned}$$

Thus, examples are now permitted to have (functional) margin less than 1, and if an example whose functional margin is $1 - \xi_i$, we would pay a cost of the objective function being increased by $C\xi_i$. The parameter C controls the relative weighting between the twin goals of making the $||w||^2$ large (which we saw earlier makes the margin small) and of ensuring that most examples have functional margin at least 1.

As before, we can form the Lagrangian:

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

Here, the α_i's and r_i's are our Lagrange multipliers (constrained to be ≥ 0). We won't go through the derivation of the dual again in detail, but after setting the derivatives with respect to w and b to zero as before, substituting them back in, and simplifying, we obtain the following dual form of the problem:

$$\begin{aligned} \max_\alpha \quad & W(\alpha) = \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{m} y^{(i)} y^{(j)} \alpha_i \alpha_j \langle x^{(i)}, x^{(j)} \rangle \\ \text{s.t.} \quad & 0 \leq \alpha_i \leq C, \quad i = 1, \ldots, m \\ & \sum_{i=1}^{m} \alpha_i y^{(i)} = 0, \end{aligned}$$

As before, we also have that w can be expressed in terms of the α_i's as given in Equation (9), so that after solving the dual problem, we can continue to use Equation (13) to make our predictions. Note that, somewhat surprisingly, in adding ℓ_1 regularization, the only change to the dual problem is that what was originally a constraint that $0 \leq \alpha_i$ has now become $0 \leq \alpha_i \leq C$. The calculation for b^* also has to be modified (Equation 11 is no longer valid); see the comments in the next section/Platt's paper.

Also, the KKT dual-complementarity conditions (which in the next section will be useful for testing for the convergence of the SMO algorithm) are:

$$\alpha_i = 0 \quad \Rightarrow \quad y^{(i)}(w^T x^{(i)} + b) \geq 1 \tag{14}$$

$$\alpha_i = C \quad \Rightarrow \quad y^{(i)}(w^T x^{(i)} + b) \leq 1 \tag{15}$$

$$0 < \alpha_i < C \quad \Rightarrow \quad y^{(i)}(w^T x^{(i)} + b) = 1. \tag{16}$$

Now, all that remains is to give an algorithm for actually solving the dual problem, which we will do in the next section.

9 The SMO algorithm

The SMO (sequential minimal optimization) algorithm, due to John Platt, gives an efficient way of solving the dual problem arising from the derivation

of the SVM. Partly to motivate the SMO algorithm, and partly because it's interesting in its own right, lets first take another digression to talk about the coordinate ascent algorithm.

9.1 Coordinate ascent

Consider trying to solve the unconstrained optimization problem

$$\max_{\alpha} W(\alpha_1, \alpha_2, \ldots, \alpha_m).$$

Here, we think of W as just some function of the parameters α_i's, and for now ignore any relationship between this problem and SVMs. We've already seen two optimization algorithms, gradient ascent and Newton's method. The new algorithm we're going to consider here is called **coordinate ascent**:

Loop until convergence: {

For $i = 1, \ldots, m$, {

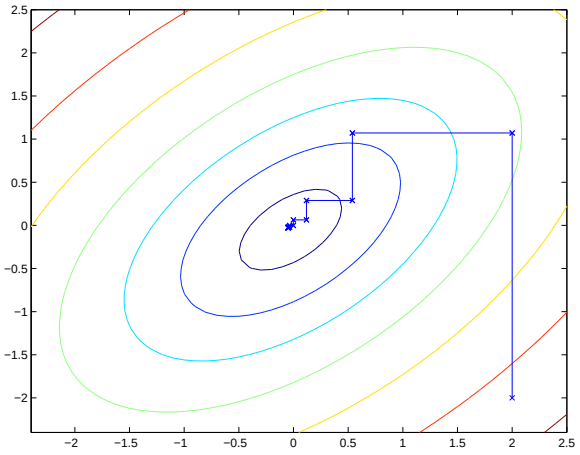
$\alpha_i := \arg\max_{\hat{\alpha}_i} W(\alpha_1, \ldots, \alpha_{i-1}, \hat{\alpha}_i, \alpha_{i+1}, \ldots, \alpha_m)$.

}

}

Thus, in the innermost loop of this algorithm, we will hold all the variables except for some α_i fixed, and reoptimize W with respect to just the parameter α_i. In the version of this method presented here, the inner-loop reoptimizes the variables in order $\alpha_1, \alpha_2, \ldots, \alpha_m, \alpha_1, \alpha_2, \ldots$. (A more sophisticated version might choose other orderings; for instance, we may choose the next variable to update according to which one we expect to allow us to make the largest increase in $W(\alpha)$.)

When the function W happens to be of such a form that the "arg max" in the inner loop can be performed efficiently, then coordinate ascent can be a fairly efficient algorithm. Here's a picture of coordinate ascent in action:



The ellipses in the figure are the contours of a quadratic function that we want to optimize. Coordinate ascent was initialized at $(2, -2)$, and also plotted in the figure is the path that it took on its way to the global maximum. Notice that on each step, coordinate ascent takes a step that's parallel to one of the axes, since only one variable is being optimized at a time.

9.2 SMO

We close off the discussion of SVMs by sketching the derivation of the SMO algorithm. Some details will be left to the homework, and for others you may refer to the paper excerpt handed out in class.

Here's the (dual) optimization problem that we want to solve:

$$\max_\alpha \quad W(\alpha) = \sum_{i=1}^{m} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{m} y^{(i)} y^{(j)} \alpha_i \alpha_j \langle x^{(i)}, x^{(j)} \rangle. \tag{17}$$

$$\text{s.t.} \quad 0 \leq \alpha_i \leq C, \quad i = 1, \ldots, m \tag{18}$$

$$\sum_{i=1}^{m} \alpha_i y^{(i)} = 0. \tag{19}$$

Lets say we have set of α_i's that satisfy the constraints (18-19). Now, suppose we want to hold $\alpha_2, \ldots, \alpha_m$ fixed, and take a coordinate ascent step and reoptimize the objective with respect to α_1. Can we make any progress? The answer is no, because the constraint (19) ensures that

$$\alpha_1 y^{(1)} = -\sum_{i=2}^{m} \alpha_i y^{(i)}.$$

Or, by multiplying both sides by $y^{(1)}$, we equivalently have

$$\alpha_1 = -y^{(1)} \sum_{i=2}^{m} \alpha_i y^{(i)}.$$

(This step used the fact that $y^{(1)} \in \{-1, 1\}$, and hence $(y^{(1)})^2 = 1$.) Hence, α_1 is exactly determined by the other α_i's, and if we were to hold $\alpha_2, \ldots, \alpha_m$ fixed, then we can't make any change to α_1 without violating the constraint (19) in the optimization problem.

Thus, if we want to update some subject of the α_i's, we must update at least two of them simultaneously in order to keep satisfying the constraints. This motivates the SMO algorithm, which simply does the following:

Repeat till convergence {

1. Select some pair α_i and α_j to update next (using a heuristic that tries to pick the two that will allow us to make the biggest progress towards the global maximum).
2. Reoptimize $W(\alpha)$ with respect to α_i and α_j, while holding all the other α_k's ($k \neq i, j$) fixed.

}

To test for convergence of this algorithm, we can check whether the KKT conditions (Equations 14-16) are satisfied to within some *tol*. Here, *tol* is the convergence tolerance parameter, and is typically set to around 0.01 to 0.001. (See the paper and pseudocode for details.)

The key reason that SMO is an efficient algorithm is that the update to α_i, α_j can be computed very efficiently. Lets now briefly sketch the main ideas for deriving the efficient update.

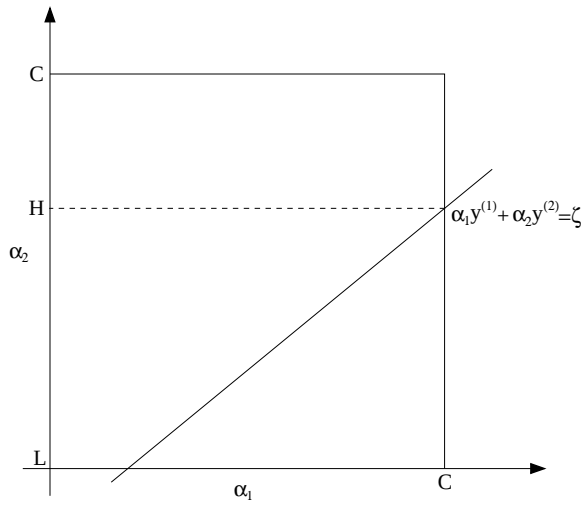
Lets say we currently have some setting of the α_i's that satisfy the constraints (18-19), and suppose we've decided to hold $\alpha_3, \ldots, \alpha_m$ fixed, and want to reoptimize $W(\alpha_1, \alpha_2, \ldots, \alpha_m)$ with respect to α_1 and α_2 (subject to the constraints). From (19), we require that

$$\alpha_1 y^{(1)} + \alpha_2 y^{(2)} = -\sum_{i=3}^{m} \alpha_i y^{(i)}.$$

Since the right hand side is fixed (as we've fixed $\alpha_3, \ldots \alpha_m$), we can just let it be denoted by some constant ζ:

$$\alpha_1 y^{(1)} + \alpha_2 y^{(2)} = \zeta. \tag{20}$$

We can thus picture the constraints on α_1 and α_2 as follows:



From the constraints (18), we know that α_1 and α_2 must lie within the box $[0, C] \times [0, C]$ shown. Also plotted is the line $\alpha_1 y^{(1)} + \alpha_2 y^{(2)} = \zeta$, on which we know α_1 and α_2 must lie. Note also that, from these constraints, we know $L \leq \alpha_2 \leq H$; otherwise, (α_1, α_2) can't simultaneously satisfy both the box and the straight line constraint. In this example, $L = 0$. But depending on what the line $\alpha_1 y^{(1)} + \alpha_2 y^{(2)} = \zeta$ looks like, this won't always necessarily be the case; but more generally, there will be some lower-bound L and some upper-bound H on the permissable values for α_2 that will ensure that α_1, α_2 lie within the box $[0, C] \times [0, C]$.

Using Equation (20), we can also write α_1 as a function of α_2:

$$\alpha_1 = (\zeta - \alpha_2 y^{(2)}) y^{(1)}.$$

(Check this derivation yourself; we again used the fact that $y^{(1)} \in \{-1, 1\}$ so that $(y^{(1)})^2 = 1$.) Hence, the objective $W(\alpha)$ can be written

$$W(\alpha_1, \alpha_2, \ldots, \alpha_m) = W((\zeta - \alpha_2 y^{(2)}) y^{(1)}, \alpha_2, \ldots, \alpha_m).$$

Treating $\alpha_3, \ldots, \alpha_m$ as constants, you should be able to verify that this is just some quadratic function in α_2. I.e., this can also be expressed in the form $a\alpha_2^2 + b\alpha_2 + c$ for some appropriate a, b, and c. If we ignore the "box" constraints (18) (or, equivalently, that $L \leq \alpha_2 \leq H$), then we can easily maximize this quadratic function by setting its derivative to zero and solving. We'll let $\alpha_2^{new,unclipped}$ denote the resulting value of α_2. You should also be able to convince yourself that if we had instead wanted to maximize W with respect to α_2 but subject to the box constraint, then we can find the resulting value optimal simply by taking $\alpha_2^{new,unclipped}$ and "clipping" it to lie in the

$[L, H]$ interval, to get

$$\alpha_2^{new} = \begin{cases} H & \text{if } \alpha_2^{new,unclipped} > H \\ \alpha_2^{new,unclipped} & \text{if } L \leq \alpha_2^{new,unclipped} \leq H \\ L & \text{if } \alpha_2^{new,unclipped} < L \end{cases}$$

Finally, having found the α_2^{new}, we can use Equation (20) to go back and find the optimal value of α_1^{new}.

There're a couple more details that are quite easy but that we'll leave you to read about yourself in Platt's paper: One is the choice of the heuristics used to select the next α_i, α_j to update; the other is how to update b as the SMO algorithm is run.

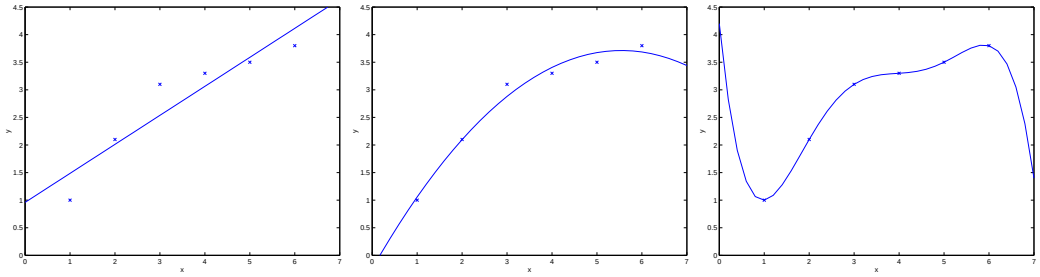
CS229 Lecture notes

Andrew Ng

Part VI
Learning Theory

1 Bias/variance tradeoff

When talking about linear regression, we discussed the problem of whether to fit a "simple" model such as the linear "$y = \theta_0 + \theta_1 x$," or a more "complex" model such as the polynomial "$y = \theta_0 + \theta_1 x + \cdots \theta_5 x^5$." We saw the following example:



Fitting a 5th order polynomial to the data (rightmost figure) did not result in a good model. Specifically, even though the 5th order polynomial did a very good job predicting y (say, prices of houses) from x (say, living area) for the examples in the training set, we do not expect the model shown to be a good one for predicting the prices of houses not in the training set. In other words, what's has been learned from the training set does not *generalize* well to other houses. The **generalization error** (which will be made formal shortly) of a hypothesis is its expected error on examples not necessarily in the training set.

Both the models in the leftmost and the rightmost figures above have large generalization error. However, the problems that the two models suffer from are very different. If the relationship between y and x is not linear,

then even if we were fitting a linear model to a very large amount of training data, the linear model would still fail to accurately capture the structure in the data. Informally, we define the **bias** of a model to be the expected generalization error even if we were to fit it to a very (say, infinitely) large training set. Thus, for the problem above, the linear model suffers from large bias, and may underfit (i.e., fail to capture structure exhibited by) the data.

Apart from bias, there's a second component to the generalization error, consisting of the **variance** of a model fitting procedure. Specifically, when fitting a 5th order polynomial as in the rightmost figure, there is a large risk that we're fitting patterns in the data that happened to be present in our small, finite training set, but that do not reflect the wider pattern of the relationship between x and y. This could be, say, because in the training set we just happened by chance to get a slightly more-expensive-than-average house here, and a slightly less-expensive-than-average house there, and so on. By fitting these "spurious" patterns in the training set, we might again obtain a model with large generalization error. In this case, we say the model has large variance.[1]

Often, there is a tradeoff between bias and variance. If our model is too "simple" and has very few parameters, then it may have large bias (but small variance); if it is too "complex" and has very many parameters, then it may suffer from large variance (but have smaller bias). In the example above, fitting a quadratic function does better than either of the extremes of a first or a fifth order polynomial.

2 Preliminaries

In this set of notes, we begin our foray into learning theory. Apart from being interesting and enlightening in its own right, this discussion will also help us hone our intuitions and derive rules of thumb about how to best apply learning algorithms in different settings. We will also seek to answer a few questions: First, can we make formal the bias/variance tradeoff that was just discussed? The will also eventually lead us to talk about model selection methods, which can, for instance, automatically decide what order polynomial to fit to a training set. Second, in machine learning it's really

[1] In these notes, we will not try to formalize the definitions of bias and variance beyond this discussion. While bias and variance are straightforward to define formally for, e.g., linear regression, there have been several proposals for the definitions of bias and variance for classification, and there is as yet no agreement on what is the "right" and/or the most useful formalism.

generalization error that we care about, but most learning algorithms fit their models to the training set. Why should doing well on the training set tell us anything about generalization error? Specifically, can we relate error on the training set to generalization error? Third and finally, are there conditions under which we can actually prove that learning algorithms will work well?

We start with two simple but very useful lemmas.

Lemma. (The union bound). Let $A_1, A_2, \ldots, A_k$ be k different events (that may not be independent). Then

$$P(A_1 \cup \cdots \cup A_k) \leq P(A_1) + \ldots + P(A_k).$$

In probability theory, the union bound is usually stated as an axiom (and thus we won't try to prove it), but it also makes intuitive sense: The probability of any one of k events happening is at most the sums of the probabilities of the k different events.

Lemma. (Hoeffding inequality) Let $Z_1, \ldots, Z_m$ be m independent and identically distributed (iid) random variables drawn from a Bernoulli(ϕ) distribution. I.e., $P(Z_i = 1) = \phi$, and $P(Z_i = 0) = 1 - \phi$. Let $\hat{\phi} = (1/m) \sum_{i=1}^{m} Z_i$ be the mean of these random variables, and let any $\gamma > 0$ be fixed. Then

$$P(|\phi - \hat{\phi}| > \gamma) \leq 2 \exp(-2\gamma^2 m)$$

This lemma (which in learning theory is also called the **Chernoff bound**) says that if we take $\hat{\phi}$—the average of m Bernoulli(ϕ) random variables—to be our estimate of ϕ, then the probability of our being far from the true value is small, so long as m is large. Another way of saying this is that if you have a biased coin whose chance of landing on heads is ϕ, then if you toss it m times and calculate the fraction of times that it came up heads, that will be a good estimate of ϕ with high probability (if m is large).

Using just these two lemmas, we will be able to prove some of the deepest and most important results in learning theory.

To simplify our exposition, lets restrict our attention to binary classification in which the labels are $y \in \{0, 1\}$. Everything we'll say here generalizes to other, including regression and multi-class classification, problems.

We assume we are given a training set $S = \{(x^{(i)}, y^{(i)}); i = 1, \ldots, m\}$ of size m, where the training examples $(x^{(i)}, y^{(i)})$ are drawn iid from some probability distribution $\mathcal{D}$. For a hypothesis h, we define the **training error** (also called the **empirical risk** or **empirical error** in learning theory) to be

$$\hat{\varepsilon}(h) = \frac{1}{m} \sum_{i=1}^{m} 1\{h(x^{(i)}) \neq y^{(i)}\}.$$

This is just the fraction of training examples that h misclassifies. When we want to make explicit the dependence of $\hat{\varepsilon}(h)$ on the training set S, we may also write this a $\hat{\varepsilon}_S(h)$. We also define the generalization error to be

$$\varepsilon(h) = P_{(x,y)\sim\mathcal{D}}(h(x) \neq y).$$

I.e. this is the probability that, if we now draw a new example (x, y) from the distribution $\mathcal{D}$, h will misclassify it.

Note that we have assumed that the training data was drawn from the *same* distribution $\mathcal{D}$ with which we're going to evaluate our hypotheses (in the definition of generalization error). This is sometimes also referred to as one of the **PAC** assumptions.[2]

Consider the setting of linear classification, and let $h_\theta(x) = 1\{\theta^T x \geq 0\}$. What's a reasonable way of fitting the parameters θ? One approach is to try to minimize the training error, and pick

$$\hat{\theta} = \arg\min_\theta \hat{\varepsilon}(h_\theta).$$

We call this process **empirical risk minimization** (ERM), and the resulting hypothesis output by the learning algorithm is $\hat{h} = h_{\hat{\theta}}$. We think of ERM as the most "basic" learning algorithm, and it will be this algorithm that we focus on in these notes. (Algorithms such as logistic regression can also be viewed as approximations to empirical risk minimization.)

In our study of learning theory, it will be useful to abstract away from the specific parameterization of hypotheses and from issues such as whether we're using a linear classifier. We define the **hypothesis class** $\mathcal{H}$ used by a learning algorithm to be the set of all classifiers considered by it. For linear classification, $\mathcal{H} = \{h_\theta : h_\theta(x) = 1\{\theta^T x \geq 0\}, \theta \in \mathbb{R}^{n+1}\}$ is thus the set of all classifiers over $\mathcal{X}$ (the domain of the inputs) where the decision boundary is linear. More broadly, if we were studying, say, neural networks, then we could let $\mathcal{H}$ be the set of all classifiers representable by some neural network architecture.

Empirical risk minimization can now be thought of as a minimization over the class of functions $\mathcal{H}$, in which the learning algorithm picks the hypothesis:

$$\hat{h} = \arg\min_{h\in\mathcal{H}} \hat{\varepsilon}(h)$$

[2]PAC stands for "probably approximately correct," which is a framework and set of assumptions under which numerous results on learning theory were proved. Of these, the assumption of training and testing on the same distribution, and the assumption of the independently drawn training examples, were the most important.

3 The case of finite $\mathcal{H}$

Lets start by considering a learning problem in which we have a finite hypothesis class $\mathcal{H} = \{h_1, \ldots, h_k\}$ consisting of k hypotheses. Thus, $\mathcal{H}$ is just a set of k functions mapping from $\mathcal{X}$ to $\{0, 1\}$, and empirical risk minimization selects $\hat{h}$ to be whichever of these k functions has the smallest training error.

We would like to give guarantees on the generalization error of $\hat{h}$. Our strategy for doing so will be in two parts: First, we will show that $\hat{\varepsilon}(h)$ is a reliable estimate of $\varepsilon(h)$ for all h. Second, we will show that this implies an upper-bound on the generalization error of $\hat{h}$.

Take any one, fixed, $h_i \in \mathcal{H}$. Consider a Bernoulli random variable Z whose distribution is defined as follows. We're going to sample $(x, y) \sim \mathcal{D}$. Then, we set $Z = 1\{h_i(x) \neq y\}$. I.e., we're going to draw one example, and let Z indicate whether h_i misclassifies it. Similarly, we also define $Z_j = 1\{h_i(x^{(j)}) \neq y^{(j)}\}$. Since our training set was drawn iid from $\mathcal{D}$, Z and the Z_j's have the same distribution.

We see that the misclassification probability on a randomly drawn example—that is, $\varepsilon(h)$—is exactly the expected value of Z (and Z_j). Moreover, the training error can be written

$$\hat{\varepsilon}(h_i) = \frac{1}{m} \sum_{j=1}^{m} Z_j.$$

Thus, $\hat{\varepsilon}(h_i)$ is exactly the mean of the m random variables Z_j that are drawn iid from a Bernoulli distribution with mean $\varepsilon(h_i)$. Hence, we can apply the Hoeffding inequality, and obtain

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

This shows that, for our particular h_i, training error will be close to generalization error with high probability, assuming m is large. But we don't just want to guarantee that $\varepsilon(h_i)$ will be close to $\hat{\varepsilon}(h_i)$ (with high probability) for just only one particular h_i. We want to prove that this will be true for simultaneously for *all* $h \in \mathcal{H}$. To do so, let A_i denote the event that $|\varepsilon(h_i) - \hat{\varepsilon}(h_i)| > \gamma$. We've already show that, for any particular A_i, it holds true that $P(A_i) \leq 2 \exp(-2\gamma^2 m)$. Thus, using the union bound, we

have that

$$\begin{aligned} P(\exists\, h \in \mathcal{H}. |\varepsilon(h_i) - \hat{\varepsilon}(h_i)| > \gamma) &= P(A_1 \cup \cdots \cup A_k) \\ &\leq \sum_{i=1}^{k} P(A_i) \\ &\leq \sum_{i=1}^{k} 2\exp(-2\gamma^2 m) \\ &= 2k\exp(-2\gamma^2 m) \end{aligned}$$

If we subtract both sides from 1, we find that

$$\begin{aligned} P(\neg\exists\, h \in \mathcal{H}. |\varepsilon(h_i) - \hat{\varepsilon}(h_i)| > \gamma) &= P(\forall h \in \mathcal{H}. |\varepsilon(h_i) - \hat{\varepsilon}(h_i)| \leq \gamma) \\ &\geq 1 - 2k\exp(-2\gamma^2 m) \end{aligned}$$

(The "$\neg$" symbol means "not.") So, with probability at least $1-2k\exp(-2\gamma^2 m)$, we have that $\varepsilon(h)$ will be within γ of $\hat{\varepsilon}(h)$ for all $h \in \mathcal{H}$. This is called a *uniform convergence* result, because this is a bound that holds simultaneously for all (as opposed to just one) $h \in \mathcal{H}$.

In the discussion above, what we did was, for particular values of m and γ, given a bound on the probability that, for some $h \in \mathcal{H}$, $|\varepsilon(h) - \hat{\varepsilon}(h)| > \gamma$. There are three quantities of interest here: m, γ, and the probability of error; we can bound either one in terms of the other two.

For instance, we can ask the following question: Given γ and some $\delta > 0$, how large must m be before we can guarantee that with probability at least $1 - \delta$, training error will be within γ of generalization error? By setting $\delta = 2k\exp(-2\gamma^2 m)$ and solving for m, [you should convince yourself this is the right thing to do!], we find that if

$$m \geq \frac{1}{2\gamma^2} \log \frac{2k}{\delta},$$

then with probability at least $1 - \delta$, we have that $|\varepsilon(h) - \hat{\varepsilon}(h)| \leq \gamma$ for all $h \in \mathcal{H}$. (Equivalently, this show that the probability that $|\varepsilon(h) - \hat{\varepsilon}(h)| > \gamma$ for some $h \in \mathcal{H}$ is at most δ.) This bound tells us how many training examples we need in order make a guarantee. The training set size m that a certain method or algorithm requires in order to achieve a certain level of performance is also called the algorithm's **sample complexity**.

The key property of the bound above is that the number of training examples needed to make this guarantee is only *logarithmic* in k, the number of hypotheses in $\mathcal{H}$. This will be important later.

Similarly, we can also hold m and δ fixed and solve for γ in the previous equation, and show [again, convince yourself that this is right!] that with probability $1-\delta$, we have that for all $h \in \mathcal{H}$,

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

Now, lets assume that uniform convergence holds, i.e., that $|\varepsilon(h)-\hat{\varepsilon}(h)| \leq \gamma$ for all $h \in \mathcal{H}$. What can we prove about the generalization of our learning algorithm that picked $\hat{h} = \arg\min_{h \in \mathcal{H}} \hat{\varepsilon}(h)$?

Define $h^* = \arg\min_{h \in \mathcal{H}} \varepsilon(h)$ to be the best possible hypothesis in $\mathcal{H}$. Note that h^* is the best that we could possibly do given that we are using $\mathcal{H}$, so it makes sense to compare our performance to that of h^*. We have:

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

The first line used the fact that $|\varepsilon(\hat{h})-\hat{\varepsilon}(\hat{h})| \leq \gamma$ (by our uniform convergence assumption). The second used the fact that $\hat{h}$ was chosen to minimize $\hat{\varepsilon}(h)$, and hence $\hat{\varepsilon}(\hat{h}) \leq \hat{\varepsilon}(h)$ for all h, and in particular $\hat{\varepsilon}(\hat{h}) \leq \hat{\varepsilon}(h^*)$. The third line used the uniform convergence assumption again, to show that $\hat{\varepsilon}(h^*) \leq \varepsilon(h^*) + \gamma$. So, what we've shown is the following: If uniform convergence occurs, then the generalization error of $\hat{h}$ is at most 2γ worse than the best possible hypothesis in $\mathcal{H}$!

Lets put all this together into a theorem.

Theorem. Let $|\mathcal{H}| = k$, and let any m, δ be fixed. Then with probability at least $1-\delta$, we have that

$$\varepsilon(\hat{h}) \leq \left(\min_{h \in \mathcal{H}} \varepsilon(h) \right) + 2\sqrt{\frac{1}{2m} \log \frac{2k}{\delta}}.$$

This is proved by letting γ equal the $\sqrt{\cdot}$ term, using our previous argument that uniform convergence occurs with probability at least $1-\delta$, and then noting that uniform convergence implies $\varepsilon(h)$ is at most 2γ higher than $\varepsilon(h^*) = \min_{h \in \mathcal{H}} \varepsilon(h)$ (as we showed previously).

This also quantifies what we were saying previously saying about the bias/variance tradeoff in model selection. Specifically, suppose we have some hypothesis class $\mathcal{H}$, and are considering switching to some much larger hypothesis class $\mathcal{H}' \supseteq \mathcal{H}$. If we switch to $\mathcal{H}'$, then the first term $\min_h \varepsilon(h)$

can only decrease (since we'd then be taking a min over a larger set of functions). Hence, by learning using a larger hypothesis class, our "bias" can only decrease. However, if k increases, then the second $2\sqrt{\cdot}$ term would also increase. This increase corresponds to our "variance" increasing when we use a larger hypothesis class.

By holding γ and δ fixed and solving for m like we did before, we can also obtain the following sample complexity bound:

Corollary. Let $|\mathcal{H}| = k$, and let any δ, γ be fixed. Then for $\varepsilon(\hat{h}) \leq \min_{h \in \mathcal{H}} \varepsilon(h) + 2\gamma$ to hold with probability at least $1 - \delta$, it suffices that

$$\begin{aligned} m &\geq \frac{1}{2\gamma^2} \log \frac{2k}{\delta} \\ &= O\left(\frac{1}{\gamma^2} \log \frac{k}{\delta}\right), \end{aligned}$$

4 The case of infinite $\mathcal{H}$

We have proved some useful theorems for the case of finite hypothesis classes. But many hypothesis classes, including any parameterized by real numbers (as in linear classification) actually contain an infinite number of functions. Can we prove similar results for this setting?

Lets start by going through something that is *not* the "right" argument. *Better and more general arguments exist*, but this will be useful for honing our intuitions about the domain.

Suppose we have an $\mathcal{H}$ that is parameterized by d real numbers. Since we are using a computer to represent real numbers, and IEEE double-precision floating point (`double`'s in C) uses 64 bits to represent a floating point number, this means that our learning algorithm, assuming we're using double-precision floating point, is parameterized by $64d$ bits. Thus, our hypothesis class really consists of at most $k = 2^{64d}$ different hypotheses. From the Corollary at the end of the previous section, we therefore find that, to guarantee $\varepsilon(\hat{h}) \leq \varepsilon(h^*) + 2\gamma$, with to hold with probability at least $1 - \delta$, it suffices that $m \geq O\left(\frac{1}{\gamma^2} \log \frac{2^{64d}}{\delta}\right) = O\left(\frac{d}{\gamma^2} \log \frac{1}{\delta}\right) = O_{\gamma,\delta}(d)$. (The γ, δ subscripts are to indicate that the last big-O is hiding constants that may depend on γ and δ.) Thus, the number of training examples needed is at most *linear* in the parameters of the model.

The fact that we relied on 64-bit floating point makes this argument not entirely satisfying, but the conclusion is nonetheless roughly correct: If what we're going to do is try to minimize training error, then in order to learn

"well" using a hypothesis class that has d parameters, generally we're going to need on the order of a linear number of training examples in d.

(At this point, it's worth noting that these results were proved for an algorithm that uses empirical risk minimization. Thus, while the linear dependence of sample complexity on d does generally hold for most discriminative learning algorithms that try to minimize training error or some approximation to training error, these conclusions do not always apply as readily to discriminative learning algorithms. Giving good theoretical guarantees on many non-ERM learning algorithms is still an area of active research.)

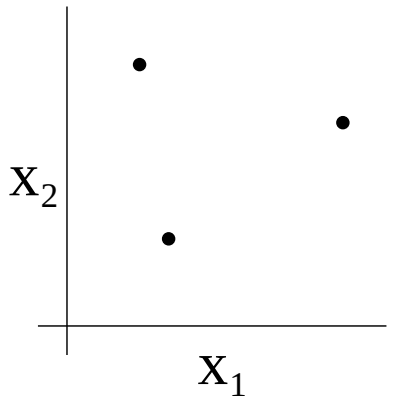
The other part of our previous argument that's slightly unsatisfying is that it relies on the parameterization of $\mathcal{H}$. Intuitively, this doesn't seem like it should matter: We had written the class of linear classifiers as $h_\theta(x) = 1\{\theta_0 + \theta_1 x_1 + \cdots \theta_n x_n \geq 0\}$, with $n+1$ parameters $\theta_0, \ldots, \theta_n$. But it could also be written $h_{u,v}(x) = 1\{(u_0^2 - v_0^2) + (u_1^2 - v_1^2)x_1 + \cdots (u_n^2 - v_n^2)x_n \geq 0\}$ with $2n+2$ parameters u_i, v_i. Yet, both of these are just defining the same $\mathcal{H}$: The set of linear classifiers in n dimensions.

To derive a more satisfying argument, lets define a few more things.

Given a set $S = \{x^{(i)}, \ldots, x^{(d)}\}$ (no relation to the training set) of points $x^{(i)} \in \mathcal{X}$, we say that $\mathcal{H}$ **shatters** S if $\mathcal{H}$ can realize any labeling on S. I.e., if for any set of labels $\{y^{(1)}, \ldots, y^{(d)}\}$, there exists some $h \in \mathcal{H}$ so that $h(x^{(i)}) = y^{(i)}$ for all $i = 1, \ldots d$.

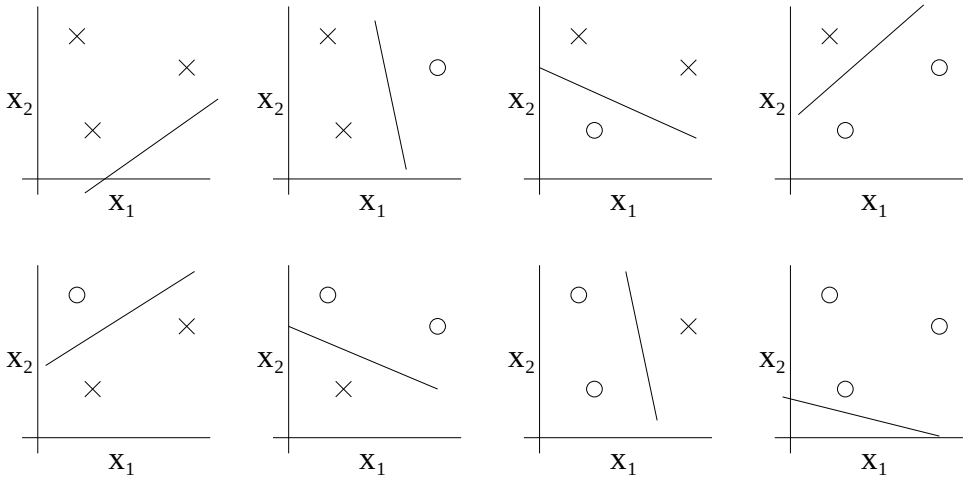
Given a hypothesis class $\mathcal{H}$, we then define its **Vapnik-Chervonenkis dimension**, written $\text{VC}(\mathcal{H})$, to be the size of the largest set that is shattered by $\mathcal{H}$. (If $\mathcal{H}$ can shatter arbitrarily large sets, then $\text{VC}(\mathcal{H}) = \infty$.)

For instance, consider the following set of three points:



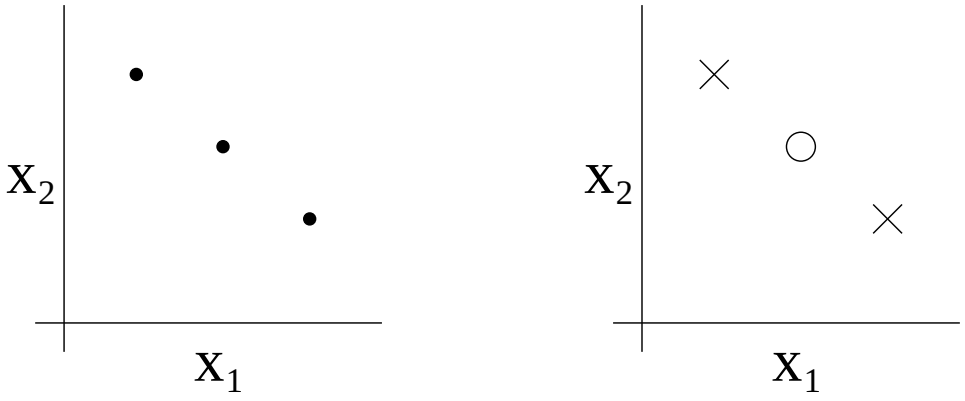
Can the set $\mathcal{H}$ of linear classifiers in two dimensions ($h(x) = 1\{\theta_0 + \theta_1 x_1 + \theta_2 x_2 \geq 0\}$) can shatter the set above? The answer is yes. Specifically, we

see that, for any of the eight possible labelings of these points, we can find a linear classifier that obtains "zero training error" on them:



Moreover, it is possible to show that there is no set of 4 points that this hypothesis class can shatter. Thus, the largest set that $\mathcal{H}$ can shatter is of size 3, and hence $\text{VC}(\mathcal{H}) = 3$.

Note that the VC dimension of $\mathcal{H}$ here is 3 even though there may be sets of size 3 that it cannot shatter. For instance, if we had a set of three points lying in a straight line (left figure), then there is no way to find a linear separator for the labeling of the three points shown below (right figure):



In order words, under the definition of the VC dimension, in order to prove that $\text{VC}(\mathcal{H})$ is at least d, we need to show only that there's at least *one* set of size d that $\mathcal{H}$ can shatter.

The following theorem, due to Vapnik, can then be shown. (This is, many would argue, the most important theorem in all of learning theory.)

Theorem. Let $\mathcal{H}$ be given, and let $d = \mathrm{VC}(\mathcal{H})$. Then with probability at least $1 - \delta$, we have that for all $h \in \mathcal{H}$,

$$|\varepsilon(h) - \hat{\varepsilon}(h)| \leq O\left(\sqrt{\frac{d}{m}\log\frac{m}{d} + \frac{1}{m}\log\frac{1}{\delta}}\right).$$

Thus, with probability at least $1 - \delta$, we also have that:

$$\varepsilon(\hat{h}) \leq \varepsilon(h^*) + O\left(\sqrt{\frac{d}{m}\log\frac{m}{d} + \frac{1}{m}\log\frac{1}{\delta}}\right).$$

In other words, if a hypothesis class has finite VC dimension, then uniform convergence occurs as m becomes large. As before, this allows us to give a bound on $\varepsilon(h)$ in terms of $\varepsilon(h^*)$. We also have the following corollary:

Corollary. For $|\varepsilon(h) - \hat{\varepsilon}(h)| \leq \gamma$ to hold for all $h \in \mathcal{H}$ (and hence $\varepsilon(\hat{h}) \leq \varepsilon(h^*) + 2\gamma$) with probability at least $1 - \delta$, it suffices that $m = O_{\gamma,\delta}(d)$.

In other words, the number of training examples needed to learn "well" using $\mathcal{H}$ is linear in the VC dimension of $\mathcal{H}$. It turns out that, for "most" hypothesis classes, the VC dimension (assuming a "reasonable" parameterization) is also roughly linear in the number of parameters. Putting these together, we conclude that (for an algorithm that tries to minimize training error) the number of training examples needed is usually roughly linear in the number of parameters of $\mathcal{H}$.

CS229 Lecture notes

Andrew Ng

Part VI
Regularization and model selection

Suppose we are trying select among several different models for a learning problem. For instance, we might be using a polynomial regression model $h_\theta(x) = g(\theta_0 + \theta_1 x + \theta_2 x^2 + \cdots + \theta_k x^k)$, and wish to decide if k should be 0, 1, ..., or 10. How can we automatically select a model that represents a good tradeoff between the twin evils of bias and variance[1]? Alternatively, suppose we want to automatically choose the bandwidth parameter τ for locally weighted regression, or the parameter C for our ℓ_1-regularized SVM. How can we do that?

For the sake of concreteness, in these notes we assume we have some finite set of models $\mathcal{M} = \{M_1, \ldots, M_d\}$ that we're trying to select among. For instance, in our first example above, the model M_i would be an i-th order polynomial regression model. (The generalization to infinite $\mathcal{M}$ is not hard.[2]) Alternatively, if we are trying to decide between using an SVM, a neural network or logistic regression, then $\mathcal{M}$ may contain these models.

[1] Given that we said in the previous set of notes that bias and variance are two very different beasts, some readers may be wondering if we should be calling them "twin" evils here. Perhaps it'd be better to think of them as non-identical twins. The phrase "the fraternal twin evils of bias and variance" doesn't have the same ring to it, though.

[2] If we are trying to choose from an infinite set of models, say corresponding to the possible values of the bandwidth $\tau \in \mathbb{R}^+$, we may discretize τ and consider only a finite number of possible values for it. More generally, most of the algorithms described here can all be viewed as performing optimization search in the space of models, and we can perform this search over infinite model classes as well.

1 Cross validation

Lets suppose we are, as usual, given a training set S. Given what we know about empirical risk minimization, here's what might initially seem like a algorithm, resulting from using empirical risk minimization for model selection:

1. Train each model M_i on S, to get some hypothesis h_i.
2. Pick the hypotheses with the smallest training error.

This algorithm does *not* work. Consider choosing the order of a polynomial. The higher the order of the polynomial, the better it will fit the training set S, and thus the lower the training error. Hence, this method will always select a high-variance, high-degree polynomial model, which we saw previously is often poor choice.

Here's an algorithm that works better. In **hold-out cross validation** (also called **simple cross validation**), we do the following:

1. Randomly split S into S_{train} (say, 70% of the data) and S_{cv} (the remaining 30%). Here, S_{cv} is called the hold-out cross validation set.
2. Train each model M_i on S_{train} only, to get some hypothesis h_i.
3. Select and output the hypothesis h_i that had the smallest error $\hat{\varepsilon}_{S_{\text{cv}}}(h_i)$ on the hold out cross validation set. (Recall, $\hat{\varepsilon}_{S_{\text{cv}}}(h)$ denotes the empirical error of h on the set of examples in S_{cv}.)

By testing on a set of examples S_{cv} that the models were not trained on, we obtain a better estimate of each hypothesis h_i's true generalization error, and can then pick the one with the smallest estimated generalization error. Usually, somewhere between $1/4 - 1/3$ of the data is used in the hold out cross validation set, and 30% is a typical choice.

Optionally, step 3 in the algorithm may also be replaced with selecting the model M_i according to $\arg\min_i \hat{\varepsilon}_{S_{\text{cv}}}(h_i)$, and then retraining M_i on the entire training set S. (This is often a good idea, with one exception being learning algorithms that are be very sensitive to perturbations of the initial conditions and/or data. For these methods, M_i doing well on S_{train} does not necessarily mean it will also do well on S_{cv}, and it might be better to forgo this retraining step.)

The disadvantage of using hold out cross validation is that it "wastes" about 30% of the data. Even if we were to take the optional step of retraining

the model on the entire training set, it's still as if we're trying to find a good model for a learning problem in which we had $0.7m$ training examples, rather than m training examples, since we're testing models that were trained on only $0.7m$ examples each time. While this is fine if data is abundant and/or cheap, in learning problems in which data is scarce (consider a problem with $m = 20$, say), we'd like to do something better.

Here is a method, called **k-fold cross validation**, that holds out less data each time:

1. Randomly split S into k disjoint subsets of m/k training examples each. Lets call these subsets $S_1, \ldots, S_k$.

2. For each model M_i, we evaluate it as follows:

   For $j = 1, \ldots, k$

   Train the model M_i on $S_1 \cup \cdots \cup S_{j-1} \cup S_{j+1} \cup \cdots S_k$ (i.e., train on all the data except S_j) to get some hypothesis h_{ij}.

   Test the hypothesis h_{ij} on S_j, to get $\hat{\varepsilon}_{S_j}(h_{ij})$.

   The estimated generalization error of model M_i is then calculated as the average of the $\hat{\varepsilon}_{S_j}(h_{ij})$'s (averaged over j).

3. Pick the model M_i with the lowest estimated generalization error, and retrain that model on the entire training set S. The resulting hypothesis is then output as our final answer.

A typical choice for the number of folds to use here would be $k = 10$. While the fraction of data held out each time is now $1/k$—much smaller than before—this procedure may also be more computationally expensive than hold-out cross validation, since we now need train to each model k times.

While $k = 10$ is a commonly used choice, in problems in which data is really scarce, sometimes we will use the extreme choice of $k = m$ in order to leave out as little data as possible each time. In this setting, we would repeatedly train on all but one of the training examples in S, and test on that held-out example. The resulting $m = k$ errors are then averaged together to obtain our estimate of the generalization error of a model. This method has its own name; since we're holding out one training example at a time, this method is called **leave-one-out cross validation.**

Finally, even though we have described the different versions of cross validation as methods for selecting a model, they can also be used more simply to evaluate a *single* model or algorithm. For example, if you have implemented

some learning algorithm and want to estimate how well it performs for your application (or if you have invented a novel learning algorithm and want to report in a technical paper how well it performs on various test sets), cross validation would give a reasonable way of doing so.

2 Feature Selection

One special and important case of model selection is called feature selection. To motivate this, imagine that you have a supervised learning problem where the number of features n is very large (perhaps $n \gg m$), but you suspect that there is only a small number of features that are "relevant" to the learning task. Even if you use the a simple linear classifier (such as the perceptron) over the n input features, the VC dimension of your hypothesis class would still be $O(n)$, and thus overfitting would be a potential problem unless the training set is fairly large.

In such a setting, you can apply a feature selection algorithm to reduce the number of features. Given n features, there are 2^n possible feature subsets (since each of the n features can either be included or excluded from the subset), and thus feature selection can be posed as a model selection problem over 2^n possible models. For large values of n, it's usually too expensive to explicitly enumerate over and compare all 2^n models, and so typically some heuristic search procedure is used to find a good feature subset. The following search procedure is called **forward search**:

1. Initialize $\mathcal{F} = \emptyset$.
2. Repeat {
   (a) For $i = 1, \ldots, n$ if $i \notin \mathcal{F}$, let $\mathcal{F}_i = \mathcal{F} \cup \{i\}$, and use some version of cross validation to evaluate features $\mathcal{F}_i$. (I.e., train your learning algorithm using only the features in $\mathcal{F}_i$, and estimate its generalization error.)
   (b) Set $\mathcal{F}$ to be the best feature subset found on step (a).
   }
3. Select and output the best feature subset that was evaluated during the entire search procedure.

The outer loop of the algorithm can be terminated either when $\mathcal{F} = \{1, \ldots, n\}$ is the set of all features, or when $|\mathcal{F}|$ exceeds some pre-set threshold (corresponding to the maximum number of features that you want the algorithm to consider using).

This algorithm described above one instantiation of **wrapper model feature selection**, since it is a procedure that "wraps" around your learning algorithm, and repeatedly makes calls to the learning algorithm to evaluate how well it does using different feature subsets. Aside from forward search, other search procedures can also be used. For example, **backward search** starts off with $\mathcal{F} = \{1, \ldots, n\}$ as the set of all features, and repeatedly deletes features one at a time (evaluating single-feature deletions in a similar manner to how forward search evaluates single-feature additions) until $\mathcal{F} = \emptyset$.

Wrapper feature selection algorithms often work quite well, but can be computationally expensive given how that they need to make many calls to the learning algorithm. Indeed, complete forward search (terminating when $\mathcal{F} = \{1, \ldots, n\}$) would take about $O(n^2)$ calls to the learning algorithm.

Filter feature selection methods give heuristic, but computationally much cheaper, ways of choosing a feature subset. The idea here is to compute some simple score $S(i)$ that measures how informative each feature x_i is about the class labels y. Then, we simply pick the k features with the largest scores $S(i)$.

One possible choice of the score would be define $S(i)$ to be (the absolute value of) the correlation between x_i and y, as measured on the training data. This would result in our choosing the features that are the most strongly correlated with the class labels. In practice, it is more common (particularly for discrete-valued features x_i) to choose $S(i)$ to be the **mutual information** $\mathrm{MI}(x_i, y)$ between x_i and y:

$$\mathrm{MI}(x_i, y) = \sum_{x_i \in \{0,1\}} \sum_{y \in \{0,1\}} p(x_i, y) \log \frac{p(x_i, y)}{p(x_i)p(y)}.$$

(The equation above assumes that x_i and y are binary-valued; more generally the summations would be over the domains of the variables.) The probabilities above $p(x_i, y)$, $p(x_i)$ and $p(y)$ can all be estimated according to their empirical distributions on the training set.

To gain intuition about what this score does, note that the mutual information can also be expressed as a Kullback-Leibler (KL) divergence:

$$\mathrm{MI}(x_i, y) = \mathrm{KL}\left(p(x_i, y) || p(x_i)p(y)\right)$$

You'll get to play more with KL-divergence in Problem set #3, but informally, this gives a measure of how different the probability distributions

$p(x_i, y)$ and $p(x_i)p(y)$ are. If x_i and y are independent random variables, then we would have $p(x_i, y) = p(x_i)p(y)$, and the KL-divergence between the two distributions will be zero. This is consistent with the idea if x_i and y are independent, then x_i is clearly very "non-informative" about y, and thus the score $S(i)$ should be small. Conversely, if x_i is very "informative" about y, then their mutual information $\mathrm{MI}(x_i, y)$ would be large.

One final detail: Now that you've ranked the features according to their scores $S(i)$, how do you decide how many features k to choose? Well, one standard way to do so is to use cross validation to select among the possible values of k. For example, when applying naive Bayes to text classification—a problem where n, the vocabulary size, is usually very large—using this method to select a feature subset often results in increased classifier accuracy.

3 Bayesian statistics and regularization

In this section, we will talk about one more tool in our arsenal for our battle against overfitting.

At the beginning of the quarter, we talked about parameter fitting using maximum likelihood (ML), and chose our parameters according to

$$\theta_{\mathrm{ML}} = \arg\max_{\theta} \prod_{i=1}^{m} p(y^{(i)}|x^{(i)}; \theta).$$

Throughout our subsequent discussions, we viewed θ as an unknown parameter of the world. This view of the θ as being *constant-valued but unknown* is taken in **frequentist** statistics. In the frequentist this view of the world, θ is not random—it just happens to be unknown—and it's our job to come up with statistical procedures (such as maximum likelihood) to try to estimate this parameter.

An alternative way to approach our parameter estimation problems is to take the **Bayesian** view of the world, and think of θ as being a *random variable* whose value is unknown. In this approach, we would specify a **prior distribution** $p(\theta)$ on θ that expresses our "prior beliefs" about the parameters. Given a training set $S = \{(x^{(i)}, y^{(i)})\}_{i=1}^{m}$, when we are asked to make a prediction on a new value of x, we can then compute the posterior

distribution on the parameters

$$\begin{aligned} p(\theta|S) &= \frac{p(S|\theta)p(\theta)}{p(S)} \\ &= \frac{\left(\prod_{i=1}^{m} p(y^{(i)}|x^{(i)},\theta)\right) p(\theta)}{\int_\theta \left(\prod_{i=1}^{m} p(y^{(i)}|x^{(i)},\theta)p(\theta)\right) d\theta} \end{aligned} \tag{1}$$

In the equation above, $p(y^{(i)}|x^{(i)},\theta)$ comes from whatever model you're using for your learning problem. For example, if you are using Bayesian logistic regression, then you might choose $p(y^{(i)}|x^{(i)},\theta) = h_\theta(x^{(i)})^{y^{(i)}}(1-h_\theta(x^{(i)}))^{(1-y^{(i)})}$, where $h_\theta(x^{(i)}) = 1/(1+\exp(-\theta^T x^{(i)}))$.[3]

When we are given a new test example x and asked to make it prediction on it, we can compute our posterior distribution on the class label using the posterior distribution on θ:

$$p(y|x,S) = \int_\theta p(y|x,\theta)p(\theta|S)d\theta \tag{2}$$

In the equation above, $p(\theta|S)$ comes from Equation (1). Thus, for example, if the goal is to the predict the expected value of y given x, then we would output[4]

$$\mathrm{E}[y|x,S] = \int_y y p(y|x,S)dy$$

The procedure that we've outlined here can be thought of as doing "fully Bayesian" prediction, where our prediction is computed by taking an average with respect to the posterior $p(\theta|S)$ over θ. Unfortunately, in general it is computationally very difficult to compute this posterior distribution. This is because it requires taking integrals over the (usually high-dimensional) θ as in Equation (1), and this typically cannot be done in closed-form.

Thus, in practice we will instead approximate the posterior distribution for θ. One common approximation is to replace our posterior distribution for θ (as in Equation 2) with a single point estimate. The **MAP (maximum a posteriori)** estimate for θ is given by

$$\theta_{\mathrm{MAP}} = \arg\max_\theta \prod_{i=1}^{m} p(y^{(i)}|x^{(i)},\theta)p(\theta). \tag{3}$$

[3]Since we are now viewing θ as a random variable, it is okay to condition on it value, and write "$p(y|x,\theta)$" instead of "$p(y|x;\theta)$."

[4]The integral below would be replaced by a summation if y is discrete-valued.

Note that this is the same formulas as for the ML (maximum likelihood) estimate for θ, except for the prior $p(\theta)$ term at the end.

In practical applications, a common choice for the prior $p(\theta)$ is to assume that $\theta \sim \mathcal{N}(0, \tau^2 I)$. Using this choice of prior, the fitted parameters θ_{MAP} will have smaller norm than that selected by maximum likelihood. (See Problem Set #3.) In practice, this causes the Bayesian MAP estimate to be less susceptible to overfitting than the ML estimate of the parameters. For example, Bayesian logistic regression turns out to be an effective algorithm for text classification, even though in text classification we usually have $n \gg m$.

CS229 Lecture notes

Andrew Ng

1 The perceptron and large margin classifiers

In this final set of notes on learning theory, we will introduce a different model of machine learning. Specifically, we have so far been considering **batch learning** settings in which we are first given a training set to learn with, and our hypothesis h is then evaluated on separate test data. In this set of notes, we will consider the **online learning** setting in which the algorithm has to make predictions continuously even while it's learning.

In this setting, the learning algorithm is given a sequence of examples $(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots (x^{(m)}, y^{(m)})$ in order. Specifically, the algorithm first sees $x^{(1)}$ and is asked to predict what it thinks $y^{(1)}$ is. After making its prediction, the true value of $y^{(1)}$ is revealed to the algorithm (and the algorithm may use this information to perform some learning). The algorithm is then shown $x^{(2)}$ and again asked to make a prediction, after which $y^{(2)}$ is revealed, and it may again perform some more learning. This proceeds until we reach $(x^{(m)}, y^{(m)})$. In the online learning setting, we are interested in the total number of errors made by the algorithm during this process. Thus, it models applications in which the algorithm has to make predictions even while it's still learning.

We will give a bound on the online learning error of the perceptron algorithm. To make our subsequent derivations easier, we will use the notational convention of denoting the class labels by $y =\in \{-1, 1\}$.

Recall that the perceptron algorithm has parameters $\theta \in \mathbb{R}^{n+1}$, and makes its predictions according to

$$h_\theta(x) = g(\theta^T x) \tag{1}$$

where

$$g(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ -1 & \text{if } z < 0. \end{cases}$$

Also, given a training example (x, y), the perceptron learning rule updates the parameters as follows. If $h_\theta(x) = y$, then it makes no change to the parameters. Otherwise, it performs the update[1]

$$\theta := \theta + yx.$$

The following theorem gives a bound on the online learning error of the perceptron algorithm, when it is run as an online algorithm that performs an update each time it gets an example wrong. Note that the bound below on the number of errors does not have an explicit dependence on the number of examples m in the sequence, or on the dimension n of the inputs (!).

Theorem (Block, 1962, and Novikoff, 1962). Let a sequence of examples $(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots (x^{(m)}, y^{(m)})$ be given. Suppose that $||x^{(i)}|| \leq D$ for all i, and further that there exists a unit-length vector u ($||u||_2 = 1$) such that $y^{(i)} \cdot (u^T x^{(i)}) \geq \gamma$ for all examples in the sequence (i.e., $u^T x^{(i)} \geq \gamma$ if $y^{(i)} = 1$, and $u^T x^{(i)} \leq -\gamma$ if $y^{(i)} = -1$, so that u separates the data with a margin of at least γ). Then the total number of mistakes that the perceptron algorithm makes on this sequence is at most $(D/\gamma)^2$.

Proof. The perceptron updates its weights only on those examples on which it makes a mistake. Let $\theta^{(k)}$ be the weights that were being used when it made its k-th mistake. So, $\theta^{(1)} = \vec{0}$ (since the weights are initialized to zero), and if the k-th mistake was on the example $(x^{(i)}, y^{(i)})$, then $g((x^{(i)})^T \theta^{(k)}) \neq y^{(i)}$, which implies that

$$(x^{(i)})^T \theta^{(k)} y^{(i)} \leq 0. \tag{2}$$

Also, from the perceptron learning rule, we would have that $\theta^{(k+1)} = \theta^{(k)} + y^{(i)} x^{(i)}$.

We then have

$$\begin{aligned} (\theta^{(k+1)})^T u &= (\theta^{(k)})^T u + y^{(i)} (x^{(i)})^T u \\ &\geq (\theta^{(k)})^T u + \gamma \end{aligned}$$

By a straightforward inductive argument, implies that

$$(\theta^{(k+1)})^T u \geq k\gamma. \tag{3}$$

[1]This looks slightly different from the update rule we had written down earlier in the quarter because here we have changed the labels to be $y \in \{-1, 1\}$. Also, the learning rate parameter α was dropped. The only effect of the learning rate is to scale all the parameters θ by some fixed constant, which does not affect the behavior of the perceptron.

Also, we have that

$$\begin{aligned} ||\theta^{(k+1)}||^2 &= ||\theta^{(k)} + y^{(i)}x^{(i)}||^2 \\ &= ||\theta^{(k)}||^2 + ||x^{(i)}||^2 + 2y^{(i)}(x^{(i)})^T\theta^{(i)} \\ &\leq ||\theta^{(k)}||^2 + ||x^{(i)}||^2 \\ &\leq ||\theta^{(k)}||^2 + D^2 \end{aligned} \tag{4}$$

The third step above used Equation (2). Moreover, again by applying a straightfoward inductive argument, we see that (4) implies

$$||\theta^{(k+1)}||^2 \leq kD^2. \tag{5}$$

Putting together (3) and (4) we find that

$$\begin{aligned} \sqrt{k}D &\geq ||\theta^{(k+1)}|| \\ &\geq (\theta^{(k+1)})^T u \\ &\geq k\gamma. \end{aligned}$$

The second inequality above follows from the fact that u is a unit-length vector (and $z^T u = ||z|| \cdot ||u|| \cos\phi \leq ||z|| \cdot ||u||$, where ϕ is the angle between z and u). Our result implies that $k \leq (D/\gamma)^2$. Hence, if the perceptron made a k-th mistake, then $k \leq (D/\gamma)^2$. $\square$

CS229 Lecture notes

Andrew Ng

The k-means clustering algorithm

In the clustering problem, we are given a training set $\{x^{(1)}, \ldots, x^{(m)}\}$, and want to group the data into a few cohesive "clusters." Here, $x^{(i)} \in \mathbb{R}^n$ as usual; but no labels $y^{(i)}$ are given. So, this is an unsupervised learning problem.

The k-means clustering algorithm is as follows:

1. Initialize **cluster centroids** $\mu_1, \mu_2, \ldots, \mu_k \in \mathbb{R}^n$ randomly.

2. Repeat until convergence: {

   For every i, set
   $$c^{(i)} := \arg\min_j ||x^{(i)} - \mu_j||^2.$$

   For each j, set
   $$\mu_j := \frac{\sum_{i=1}^m 1\{c^{(i)} = j\} x^{(i)}}{\sum_{i=1}^m 1\{c^{(i)} = j\}}.$$

   }

In the algorithm above, k (a parameter of the algorithm) is the number of clusters we want to find; and the cluster centroids μ_j represent our current guesses for the positions of the centers of the clusters. To initialize the cluster centroids (in step 1 of the algorithm above), we could choose k training examples randomly, and set the cluster centroids to be equal to the values of these k examples. (Other initialization methods are also possible.)

The inner-loop of the algorithm repeatedly carries out two steps: (i) "Assigning" each training example $x^{(i)}$ to the closest cluster centroid μ_j, and (ii) Moving each cluster centroid μ_j to the mean of the points assigned to it. Figure 1 shows an illustration of running k-means.

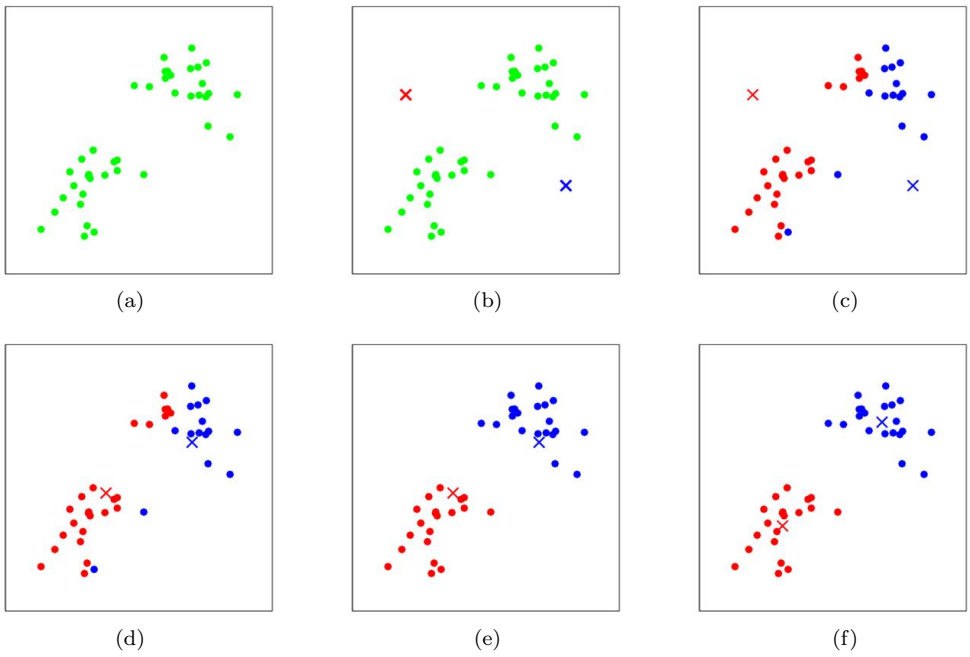


Figure 1: K-means algorithm. Training examples are shown as dots, and cluster centroids are shown as crosses. (a) Original dataset. (b) Random initial cluster centroids (in this instance, not chosen to be equal to two training examples). (c-f) Illustration of running two iterations of k-means. In each iteration, we assign each training example to the closest cluster centroid (shown by "painting" the training examples the same color as the cluster centroid to which is assigned); then we move each cluster centroid to the mean of the points assigned to it. (Best viewed in color.) Images courtesy Michael Jordan.

Is the k-means algorithm guaranteed to converge? Yes it is, in a certain sense. In particular, let us define the **distortion function** to be:

$$J(c, \mu) = \sum_{i=1}^{m} ||x^{(i)} - \mu_{c^{(i)}}||^2$$

Thus, J measures the sum of squared distances between each training example $x^{(i)}$ and the cluster centroid $\mu_{c^{(i)}}$ to which it has been assigned. It can be shown that k-means is exactly coordinate descent on J. Specifically, the inner-loop of k-means repeatedly minimizes J with respect to c while holding μ fixed, and then minimizes J with respect to μ while holding c fixed. Thus, J must monotonically decrease, and the value of J must converge. (Usually, this implies that c and μ will converge too. In theory, it is possible for

k-means to oscillate between a few different clusterings—i.e., a few different values for c and/or μ—that have exactly the same value of J, but this almost never happens in practice.)

The distortion function J is a non-convex function, and so coordinate descent on J is not guaranteed to converge to the global minimum. In other words, k-means can be susceptible to local optima. Very often k-means will work fine and come up with very good clusterings despite this. But if you are worried about getting stuck in bad local minima, one common thing to do is run k-means many times (using different random initial values for the cluster centroids μ_j). Then, out of all the different clusterings found, pick the one that gives the lowest distortion $J(c, \mu)$.

CS229 Lecture notes

Andrew Ng

Mixtures of Gaussians and the EM algorithm

In this set of notes, we discuss the EM (Expectation-Maximization) for density estimation.

Suppose that we are given a training set $\{x^{(1)}, \ldots, x^{(m)}\}$ as usual. Since we are in the unsupervised learning setting, these points do not come with any labels.

We wish to model the data by specifying a joint distribution $p(x^{(i)}, z^{(i)}) = p(x^{(i)}|z^{(i)})p(z^{(i)})$. Here, $z^{(i)} \sim \text{Multinomial}(\phi)$ (where $\phi_j \geq 0$, $\sum_{j=1}^{k} \phi_j = 1$, and the parameter ϕ_j gives $p(z^{(i)} = j)$,), and $x^{(i)}|z^{(i)} = j \sim \mathcal{N}(\mu_j, \Sigma_j)$. We let k denote the number of values that the $z^{(i)}$'s can take on. Thus, our model posits that each $x^{(i)}$ was generated by randomly choosing $z^{(i)}$ from $\{1, \ldots, k\}$, and then $x^{(i)}$ was drawn from one of k Gaussians depeneding on $z^{(i)}$. This is called the **mixture of Gaussians** model. Also, note that the $z^{(i)}$'s are **latent** random variables, meaning that they're hidden/unobserved. This is what will make our estimation problem difficult.

The parameters of our model are thus ϕ, ϕ and Σ. To estimate them, we can write down the likelihood of our data:

$$\begin{aligned}
\ell(\phi, \mu, \Sigma) &= \sum_{i=1}^{m} \log p(x^{(i)}; \phi, \mu, \Sigma) \\
&= \sum_{i=1}^{m} \log \sum_{z^{(i)}=1}^{k} p(x^{(i)}|z^{(i)}; \mu, \Sigma)p(z^{(i)}; \phi).
\end{aligned}$$

However, if we set to zero the derivatives of this formula with respect to the parameters and try to solve, we'll find that it is not possible to find the maximum likelihood estimates of the parameters in closed form. (Try this yourself at home.)

The random variables $z^{(i)}$ indicate which of the k Gaussians each $x^{(i)}$ had come from. Note that if we knew what the $z^{(i)}$'s were, the maximum

likelihood problem would have been easy. Specifically, we could then write down the likelihood as

$$\ell(\phi, \mu, \Sigma) = \sum_{i=1}^{m} \log p(x^{(i)}|z^{(i)}; \mu, \Sigma) + \log p(z^{(i)}; \phi).$$

Maximizing this with respect to ϕ, μ and Σ gives the parameters:

$$\begin{aligned}
\phi_j &= \frac{1}{m} \sum_{i=1}^{m} 1\{z^{(i)} = j\}, \\
\mu_j &= \frac{\sum_{i=1}^{m} 1\{z^{(i)} = j\} x^{(i)}}{\sum_{i=1}^{m} 1\{z^{(i)} = j\}}, \\
\Sigma_j &= \frac{\sum_{i=1}^{m} 1\{z^{(i)} = j\} (x^{(i)} - \mu_j)(x^{(i)} - \mu_j)^T}{\sum_{i=1}^{m} 1\{z^{(i)} = j\}}.
\end{aligned}$$

Indeed, we see that if the $z^{(i)}$'s were known, then maximum likelihood estimation becomes nearly identical to what we had when estimating the parameters of the Gaussian discriminant analysis model, except that here the $z^{(i)}$'s playing the role of the class labels.[1]

However, in our density estimation problem, the $z^{(i)}$'s are *not* known. What can we do?

The EM algorithm is an iterative algorithm that has two main steps. Applied to our problem, in the E-step, it tries to "guess" the values of the $z^{(i)}$'s. In the M-step, it updates the parameters of our model based on our guesses. Since in the M-step we are pretending that the guesses in the first part were correct, the maximization becomes easy. Here's the algorithm:

Repeat until convergence: {

(E-step) For each i, j, set

$$w_j^{(i)} := p(z^{(i)} = j|x^{(i)}; \phi, \mu, \Sigma)$$

[1]There are other minor differences in the formulas here from what we'd obtained in PS1 with Gaussian discriminant analysis, first because we've generalized the $z^{(i)}$'s to be multinomial rather than Bernoulli, and second because here we are using a different Σ_j for each Gaussian.

(M-step) Update the parameters:

$$\begin{aligned}
\phi_j &:= \frac{1}{m}\sum_{i=1}^{m} w_j^{(i)}, \\
\mu_j &:= \frac{\sum_{i=1}^{m} w_j^{(i)} x^{(i)}}{\sum_{i=1}^{m} w_j^{(i)}}, \\
\Sigma_j &:= \frac{\sum_{i=1}^{m} w_j^{(i)} (x^{(i)} - \mu_j)(x^{(i)} - \mu_j)^T}{\sum_{i=1}^{m} w_j^{(i)}}
\end{aligned}$$

}

In the E-step, we calculate the posterior probability of our parameters the $z^{(i)}$'s, given the $x^{(i)}$ and using the current setting of our parameters. I.e., using Bayes rule, we obtain:

$$p(z^{(i)} = j | x^{(i)}; \phi, \mu, \Sigma) = \frac{p(x^{(i)} | z^{(i)} = j; \mu, \Sigma) p(z^{(i)} = j; \phi)}{\sum_{l=1}^{k} p(x^{(i)} | z^{(i)} = l; \mu, \Sigma) p(z^{(i)} = l; \phi)}$$

Here, $p(x^{(i)}|z^{(i)} = j; \mu, \Sigma)$ is given by evaluating the density of a Gaussian with mean μ_j and covariance Σ_j at $x^{(i)}$; $p(z^{(i)} = j; \phi)$ is given by ϕ_j, and so on. The values $w_j^{(i)}$ calculated in the E-step represent our "soft" guesses[2] for the values of $z^{(i)}$.

Also, you should contrast the updates in the M-step with the formulas we had when the $z^{(i)}$'s were known exactly. They are identical, except that instead of the indicator functions "$1\{z^{(i)} = j\}$" indicating from which Gaussian each datapoint had come, we now instead have the $w_j^{(i)}$'s.

The EM-algorithm is also reminiscent of the K-means clustering algorithm, except that instead of the "hard" cluster assignments $c(i)$, we instead have the "soft" assignments $w_j^{(i)}$. Similar to K-means, it is also susceptible to local optima, so reinitializing at several different initial parameters may be a good idea.

It's clear that the EM algorithm has a very natural interpretation of repeatedly trying to guess the unknown $z^{(i)}$'s; but how did it come about, and can we make any guarantees about it, such as regarding its convergence? In the next set of notes, we will describe a more general view of EM, one

[2]The term "soft" refers to our guesses being probabilities and taking values in $[0, 1]$; in contrast, a "hard" guess is one that represents a single best guess (such as taking values in $\{0, 1\}$ or $\{1, \ldots, k\}$).

that will allow us to easily apply it to other estimation problems in which there are also latent variables, and which will allow us to give a convergence guarantee.

CS229 Lecture notes

Andrew Ng

Part IX
The EM algorithm

In the previous set of notes, we talked about the EM algorithm as applied to fitting a mixture of Gaussians. In this set of notes, we give a broader view of the EM algorithm, and show how it can be applied to a large family of estimation problems with latent variables. We begin our discussion with a very useful result called **Jensen's inequality**

1 Jensen's inequality

Let f be a function whose domain is the set of real numbers. Recall that f is a convex function if $f''(x) \geq 0$ (for all $x \in \mathbb{R}$). In the case of f taking vector-valued inputs, this is generalized to the condition that its hessian H is positive semi-definite ($H \geq 0$). If $f''(x) > 0$ for all x, then we say f is **strictly** convex (in the vector-valued case, the corresponding statement is that H must be strictly positive semi-definite, written $H > 0$). Jensen's inequality can then be stated as follows:

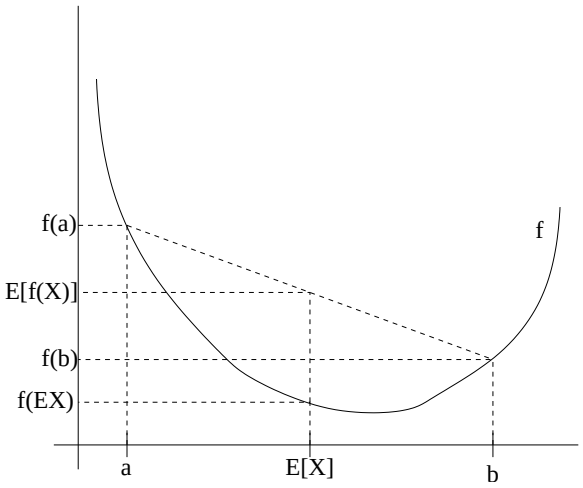
Theorem. Let f be a convex function, and let X be a random variable. Then:

$$\mathrm{E}[f(X)] \geq f(\mathrm{E}X).$$

Moreover, if f is strictly convex, then $\mathrm{E}[f(X)] = f(\mathrm{E}X)$ holds true if and only if $X = \mathrm{E}[X]$ with probability 1 (i.e., if X is a constant).

Recall our convention of occasionally dropping the parentheses when writing expectations, so in the theorem above, $f(\mathrm{E}X) = f(\mathrm{E}[X])$.

For an interpretation of the theorem, consider the figure below.



Here, f is a convex function shown by the solid line. Also, X is a random variable that has a 0.5 chance of taking the value a, and a 0.5 chance of taking the value b (indicated on the x-axis). Thus, the expected value of X is given by the midpoint between a and b.

We also see the values $f(a)$, $f(b)$ and $f(\mathrm{E}[X])$ indicated on the y-axis. Moreover, the value $\mathrm{E}[f(X)]$ is now the midpoint on the y-axis between $f(a)$ and $f(b)$. From our example, we see that because f is convex, it must be the case that $\mathrm{E}[f(X)] \geq f(\mathrm{E}X)$.

Incidentally, quite a lot of people have trouble remembering which way the inequality goes, and remembering a picture like this is a good way to quickly figure out the answer.

Remark. Recall that f is [strictly] concave if and only if $-f$ is [strictly] convex (i.e., $f''(x) \leq 0$ or $H \leq 0$). Jensen's inequality also holds for concave functions f, but with the direction of all the inequalities reversed ($\mathrm{E}[f(X)] \leq f(\mathrm{E}X)$, etc.).

2 The EM algorithm

Suppose we have an estimation problem in which we have a training set $\{x^{(1)}, \ldots, x^{(m)}\}$ consisting of m independent examples. We wish to fit the parameters of a model $p(x, z)$ to the data, where the likelihood is given by

$$\begin{aligned} \ell(\theta) &= \sum_{i=1}^{m} \log p(x; \theta) \\ &= \sum_{i=1}^{m} \log \sum_{z} p(x, z; \theta). \end{aligned}$$

But, explicitly finding the maximum likelihood estimates of the parameters θ may be hard. Here, the $z^{(i)}$'s are the latent random variables; and it is often the case that if the $z^{(i)}$'s were observed, then maximum likelihood estimation would be easy.

In such a setting, the EM algorithm gives an efficient method for maximum likelihood estimation. Maximizing $\ell(\theta)$ explicitly might be difficult, and our strategy will be to instead repeatedly construct a lower-bound on ℓ (E-step), and then optimize that lower-bound (M-step).

For each i, let Q_i be some distribution over the z's ($\sum_z Q_i(z) = 1$, $Q_i(z) \geq 0$). Consider the following:[1]

$$\begin{aligned}
\sum_i \log p(x^{(i)}; \theta) &= \sum_i \log \sum_{z^{(i)}} p(x^{(i)}, z^{(i)}; \theta) && (1)\\
&= \sum_i \log \sum_{z^{(i)}} Q_i(z^{(i)}) \frac{p(x^{(i)}, z^{(i)}; \theta)}{Q_i(z^{(i)})} && (2)\\
&\geq \sum_i \sum_{z^{(i)}} Q_i(z^{(i)}) \log \frac{p(x^{(i)}, z^{(i)}; \theta)}{Q_i(z^{(i)})} && (3)
\end{aligned}$$

The last step of this derivation used Jensen's inequality. Specifically, $f(x) = \log x$ is a concave function, since $f''(x) = -1/x^2 < 0$ over its domain $x \in \mathbb{R}^+$. Also, the term

$$\sum_{z^{(i)}} Q_i(z^{(i)}) \left[\frac{p(x^{(i)}, z^{(i)}; \theta)}{Q_i(z^{(i)})} \right]$$

in the summation is just an expectation of the quantity $\left[p(x^{(i)}, z^{(i)}; \theta)/Q_i(z^{(i)})\right]$ with respect to $z^{(i)}$ drawn according to the distribution given by Q_i. By Jensen's inequality, we have

$$f\left(\mathrm{E}_{z^{(i)} \sim Q_i} \left[\frac{p(x^{(i)}, z^{(i)}; \theta)}{Q_i(z^{(i)})} \right] \right) \geq \mathrm{E}_{z^{(i)} \sim Q_i} \left[f\left(\frac{p(x^{(i)}, z^{(i)}; \theta)}{Q_i(z^{(i)})} \right) \right],$$

where the "$z^{(i)} \sim Q_i$" subscripts above indicate that the expectations are with respect to $z^{(i)}$ drawn from Q_i. This allowed us to go from Equation (2) to Equation (3).

Now, for *any* set of distributions Q_i, the formula (3) gives a lower-bound on $\ell(\theta)$. There're many possible choices for the Q_i's. Which should we choose? Well, if we have some current guess θ of the parameters, it seems

[1] If z were continuous, then Q_i would be a density, and the summations over z in our discussion are replaced with integrals over z.

natural to try to make the lower-bound tight at that value of θ. I.e., we'll make the inequality above hold with equality at our particular value of θ. (We'll see later how this enables us to prove that $\ell(\theta)$ increases monotonically with successsive iterations of EM.)

To make the bound tight for a particular value of θ, we need for the step involving Jensen's inequality in our derivation above to hold with equality. For this to be true, we know it is sufficient that that the expectation be taken over a "constant"-valued random variable. I.e., we require that

$$\frac{p(x^{(i)}, z^{(i)}; \theta)}{Q_i(z^{(i)})} = c$$

for some constant c that does not depend on $z^{(i)}$. This is easily accomplished by choosing

$$Q_i(z^{(i)}) \propto p(x^{(i)}, z^{(i)}; \theta).$$

Actually, since we know $\sum_z Q_i(z^{(i)}) = 1$ (because it is a distribution), this further tells us that

$$\begin{aligned} Q_i(z^{(i)}) &= \frac{p(x^{(i)}, z^{(i)}; \theta)}{\sum_z p(x^{(i)}, z; \theta)} \\ &= \frac{p(x^{(i)}, z^{(i)}; \theta)}{p(x^{(i)}; \theta)} \\ &= p(z^{(i)}|x^{(i)}; \theta) \end{aligned}$$

Thus, we simply set the Q_i's to be the posterior distribution of the $z^{(i)}$'s given $x^{(i)}$ and the setting of the parameters θ.

Now, for this choice of the Q_i's, Equation (3) gives a lower-bound on the loglikelihood ℓ that we're trying to maximize. This is the E-step. In the M-step of the algorithm, we then maximize our formula in Equation (3) with respect to the parameters to obtain a new setting of the θ's. Repeatedly carrying out these two steps gives us the EM algorithm, which is as follows:

Repeat until convergence {

(E-step) For each i, set

$$Q_i(z^{(i)}) := p(z^{(i)}|x^{(i)}; \theta).$$

(M-step) Set

$$\theta := \arg\max_\theta \sum_i \sum_{z^{(i)}} Q_i(z^{(i)}) \log \frac{p(x^{(i)}, z^{(i)}; \theta)}{Q_i(z^{(i)})}.$$

}

How we we know if this algorithm will converge? Well, suppose $\theta^{(t)}$ and $\theta^{(t+1)}$ are the parameters from two successive iterations of EM. We will now prove that $\ell(\theta^{(t)}) \leq \ell(\theta^{(t+1)})$, which shows EM always monotonically improves the log-likelihood. The key to showing this result lies in our choice of the Q_i's. Specifically, on the iteration of EM in which the parameters had started out as $\theta^{(t)}$, we would have chosen $Q_i^{(t)}(z^{(i)}) := p(z^{(i)}|x^{(i)};\theta^{(t)})$. We saw earlier that this choice ensures that Jensen's inequality, as applied to get Equation (3), holds with equality, and hence

$$\ell(\theta^{(t)}) = \sum_i \sum_{z^{(i)}} Q_i^{(t)}(z^{(i)}) \log \frac{p(x^{(i)}, z^{(i)}; \theta^{(t)})}{Q_i^{(t)}(z^{(i)})}.$$

The parameters $\theta^{(t+1)}$ are then obtained by maximizing the right hand side of the equation above. Thus,

$$\begin{aligned}
\ell(\theta^{(t+1)}) &\geq \sum_i \sum_{z^{(i)}} Q_i^{(t)}(z^{(i)}) \log \frac{p(x^{(i)}, z^{(i)}; \theta^{(t+1)})}{Q_i^{(t)}(z^{(i)})} && (4) \\
&\geq \sum_i \sum_{z^{(i)}} Q_i^{(t)}(z^{(i)}) \log \frac{p(x^{(i)}, z^{(i)}; \theta^{(t)})}{Q_i^{(t)}(z^{(i)})} && (5) \\
&= \ell(\theta^{(t)}) && (6)
\end{aligned}$$

This first inequality comes from the fact that

$$\ell(\theta) \geq \sum_i \sum_{z^{(i)}} Q_i(z^{(i)}) \log \frac{p(x^{(i)}, z^{(i)}; \theta)}{Q_i(z^{(i)})}$$

holds for any values of Q_i and θ, and in particular holds for $Q_i = Q_i^{(t)}$, $\theta = \theta^{(t+1)}$. To get Equation (5), we used the fact that $\theta^{(t+1)}$ is chosen explicitly to be

$$\arg\max_\theta \sum_i \sum_{z^{(i)}} Q_i(z^{(i)}) \log \frac{p(x^{(i)}, z^{(i)}; \theta)}{Q_i(z^{(i)})},$$

and thus this formula evaluated at $\theta^{(t+1)}$ must be equal to or larger than the same formula evaluated at $\theta^{(t)}$. Finally, the step used to get (6) was shown earlier, and follows from $Q_i^{(t)}$ having been chosen to make Jensen's inequality hold with equality at $\theta^{(t)}$.

Hence, EM causes the likelihood to converge monotonically. In our description of the EM algorithm, we said we'd run it until convergence. Given the result that we just showed, one reasonable convergence test would be to check if the increase in $\ell(\theta)$ between successive iterations is smaller than some tolerance parameter, and to declare convergence if EM is improving $\ell(\theta)$ too slowly.

Remark. If we define

$$J(Q, \theta) = \sum_i \sum_{z^{(i)}} Q_i(z^{(i)}) \log \frac{p(x^{(i)}, z^{(i)}; \theta)}{Q_i(z^{(i)})},$$

the we know $\ell(\theta) \geq J(Q, \theta)$ from our previous derivation. The EM can also be viewed a coordinate ascent on J, in which the E-step maximizes it with respect to Q (check this yourself), and the M-step maximizes it with respect to θ.

3 Mixture of Gaussians revisited

Armed with our general definition of the EM algorithm, lets go back to our old example of fitting the parameters ϕ, μ and Σ in a mixture of Gaussians. For the sake of brevity, we carry out the derivations for the M-step updates only for ϕ and μ_j, and leave the updates for Σ_j as an exercise for the reader.

The E-step is easy. Following our algorithm derivation above, we simply calculate

$$w_j^{(i)} = Q_i(z^{(i)} = j) = P(z^{(i)} = j | x^{(i)}; \phi, \mu, \Sigma).$$

Here, "$Q_i(z^{(i)} = j)$" denotes the probability of $z^{(i)}$ taking the value j under the distribution Q_i.

Next, in the M-step, we need to maximize, with respect to our parameters ϕ, μ, Σ, the quantity

$$\begin{aligned}
&\sum_{i=1}^{m} \sum_{z^{(i)}} Q_i(z^{(i)}) \log \frac{p(x^{(i)}, z^{(i)}; \phi, \mu, \Sigma)}{Q_i(z^{(i)})} \\
&\quad = \sum_{i=1}^{m} \sum_{j=1}^{k} Q_i(z^{(i)} = j) \log \frac{p(x^{(i)} | z^{(i)} = j; \mu, \Sigma) p(z^{(i)} = j; \phi)}{Q_i(z^{(i)} = j)} \\
&\quad = \sum_{i=1}^{m} \sum_{j=1}^{k} w_j^{(i)} \log \frac{\frac{1}{(2\pi)^{n/2} |\Sigma_j|^{1/2}} \exp\left(-\frac{1}{2}(x^{(i)} - \mu_j)^T \Sigma_j^{-1} (x^{(i)} - \mu_j)\right) \cdot \phi_j}{w_j^{(i)}}
\end{aligned}$$

Lets maximize this with respect to μ_l. If we take the derivative with respect to μ_l, we find

$$\begin{aligned}
&\nabla_{\mu_l} \sum_{i=1}^{m} \sum_{j=1}^{k} w_j^{(i)} \log \frac{\frac{1}{(2\pi)^{n/2}|\Sigma_j|^{1/2}} \exp\left(-\frac{1}{2}(x^{(i)} - \mu_j)^T \Sigma_j^{-1}(x^{(i)} - \mu_j)\right) \cdot \phi_j}{w_j^{(i)}} \\
&= -\nabla_{\mu_l} \sum_{i=1}^{m} \sum_{j=1}^{k} w_j^{(i)} \frac{1}{2}(x^{(i)} - \mu_j)^T \Sigma_j^{-1}(x^{(i)} - \mu_j) \\
&= \frac{1}{2} \sum_{i=1}^{m} w_l^{(i)} \nabla_{\mu_l} 2\mu_l^T \Sigma_l^{-1} x^{(i)} - \mu_l^T \Sigma_l^{-1} \mu_l \\
&= \sum_{i=1}^{m} w_l^{(i)} \left(\Sigma_l^{-1} x^{(i)} - \Sigma_l^{-1} \mu_l\right)
\end{aligned}$$

Setting this to zero and solving for μ_l therefore yields the update rule

$$\mu_l := \frac{\sum_{i=1}^{m} w_l^{(i)} x^{(i)}}{\sum_{i=1}^{m} w_l^{(i)}},$$

which was what we had in the previous set of notes.

Lets do one more example, and derive the M-step update for the parameters ϕ_j. Grouping together only the terms that depend on ϕ_j, we find that we need to maximize

$$\sum_{i=1}^{m} \sum_{j=1}^{k} w_j^{(i)} \log \phi_j.$$

However, there is an additional constraint that the ϕ_j's sum to 1, since they represent the probabilities $\phi_j = p(z^{(i)} = j; \phi)$. To deal with the constraint that $\sum_{j=1}^{k} \phi_j = 1$, we construct the Lagrangian

$$\mathcal{L}(\phi) = \sum_{i=1}^{m} \sum_{j=1}^{k} w_j^{(i)} \log \phi_j + \beta(\sum_{j=1}^{k} \phi_j - 1),$$

where β is the Lagrange multiplier.[2] Taking derivatives, we find

$$\frac{\partial}{\partial \phi_j} \mathcal{L}(\phi) = \sum_{i=1}^{m} \frac{w_j^{(i)}}{\phi_j} + 1$$

[2]We don't need to worry about the constraint that $\phi_j \geq 0$, because as we'll shortly see, the solution we'll find from this derivation will automatically satisfy that anyway.

Setting this to zero and solving, we get

$$\phi_j = \frac{\sum_{i=1}^{m} w_j^{(i)}}{-\beta}$$

I.e., $\phi_j \propto \sum_{i=1}^{m} w_j^{(i)}$. Using the constraint that $\sum_j \phi_j = 1$, we easily find that $-\beta = \sum_{i=1}^{m} \sum_{j=1}^{k} w_j^{(i)} = \sum_{i=1}^{m} 1 = m$. (This used the fact that $w_j^{(i)} = Q_i(z^{(i)} = j)$, and since probabilities sum to 1, $\sum_j w_j^{(i)} = 1$.) We therefore have our M-step updates for the parameters ϕ_j:

$$\phi_j := \frac{1}{m} \sum_{i=1}^{m} w_j^{(i)}.$$

The derivation for the M-step updates to Σ_j are also entirely straightforward.

CS229 Lecture notes

Andrew Ng

Part X
Factor analysis

When we have data $x^{(i)} \in \mathbb{R}^n$ that comes from a mixture of several Gaussians, the EM algorithm can be applied to fit a mixture model. In this setting, we usually imagine problems were the we have sufficient data to be able to discern the multiple-Gaussian structure in the data. For instance, this would be the case if our training set size m was significantly larger than the dimension n of the data.

Now, consider a setting in which $n \gg m$. In such a problem, it might be difficult to model the data even with a single Gaussian, much less a mixture of Gaussian. Specifically, since the m data points span only a low-dimensional subspace of $\mathbb{R}^n$, if we model the data as Gaussian, and estimate the mean and covariance using the usual maximum likelihood estimators,

$$\begin{aligned}\mu &= \frac{1}{m}\sum_{i=1}^{m} x^{(i)} \\ \Sigma &= \frac{1}{m}\sum_{i=1}^{m}(x^{(i)}-\mu)(x^{(i)}-\mu)^T,\end{aligned}$$

we would find that the matrix Σ is singular. This means that Σ^{-1} does not exist, and $1/|\Sigma|^{1/2} = 1/0$. But both of these terms are needed in computing the usual density of a multivariate Gaussian distribution. Another way of stating this difficulty is that maximum likelihood estimates of the parameters result in a Gaussian that places all of its probability in the affine space spanned by the data,[1] and this corresponds to a singular covariance matrix.

[1] This is the set of points x satisfying $x = \sum_{i=1}^{m} \alpha_i x^{(i)}$, for some α_i's so that $\sum_{i=1}^{m} \alpha_1 = 1$.

More generally, unless m exceeds n by some reasonable amount, the maximum likelihood estimates of the mean and covariance may be quite poor. Nonetheless, we would still like to be able to fit a reasonable Gaussian model to the data, and perhaps capture some interesting covariance structure in the data. How can we do this?

In the next section, we begin by reviewing two possible restrictions on Σ, ones that allow us to fit Σ with small amounts of data but neither of which will give a satisfactory solution to our problem. We next discuss some properties of Gaussians that will be needed later; specifically, how to find marginal and conditonal distributions of Gaussians. Finally, we present the factor analysis model, and EM for it.

1 Restrictions of Σ

If we do not have sufficient data to fit a full covariance matrix, we may place some restrictions on the space of matrices Σ that we will consider. For instance, we may choose to fit a covariance matrix Σ that is diagonal. In this setting, the reader may easily verify that the maximum likelihood estimate of the covariance matrix is given by the diagonal matrix Σ satisfying

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

Thus, Σ_{jj} is just the empirical estimate of the variance of the j-th coordinate of the data.

Recall that the contours of a Gaussian density are ellipses. A diagonal Σ corresponds to a Gaussian where the major axes of these ellipses are axis-aligned.

Sometimes, we may place a further restriction on the covariance matrix that not only must it be diagonal, but its diagonal entries must all be equal. In this setting, we have $\Sigma = \sigma^2 I$, where σ^2 is the parameter under our control. The maximum likelihood estimate of σ^2 can be found to be:

$$\sigma^2 = \frac{1}{mn}\sum_{j=1}^{n}\sum_{i=1}^{m}(x_j^{(i)} - \mu_j)^2.$$

This model corresponds to using Gaussians whose densities have contours that are circles (in 2 dimesions; or spheres/hyperspheres in higher dimensions).

If we were fitting a full, unconstrained, covariance matrix Σ to data, it was necessary that $m \geq n+1$ in order for the maximum likelihood estimate of Σ not to be singular. Under either of the two restrictions above, we may obtain non-singular Σ when $m \geq 2$.

However, restricting Σ to be diagonal also means modeling the different coordinates x_i, x_j of the data as being uncorrelated and independent. Often, it would be nice to be able to capture some interesting correlation structure in the data. If we were to use either of the restrictions on Σ described above, we would therefore fail to do so. In this set of notes, we will describe the factor analysis model, which uses more parameters than the diagonal Σ and captures some correlations in the data, but also without having to fit a full covariance matrix.

2 Marginals and conditionals of Gaussians

Before describing factor analysis, we digress to talk about how to find conditional and marginal distributions of random variables with a joint multivariate Gaussian distribution.

Suppose we have a vector-valued random variable

$$x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix},$$

where $x_1 \in \mathbb{R}^r$, $x_2 \in \mathbb{R}^s$, and $x \in \mathbb{R}^{r+s}$. Suppose $x \sim \mathcal{N}(\mu, \Sigma)$, where

$$\mu = \begin{bmatrix} \mu_1 \\ \mu_2 \end{bmatrix}, \quad \Sigma = \begin{bmatrix} \Sigma_{11} & \Sigma_{12} \\ \Sigma_{21} & \Sigma_{22} \end{bmatrix}.$$

Here, $\mu_1 \in \mathbb{R}^r$, $\mu_2 \in \mathbb{R}^s$, $\Sigma_{11} \in \mathbb{R}^{r \times r}$, $\Sigma_{12} \in \mathbb{R}^{r \times s}$, and so on. Note that since covariance matrices are symmetric, $\Sigma_{12} = \Sigma_{21}^T$.

Under our assumptions, x_1 and x_2 are jointly multivariate Gaussian. What is the marginal distribution of x_1? It is not hard to see that $\mathrm{E}[x_1] = \mu_1$, and that $\mathrm{Cov}(x_1) = \mathrm{E}[(x_1 - \mu_1)(x_1 - \mu_1)] = \Sigma_{11}$. To see that the latter is true, note that by definition of the joint covariance of x_1 and x_2, we have

that

$$\begin{aligned}
\mathrm{Cov}(x) &= \Sigma \\
&= \begin{bmatrix} \Sigma_{11} & \Sigma_{12} \\ \Sigma_{21} & \Sigma_{22} \end{bmatrix} \\
&= \mathrm{E}[(x-\mu)(x-\mu)^T] \\
&= \mathrm{E}\left[\begin{pmatrix} x_1-\mu_1 \\ x_2-\mu_2 \end{pmatrix} \begin{pmatrix} x_1-\mu_1 \\ x_2-\mu_2 \end{pmatrix}^T \right] \\
&= \mathrm{E}\begin{bmatrix} (x_1-\mu_1)(x_1-\mu_1)^T & (x_1-\mu_1)(x_2-\mu_2)^T \\ (x_2-\mu_2)(x_1-\mu_1)^T & (x_2-\mu_2)(x_2-\mu_2)^T \end{bmatrix}.
\end{aligned}$$

Matching the upper-left subblocks in the matrices in the second and the last lines above gives the result.

Since marginal distributions of Gaussians are themselves Gaussian, we therefore have that the marginal distribution of x_1 is given by $x_1 \sim \mathcal{N}(\mu_1, \Sigma_{11})$.

Also, we can ask, what is the conditional distribution of x_1 given x_2? By referring to the definition of the multivariate Gaussian distribution, it can be shown that $x_1|x_2 \sim \mathcal{N}(\mu_{1|2}, \Sigma_{1|2})$, where

$$\begin{aligned}
\mu_{1|2} &= \mu_1 + \Sigma_{12}\Sigma_{22}^{-1}(x_2-\mu_2), &(1)\\
\Sigma_{1|2} &= \Sigma_{11} - \Sigma_{12}\Sigma_{22}^{-1}\Sigma_{21}. &(2)
\end{aligned}$$

When working with the factor analysis model in the next section, these formulas for finding conditional and marginal distributions of Gaussians will be very useful.

3 The Factor analysis model

In the factor analysis model, we posit a joint distribution on (x, z) as follows, where $z \in \mathbb{R}^k$ is a latent random variable:

$$\begin{aligned}
z &\sim \mathcal{N}(0, I) \\
x|z &\sim \mathcal{N}(\mu + \Lambda z, \Psi).
\end{aligned}$$

Here, the parameters of our model are the vector $\mu \in \mathbb{R}^n$, the matrix $\Lambda \in \mathbb{R}^{n\times k}$, and the diagonal matrix $\Psi \in \mathbb{R}^{n\times n}$. The value of k is usually chosen to be smaller than n.

Thus, we imagine that each datapoint $x^{(i)}$ is generated by sampling a k dimension multivariate Gaussian $z^{(i)}$. Then, it is mapped to a k-dimensional affine space of $\mathbb{R}^n$ by computing $\mu + \Lambda z^{(i)}$. Lastly, $x^{(i)}$ is generated by adding covariance Ψ noise to $\mu + \Lambda z^{(i)}$.

Equivalently (convince yourself that this is the case), we can therefore also define the factor analysis model according to

$$\begin{aligned} z &\sim \mathcal{N}(0, I) \\ \epsilon &\sim \mathcal{N}(0, \Psi) \\ x &= \mu + \Lambda z + \epsilon. \end{aligned}$$

where ϵ and z are independent.

Lets work out exactly what distribution our model defines. Our random variables z and x have a joint Gaussian distribution

$$\begin{bmatrix} z \\ x \end{bmatrix} \sim \mathcal{N}(\mu_{zx}, \Sigma).$$

We will now find μ_{zx} and Σ.

We know that $\mathrm{E}[z] = 0$, from the fact that $z \sim \mathcal{N}(0, I)$. Also, we have that

$$\begin{aligned} \mathrm{E}[x] &= \mathrm{E}[\mu + \Lambda z + \epsilon] \\ &= \mu + \Lambda \mathrm{E}[z] + \mathrm{E}[\epsilon] \\ &= \mu. \end{aligned}$$

Putting these together, we obtain

$$\mu_{zx} = \begin{bmatrix} \vec{0} \\ \mu \end{bmatrix}$$

Next, to find, Σ, we need to calculate $\Sigma_{zz} = \mathrm{E}[(z - \mathrm{E}[z])(z - \mathrm{E}[z])^T]$ (the upper-left block of Σ), $\Sigma_{zx} = \mathrm{E}[(z - \mathrm{E}[z])(x - \mathrm{E}[x])^T]$ (upper-right block), and $\Sigma_{xx} = \mathrm{E}[(x - \mathrm{E}[x])(x - \mathrm{E}[x])^T]$ (lower-right block).

Now, since $z \sim \mathcal{N}(0, I)$, we easily find that $\Sigma_{zz} = \mathrm{Cov}(z) = I$. Also,

$$\begin{aligned} \mathrm{E}[(z - \mathrm{E}[z])(x - \mathrm{E}[x])^T] &= \mathrm{E}[z(\mu + \Lambda z + \epsilon - \mu)^T] \\ &= \mathrm{E}[zz^T]\Lambda^T + \mathrm{E}[z\epsilon^T] \\ &= \Lambda^T. \end{aligned}$$

In the last step, we used the fact that $\mathrm{E}[zz^T] = \mathrm{Cov}(z)$ (since z has zero mean), and $\mathrm{E}[z\epsilon^T] = \mathrm{E}[z]\mathrm{E}[\epsilon^T] = 0$ (since z and ϵ are independent, and

hence the expectation of their product is the product of their expectations). Similarly, we can find Σ_{xx} as follows:

$$\begin{aligned}
\mathrm{E}[(x - \mathrm{E}[x])(x - \mathrm{E}[x])^T] &= \mathrm{E}[(\mu + \Lambda z + \epsilon - \mu)(\mu + \Lambda z + \epsilon - \mu)^T] \\
&= \mathrm{E}[\Lambda z z^T \Lambda^T + \epsilon z^T \Lambda^T + \Lambda z \epsilon^T + \epsilon \epsilon^T] \\
&= \Lambda \mathrm{E}[z z^T] \Lambda^T + \mathrm{E}[\epsilon \epsilon^T] \\
&= \Lambda \Lambda^T + \Psi.
\end{aligned}$$

Putting everything together, we therefore have that

$$\begin{bmatrix} z \\ x \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} \vec{0} \\ \mu \end{bmatrix}, \begin{bmatrix} I & \Lambda^T \\ \Lambda & \Lambda\Lambda^T + \Psi \end{bmatrix} \right). \tag{3}$$

Hence, we also see that the marginal distribution of x is given by $x \sim \mathcal{N}(\mu, \Lambda\Lambda^T + \Psi)$. Thus, given a training set $\{x^{(i)}; i = 1, \ldots, m\}$, we can write down the log likelihood of the parameters:

$$\ell(\mu, \Lambda, \Psi) = \log \prod_{i=1}^{m} \frac{1}{(2\pi)^{n/2} |\Lambda\Lambda^T + \Psi|} \exp\left(-\frac{1}{2} (x^{(i)} - \mu)^T (\Lambda\Lambda^T + \Psi)^{-1} (x^{(i)} - \mu) \right).$$

To perform maximum likelihood estimation, we would like to maximize this quantity with respect to the parameters. But maximizing this formula explicitly is hard (try it yourself), and we are aware of no algorithm that does so in closed-form. So, we will instead use to the EM algorithm. In the next section, we derive EM for factor analysis.

4 EM for factor analysis

The derivation for the E-step is easy. We need to compute $Q_i(z^{(i)}) = p(z^{(i)}|x^{(i)}; \mu, \Lambda, \Psi)$. By substituting the distribution given in Equation (3) into the formulas (1-2) used for finding the conditional distribution of a Gaussian, we find that $z^{(i)}|x^{(i)}; \mu, \Lambda, \Psi \sim \mathcal{N}(\mu_{z^{(i)}|x^{(i)}}, \Sigma_{z^{(i)}|x^{(i)}})$, where

$$\begin{aligned}
\mu_{z^{(i)}|x^{(i)}} &= \Lambda^T (\Lambda\Lambda^T + \Psi)^{-1} (x^{(i)} - \mu), \\
\Sigma_{z^{(i)}|x^{(i)}} &= I - \Lambda^T (\Lambda\Lambda^T + \Psi)^{-1} \Lambda.
\end{aligned}$$

So, using these definitions for $\mu_{z^{(i)}|x^{(i)}}$ and $\Sigma_{z^{(i)}|x^{(i)}}$, we have

$$Q_i(z^{(i)}) = \frac{1}{(2\pi)^{k/2} |\Sigma_{z^{(i)}|x^{(i)}}|^{1/2}} \exp\left(-\frac{1}{2} (z^{(i)} - \mu_{z^{(i)}|x^{(i)}})^T \Sigma_{z^{(i)}|x^{(i)}}^{-1} (z^{(i)} - \mu_{z^{(i)}|x^{(i)}}) \right).$$

Lets now work out the M-step. Here, we need to maximize

$$\sum_{i=1}^{m} \int_{z^{(i)}} Q_i(z^{(i)}) \log \frac{p(x^{(i)}, z^{(i)}; \mu, \Lambda, \Psi)}{Q_i(z^{(i)})} dz^{(i)} \tag{4}$$

with respect to the parameters μ, Λ, Ψ. We will work out only the optimization with respect to Λ, and leave the derivations of the updates for μ and Ψ as an exercise to the reader.

We can simplify Equation (4) as follows:

$$\sum_{i=1}^{m} \int_{z^{(i)}} Q_i(z^{(i)}) \left[\log p(x^{(i)}|z^{(i)}; \mu, \Lambda, \Psi) + \log p(z^{(i)}) - \log Q_i(z^{(i)})\right] dz^{(i)} \tag{5}$$

$$= \sum_{i=1}^{m} \mathrm{E}_{z^{(i)} \sim Q_i} \left[\log p(x^{(i)}|z^{(i)}; \mu, \Lambda, \Psi) + \log p(z^{(i)}) - \log Q_i(z^{(i)})\right] \tag{6}$$

Here, the "$z^{(i)} \sim Q_i$" subscript indicates that the expectation is with respect to $z^{(i)}$ drawn from Q_i. In the subsequent development, we will omit this subscript when there is no risk of ambiguity. Dropping terms that do not depend on the parameters, we find that we need to maximize:

$$\sum_{i=1}^{m} \mathrm{E}\left[\log p(x^{(i)}|z^{(i)}; \mu, \Lambda, \Psi)\right]$$

$$= \sum_{i=1}^{m} \mathrm{E}\left[\log \frac{1}{(2\pi)^{n/2}|\Psi|^{1/2}} \exp\left(-\frac{1}{2}(x^{(i)} - \mu - \Lambda z^{(i)})^T \Psi^{-1} (x^{(i)} - \mu - \Lambda z^{(i)})\right)\right]$$

$$= \sum_{i=1}^{m} \mathrm{E}\left[-\frac{1}{2}\log|\Psi| - \frac{n}{2}\log(2\pi) - \frac{1}{2}(x^{(i)} - \mu - \Lambda z^{(i)})^T \Psi^{-1} (x^{(i)} - \mu - \Lambda z^{(i)})\right]$$

Lets maximize this with respect to Λ. Only the last term above depends on Λ. Taking derivatives, and using the facts that $\mathrm{tr}\, a = a$ (for $a \in \mathbb{R}$), $\mathrm{tr}AB = \mathrm{tr}BA$, and $\nabla_A \mathrm{tr}ABA^TC = CAB + C^TAB$, we get:

$$\nabla_\Lambda \sum_{i=1}^{m} -\mathrm{E}\left[\frac{1}{2}(x^{(i)} - \mu - \Lambda z^{(i)})^T \Psi^{-1} (x^{(i)} - \mu - \Lambda z^{(i)})\right]$$

$$= \sum_{i=1}^{m} \nabla_\Lambda \mathrm{E}\left[-\mathrm{tr}\frac{1}{2} z^{(i)T} \Lambda^T \Psi^{-1} \Lambda z^{(i)} + \mathrm{tr} z^{(i)T} \Lambda^T \Psi^{-1} (x^{(i)} - \mu)\right]$$

$$= \sum_{i=1}^{m} \nabla_\Lambda \mathrm{E}\left[-\mathrm{tr}\frac{1}{2} \Lambda^T \Psi^{-1} \Lambda z^{(i)} z^{(i)T} + \mathrm{tr}\Lambda^T \Psi^{-1} (x^{(i)} - \mu) z^{(i)T}\right]$$

$$= \sum_{i=1}^{m} \mathrm{E}\left[-\Psi^{-1} \Lambda z^{(i)} z^{(i)T} + \Psi^{-1} (x^{(i)} - \mu) z^{(i)T}\right]$$

Setting this to zero and simplifying, we get:

$$\sum_{i=1}^{m} \Lambda \mathrm{E}_{z^{(i)}\sim Q_i}\left[z^{(i)}z^{(i)T}\right] = \sum_{i=1}^{m}(x^{(i)}-\mu)\mathrm{E}_{z^{(i)}\sim Q_i}\left[z^{(i)T}\right].$$

Hence, solving for Λ, we obtain

$$\Lambda = \left(\sum_{i=1}^{m}(x^{(i)}-\mu)\mathrm{E}_{z^{(i)}\sim Q_i}\left[z^{(i)T}\right]\right)\left(\sum_{i=1}^{m}\mathrm{E}_{z^{(i)}\sim Q_i}\left[z^{(i)}z^{(i)T}\right]\right)^{-1}. \quad (7)$$

It is interesting to note the close relationship between this equation and the normal equation that we'd derived for least squares regression,

$$\text{“}\theta^T = (y^T X)(X^T X)^{-1}.\text{”}$$

The analogy is that here, the x's are a linear function of the z's (plus noise). Given the "guesses" for z that the E-step has found, we will now try to estimate the unknown linearity Λ relating the x's and z's. It is therefore no surprise that we obtain something similar to the normal equation. There is, however, one important difference between this and an algorithm that performs least squares using just the "best guesses" of the z's; we will see this difference shortly.

To complete our M-step update, lets work out the values of the expectations in Equation (7). From our definition of Q_i being Gaussian with mean $\mu_{z^{(i)}|x^{(i)}}$ and covariance $\Sigma_{z^{(i)}|x^{(i)}}$, we easily find

$$\begin{aligned}\mathrm{E}_{z^{(i)}\sim Q_i}\left[z^{(i)T}\right] &= \mu^T_{z^{(i)}|x^{(i)}} \\ \mathrm{E}_{z^{(i)}\sim Q_i}\left[z^{(i)}z^{(i)T}\right] &= \mu_{z^{(i)}|x^{(i)}}\mu^T_{z^{(i)}|x^{(i)}} + \Sigma_{z^{(i)}|x^{(i)}}.\end{aligned}$$

The latter comes from the fact that, for a random variable Y, $\mathrm{Cov}(Y) = \mathrm{E}[YY^T] - \mathrm{E}[Y]\mathrm{E}[Y]^T$, and hence $\mathrm{E}[YY^T] = \mathrm{E}[Y]\mathrm{E}[Y]^T + \mathrm{Cov}(Y)$. Substituting this back into Equation (7), we get the M-step update for Λ:

$$\Lambda = \left(\sum_{i=1}^{m}(x^{(i)}-\mu)\mu^T_{z^{(i)}|x^{(i)}}\right)\left(\sum_{i=1}^{m}\mu_{z^{(i)}|x^{(i)}}\mu^T_{z^{(i)}|x^{(i)}} + \Sigma_{z^{(i)}|x^{(i)}}\right)^{-1}. \quad (8)$$

It is important to note the presence of the $\Sigma_{z^{(i)}|x^{(i)}}$ on the right hand side of this equation. This is the covariance in the posterior distribution $p(z^{(i)}|x^{(i)})$ of $z^{(i)}$ give $x^{(i)}$, and the M-step must take into account this uncertainty

about $z^{(i)}$ in the posterior. A common mistake in deriving EM is to assume that in the E-step, we need to calculate only expectation $E[z]$ of the latent random variable z, and then plug that into the optimization in the M-step everywhere z occurs. While this worked for simple problems such as the mixture of Gaussians, in our derivation for factor analysis, we needed $E[zz^T]$ as well E[z]; and as we saw, $E[zz^T]$ and E[z]E[z]^T differ by the quantity $\Sigma_{z|x}$. Thus, the M-step update must take into account the covariance of z in the posterior distribution $p(z^{(i)}|x^{(i)})$.

Lastly, we can also find the M-step optimizations for the parameters μ and Ψ. It is not hard to show that the first is given by

$$\mu = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}.$$

Since this doesn't change as the parameters are varied (i.e., unlike the update for Λ, the right hand side does not depend on $Q_i(z^{(i)}) = p(z^{(i)}|x^{(i)}; \mu, \Lambda, \Psi)$, which in turn depends on the parameters), this can be calculated just once and needs not be further updated as the algorithm is run. Similarly, the diagonal Ψ can be found by calculating

$$\Phi = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}x^{(i)^T} - x^{(i)}\mu_{z^{(i)}|x^{(i)}}^T\Lambda^T - \Lambda\mu_{z^{(i)}|x^{(i)}}x^{(i)^T} + \Lambda(\mu_{z^{(i)}|x^{(i)}}\mu_{z^{(i)}|x^{(i)}}^T + \Sigma_{z^{(i)}|x^{(i)}})\Lambda^T,$$

and setting $\Psi_{ii} = \Phi_{ii}$ (i.e., letting Ψ be the diagonal matrix containing only the diagonal entries of Φ).

CS229 Lecture notes

Andrew Ng

Part XI

Principal components analysis

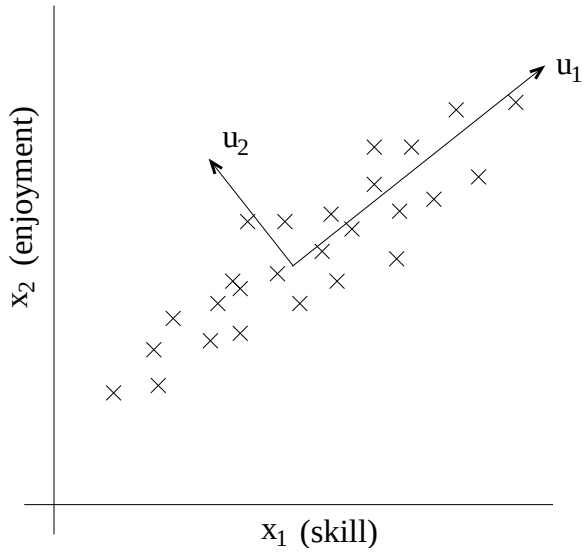
In our discussion of factor analysis, we gave a way to model data $x \in \mathbb{R}^n$ as "approximately" lying in some k-dimension subspace, where $k \ll n$. Specifically, we imagined that each point $x^{(i)}$ was created by first generating some $z^{(i)}$ lying in the k-dimension affine space $\{\Lambda z + \mu; z \in \mathbb{R}^k\}$, and then adding Ψ-covariance noise. Factor analysis is based on a probabilistic model, and parameter estimation used the iterative EM algorithm.

In this set of notes, we will develop a method, Principal Components Analysis (PCA), that also tries to identify the subspace in which the data approximately lies. However, PCA will do so more directly, and will require only an eigenvector calculation (easily done with the `eig` function in Matlab), and does not need to resort to EM.

Suppose we are given dataset $\{x^{(i)}; i = 1, \ldots, m\}$ of attributes of m different types of automobiles, such as their maximum speed, turn radius, and so on. Lets $x^{(i)} \in \mathbb{R}^n$ for each i ($n \ll m$). But unknown to us, two different attributes—some x_i and x_j—respectively give a car's maximum speed measured in miles per hour, and the maximum speed measured in kilometers per hour. These two attributes are therefore almost linearly dependent, up to only small differences introduced by rounding off to the nearest mph or kph. Thus, the data really lies approximately on an $n - 1$ dimensional subspace. How can we automatically detect, and perhaps remove, this redundancy?

For a less contrived example, consider a dataset resulting from a survey of pilots for radio-controlled helicopters, where $x_1^{(i)}$ is a measure of the piloting skill of pilot i, and $x_2^{(i)}$ captures how much he/she enjoys flying. Because RC helicopters are very difficult to fly, only the most committed students, ones that truly enjoy flying, become good pilots. So, the two attributes x_1 and x_2 are strongly correlated. Indeed, we might posit that that the

data actually likes along some diagonal axis (the u_1 direction) capturing the intrinsic piloting "karma" of a person, with only a small amount of noise lying off this axis. (See figure.) How can we automatically compute this u_1 direction?



We will shortly develop the PCA algorithm. But prior to running PCA per se, typically we first pre-process the data to normalize its mean and variance, as follows:

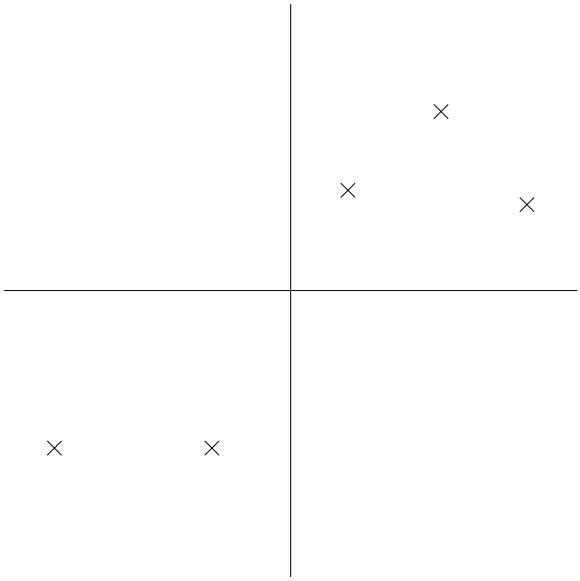
1. Let $\mu = \frac{1}{m} \sum_{i=1}^{m} x^{(i)}$.
2. Replace each $x^{(i)}$ with $x^{(i)} - \mu$.
3. Let $\sigma_j^2 = \frac{1}{m} \sum_i (x_j^{(i)})^2$
4. Replace each $x_j^{(i)}$ with $x_j^{(i)} / \sigma_j$.

Steps (1-2) zero out the mean of the data, and may be omitted for data known to have zero mean (for instance, time series corresponding to speech or other acoustic signals). Steps (3-4) rescale each coordinate to have unit variance, which ensures that different attributes are all treated on the same "scale." For instance, if x_1 was cars' maximum speed in mph (taking values in the high tens or low hundreds) and x_2 were the number of seats (taking values around 2-4), then this renormalization rescales the different attributes to make them more comparable. Steps (3-4) may be omitted if we had apriori knowledge that the different attributes are all on the same scale. One

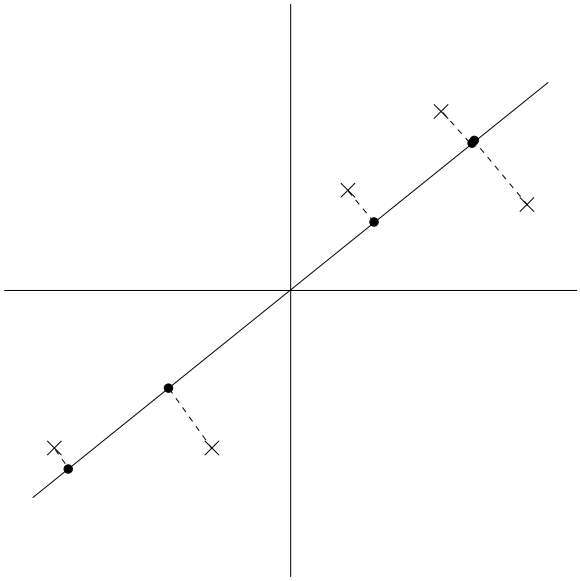
example of this is if each data point represented a grayscale image, and each $x_j^{(i)}$ took a value in $\{0, 1, \ldots, 255\}$ corresponding to the intensity value of pixel j in image i.

Now, having carried out the normalization, how do we compute the "major axis of variation" u—that is, the direction on which the data approximately lies? One way to pose this problem is as finding the unit vector u so that when the data is projected onto the direction corresponding to u, the variance of the projected data is maximized. Intuitively, the data starts off with some amount of variance/information in it. We would like to choose a direction u so that if we were to approximate the data as lying in the direction/subspace corresponding to u, as much as possible of this variance is still retained.

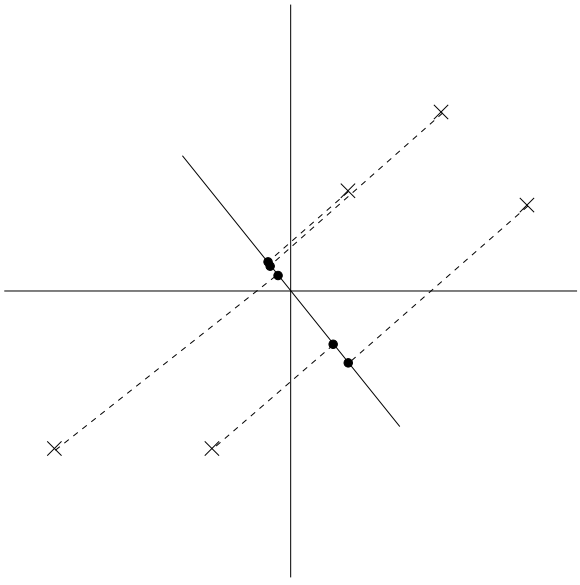
Consider the following dataset, on which we have already carried out the normalization steps:



Now, suppose we pick u to correspond the the direction shown in the figure below. The circles denote the projections of the original data onto this line.



We see that the projected data still has a fairly large variance, and the points tend to be far from zero. In contrast, suppose had instead picked the following direction:



Here, the projections have a significantly smaller variance, and are much closer to the origin.

We would like to automatically select the direction u corresponding to the first of the two figures shown above. To formalize this, note that given a

unit vector u and a point x, the length of the projection of x onto u is given by $x^T u$. I.e., if $x^{(i)}$ is a point in our dataset (one of the crosses in the plot), then its projection onto u (the corresponding circle in the figure) is distance $x^T u$ from the origin. Hence, to maximize the variance of the projections, we would like to choose a unit-length u so as to maximize:

$$\begin{aligned}\frac{1}{m}\sum_{i=1}^{m}(x^{(i)^T}u)^2 &= \frac{1}{m}\sum_{i=1}^{m} u^T x^{(i)} x^{(i)^T} u \\ &= u^T\left(\frac{1}{m}\sum_{i=1}^{m} x^{(i)}x^{(i)^T}\right)u.\end{aligned}$$

We easily recognize that the maximizing this subject to $||u||_2 = 1$ gives the principal eigenvector of $\Sigma = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}x^{(i)^T}$, which is just the empirical covariance matrix of the data (assuming it has zero mean).[1]

To summarize, we have found that if we wish to find a 1-dimensional subspace with with to approximate the data, we should choose u to be the principal eigenvector of Σ. More generally, if we wish to project our data into a k-dimensional subspace ($k < n$), we should choose $u_1, \ldots, u_k$ to be the top k eigenvectors of Σ. The u_i's now form a new, orthogonal basis for the data.[2]

Then, to represent $x^{(i)}$ in this basis, we need only compute the corresponding vector

$$y^{(i)} = \begin{bmatrix} u_1^T x^{(i)} \\ u_2^T x^{(i)} \\ \vdots \\ u_k^T x^{(i)} \end{bmatrix} \in \mathbb{R}^k.$$

Thus, whereas $x^{(i)} \in \mathbb{R}^n$, the vector $y^{(i)}$ now gives a lower, k-dimensional, approximation/representation for $x^{(i)}$. PCA is therefore also referred to as a **dimensionality reduction** algorithm. The vectors $u_1, \ldots, u_k$ are called the first k **principal components** of the data.

Remark. Although we have shown it formally only for the case of $k = 1$, using well-known properties of eigenvectors it is straightforward to show that

[1] If you haven't seen this before, try using the method of Lagrange multipliers to maximize $u^T\Sigma u$ subject to that $u^T u = 1$. You should be able to show that $\Sigma u = \lambda u$, for some λ, which implies u is an eigenvector of Σ, with eigenvalue λ.

[2] Because Σ is symmetric, the u_i's will (or always can be chosen to be) orthogonal to each other.

of all possible orthogonal bases $u_1, \ldots, u_k$, the one that we have chosen maximizes $\sum_i ||y^{(i)}||_2^2$. Thus, our choice of a basis preserves as much variability as possible in the original data.

In problem set 4, you will see that PCA can also be derived by picking the basis that minimizes the approximation error arising from projecting the data onto the k-dimensional subspace spanned by them.

PCA has many applications, our discussion with a small number of examples. First, compression—representing $x^{(i)}$'s with lower dimension $y^{(i)}$'s—is an obvious application. If we reduce high dimensional data to $k = 2$ or 3 dimensions, then we can also plot the $y^{(i)}$'s to visualize the data. For instance, if we were to reduce our automobiles data to 2 dimensions, then we can plot it (one point in our plot would correspond to one car type, say) to see what cars are similar to each other and what groups of cars may cluster together.

Another standard application is to preprocess a dataset to reduce its dimension before running a supervised learning learning algorithm with the $x^{(i)}$'s as inputs. Apart from computational benefits, reducing the data's dimension can also reduce the complexity of the hypothesis class considered and help avoid overfitting (e.g., linear classifiers over lower dimensional input spaces will have smaller VC dimension).

Lastly, as in our RC pilot example, we can also view PCA as a noise reduction algorithm. In our example it, estimates the intrinsic "piloting karma" from the noisy measures of piloting skill and enjoyment. In class, we also saw the application of this idea to face images, resulting in **eigenfaces** method. Here, each point $x^{(i)} \in \mathbb{R}^{100\times100}$ was a 10000 dimensional vector, with each coordinate corresponding to a pixel intensity value in a 100x100 image of a face. Using PCA, we represent each image $x^{(i)}$ with a much lower-dimensional $y^{(i)}$. In doing so, we hope that the principal components we found retain the interesting, systematic variations between faces that capture what a person really looks like, but not the "noise" in the images introduced by minor lighting variations, slightly different imaging conditions, and so on. We then measure distances between faces i and j by working in the reduced dimension, and computing $||y^{(i)} - y^{(j)}||_2$. This resulted in a surprisingly good face-matching and retrieval algorithm.

CS229 Lecture notes

Andrew Ng

Part XII
Independent Components Analysis

Our next topic is Independent Components Analysis (ICA). Similar to PCA, this will find a new basis in which to represent our data. However, the goal is very different.

As a motivating example, consider the "cocktail party problem." Here, n speakers are speaking simultaneously at a party, and any microphone placed in the room records only an overlapping combination of the n speakers' voices. But lets say we have n different microphones placed in the room, and because each microphone is a different distance from each of the speakers, it records a different combination of the speakers' voices. Using these microphone recordings, can we separate out the original n speakers' speech signals?

To formalize this problem, we imagine that there is some data $s \in \mathbb{R}^n$ that is generated via n independent sources. What we observe is

$$x = As,$$

where A is an unknown square matrix called the **mixing matrix**. Repeated observations gives us a dataset $\{x^{(i)}; i = 1, \ldots, m\}$, and our goal is to recover the sources $s^{(i)}$ that had generated our data ($x^{(i)} = As^{(i)}$).

In our cocktail party problem, $s^{(i)}$ is an n-dimensional vector, and $s_j^{(i)}$ is the sound that speaker j was uttering at time i. Also, $x^{(i)}$ in an n-dimensional vector, and $x_j^{(i)}$ is the acoustic reading recorded by microphone j at time i.

Let $W = A^{-1}$ be the **unmixing matrix.** Our goal is to find W, so that given our microphone recordings $x^{(i)}$, we can recover the sources by computing $s^{(i)} = Wx^{(i)}$. For notational convenience, we also let w_i^T denote

the i-th row of W, so that

$$W = \begin{bmatrix} — & w_1^T & — \\ & \vdots & \\ — & w_n^T & — \end{bmatrix}.$$

Thus, $w_i \in \mathbb{R}^n$, and the j-th source can be recovered by computing $s_j^{(i)} = w_j^T x^{(i)}$.

1 ICA ambiguities

To what degree can $W = A^{-1}$ be recovered? If we have no prior knowledge about the sources and the mixing matrix, it is not hard to see that there are some inherent ambiguities in A that are impossible to recover, given only the $x^{(i)}$'s.

Specifically, let P be any n-by-n permutation matrix. This means that each row and each column of P has exactly one "1." Here're some examples of permutation matrices:

$$P = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}; \quad P = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}; \quad P = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

If z is a vector, then Pz is another vector that's contains a permuted version of z's coordinates. Given only the $x^{(i)}$'s, there will be no way to distinguish between W and PW. Specifically, the permutation of the original sources is ambiguous, which should be no surprise. Fortunately, this does not matter for most applications.

Further, there is no way to recover the correct scaling of the w_i's. For instance, if A were replaced with $2A$, and every $s^{(i)}$ were replaced with $(0.5)s^{(i)}$, then our observed $x^{(i)} = 2A \cdot (0.5)s^{(i)}$ would still be the same. More broadly, if a single column of A were scaled by a factor of α, and the corresponding source were scaled by a factor of $1/\alpha$, then there is again no way, given only the $x^{(i)}$'s to determine that this had happened. Thus, we cannot recover the "correct" scaling of the sources. However, for the applications that we are concerned with—including the cocktail party problem—this ambiguity also does not matter. Specifically, scaling a speaker's speech signal $s_j^{(i)}$ by some positive factor α affects only the volume of that speaker's speech. Also, sign changes do not matter, and $s_j^{(i)}$ and $-s_j^{(i)}$ sound identical when played on a speaker. Thus, if the w_i found by an algorithm is scaled by any non-zero real

number, the corresponding recovered source $s_i = w_i^T x$ will be scaled by the same factor; but this usually does not matter. (These comments also apply to ICA for the brain/MEG data that we talked about in class.)

Are these the only sources of ambiguity in ICA? It turns out that they are, so long as the sources s_i are *non-Gaussian*. To see what the difficulty is with Gaussian data, consider an example in which $n = 2$, and $s \sim \mathcal{N}(0, I)$. Here, I is the 2x2 identity matrix. Note that the contours of the density of the standard normal distribution $\mathcal{N}(0, I)$ are circles centered on the origin, and the density is rotationally symmetric.

Now, suppose we observe some $x = As$, where A is our mixing matrix. The distribution of x will also be Gaussian, with zero mean and covariance $\mathrm{E}[xx^T] = \mathrm{E}[Ass^TA^T] = AA^T$. Now, let R be an arbitrary orthogonal (less formally, a rotation/reflection) matrix, so that $RR^T = R^TR = I$, and let $A' = AR$. Then if the data had been mixed according to A' instead of A, we would have instead observed $x' = A's$. The distribution of x' is also Gaussian, with zero mean and covariance $\mathrm{E}[x'(x')^T] = \mathrm{E}[A'ss^T(A')^T] = \mathrm{E}[ARss^T(AR)^T] = ARR^TA^T = AA^T$. Hence, whether the mixing matrix is A or A', we would observe data from a $\mathcal{N}(0, AA^T)$ distribution. Thus, there is no way to tell if the sources were mixed using A and A'. So, there is an arbitrary rotational component in the mixing matrix that cannot be determined from the data, and we cannot recover the original sources.

Our argument above was based on the fact that the multivariate standard normal distribution is rotationally symmetric. Despite the bleak picture that this paints for ICA on Gaussian data, it turns out that, so long as the data is *not* Gaussian, it is possible, given enough data, to recover the n independent sources.

2 Densities and linear transformations

Before moving on to derive the ICA algorithm proper, we first digress briefly to talk about the effect of linear transformations on densities.

Suppose we have a random variable s drawn according to some density $p_s(s)$. For simplicity, let us say for now that $s \in \mathbb{R}$ is a real number. Now, let the random variable x be defined according to $x = As$ (here, $x \in \mathbb{R}, A \in \mathbb{R}$). Let p_x be the density of x. What is p_x?

Let $W = A^{-1}$. To calculate the "probability" of a particular value of x, it is tempting to compute $s = Wx$, then then evaluate p_s at that point, and conclude that "$p_x(x) = p_s(Wx)$." However, *this is incorrect*. For example, let $s \sim \mathrm{Uniform}[0, 1]$, so that s's density is $p_s(s) = 1\{0 \leq s \leq 1\}$. Now, let

$A = 2$, so that $x = 2s$. Clearly, x is distributed uniformly in the interval $[0, 2]$. Thus, its density is given by $p_x(x) = (0.5)1\{0 \leq x \leq 2\}$. This does not equal $p_s(Wx)$, where $W = 0.5 = A^{-1}$. Instead, the correct formula is $p_x(x) = p_s(Wx)|W|$.

More generally, if s is a vector-valued distribution with density p_s, and $x = As$ for a square, invertible matrix A, then the density of x is given by

$$p_x(x) = p_s(Wx) \cdot |W|,$$

where $W = A^{-1}$.

Remark. If you're seen the result that A maps $[0, 1]^n$ to a set of volume $|A|$, then here's another way to remember the formula for p_x given above, that also generalizes our previous 1-dimensional example. Specifically, let $A \in \mathbb{R}^{n \times n}$ be given, and let $W = A^{-1}$ as usual. Also let $C_1 = [0, 1]^n$ be the n-dimensional hypercube, and define $C_2 = \{As : s \in C_1\} \subseteq \mathbb{R}^n$ to be the image of C_1 under the mapping given by A. Then it is a standard result in linear algebra (and, indeed, one of the ways of defining determinants) that the volume of C_2 is given by $|A|$. Now, suppose s is uniformly distributed in $[0, 1]^n$, so its density is $p_s(s) = 1\{s \in C_1\}$. Then clearly x will be uniformly distributed in C_2. Its density is therefore found to be $p_x(x) = 1\{x \in C_2\}/\text{vol}(C_2)$ (since it must integrate over C_2 to 1). But using the fact that the determinant of the inverse of a matrix is just the inverse of the determinant, we have $1/\text{vol}(C_2) = 1/|A| = |A^{-1}| = |W|$. Thus, $p_x(x) = 1\{x \in C_2\}|W| = 1\{Wx \in C_1\}|W| = p_s(Wx)|W|$.

3 ICA algorithm

We are now ready to derive an ICA algorithm. The algorithm we describe is due to Bell and Sejnowski, and the interpretation we give will be of their algorithm as a method for maximum likelihood estimation. (This is different from their original interpretation, which involved a complicated idea called the infomax principal, that is no longer necessary in the derivation given the modern understanding of ICA.)

We suppose that the distribution of each source s_i is given by a density p_s, and that the joint distribution of the sources s is given by

$$p(s) = \prod_{i=1}^{n} p_s(s_i).$$

Note that by modeling the joint distribution as a product of the marginal, we capture the assumption that the sources are independent. Using our

formulas from the previous section, this implies the following density on $x = As = W^{-1}s$:

$$p(x) = \prod_{i=1}^{n} p_s(w_i^T x) \cdot |W|.$$

All that remains is to specify a density for the individual sources p_s.

Recall that, given a real-valued random variable z, its cumulative distribution function (cdf) F is defined by $F(z_0) = P(z \leq z_0) = \int_{-\infty}^{z_0} p_z(z)dz$. Also, the density of z can be found from the cdf by taking its derivative: $p_z(z) = F'(z)$.

Thus, to specify a density for the s_i's, all we need to do is to specify some cdf for it. A cdf has to be a monotonic function that increases from zero to one. Following our previous discussion, we cannot choose the cdf to be the cdf of the Gaussian, as ICA doesn't work on Gaussian data. What we'll choose instead for the cdf, as a reasonable "default" function that slowly increases from 0 to 1, is the sigmoid function $g(s) = 1/(1 + e^{-s})$. Hence, $p_s(s) = g'(s)$.[1]

The square matrix W is the parameter in our model. Given a training set $\{x^{(i)}; i = 1, \ldots, m\}$, the log likelihood is given by

$$\ell(W) = \sum_{i=1}^{m} \left(\sum_{j=1}^{n} \log g'(w_j^T x^{(i)}) + \log |W| \right).$$

We would like to maximize this in terms W. By taking derivatives and using the fact (from the first set of notes) that $\nabla_W |W| = |W|(W^{-1})^T$, we easily derive a stochastic gradient ascent learning rule. For a training example $x^{(i)}$, the update rule is:

$$W := W + \alpha \left(\begin{bmatrix} 1 - 2g(w_1^T x^{(i)}) \\ 1 - 2g(w_2^T x^{(i)}) \\ \vdots \\ 1 - 2g(w_n^T x^{(i)}) \end{bmatrix} x^{(i)T} + (W^T)^{-1} \right),$$

[1] If you have prior knowledge that the sources' densities take a certain form, then it is a good idea to substitute that in here. But in the absence of such knowledge, the sigmoid function can be thought of as a reasonable default that seems to work well for many problems. Also, the presentation here assumes that either the data $x^{(i)}$ has been preprocessed to have zero mean, or that it can naturally be expected to have zero mean (such as acoustic signals). This is necessary because our assumption that $p_s(s) = g'(s)$ implies $\mathrm{E}[s] = 0$ (the derivative of the logistic function is a symmetric function, and hence gives a density corresponding to a random variable with zero mean), which implies $\mathrm{E}[x] = \mathrm{E}[As] = 0$.

where α is the learning rate.

After the algorithm converges, we then compute $s^{(i)} = Wx^{(i)}$ to recover the original sources.

Remark. When writing down the likelihood of the data, we implicity assumed that the $x^{(i)}$'s were independent of each other (for different values of i; note this issue is different from whether the different coordinates of $x^{(i)}$ are independent), so that the likelihood of the training set was given by $\prod_i p(x^{(i)}; W)$. This assumption is clearly incorrect for speech data and other time series where the $x^{(i)}$'s are dependent, but it can be shown that having correlated training examples will not hurt the performance of the algorithm if we have sufficient data. But, for problems where successive training examples are correlated, when implementing stochastic gradient ascent, it also sometimes helps accelerate convergence if we visit training examples in a randomly permuted order. (I.e., run stochastic gradient ascent on a randomly shuffled copy of the training set.)

CS229 Lecture notes

Andrew Ng

Part XIII
Reinforcement Learning and Control

We now begin our study of reinforcement learning and adaptive control.

In supervised learning, we saw algorithms that tried to make their outputs mimic the labels y given in the training set. In that setting, the labels gave an unambiguous "right answer" for each of the inputs x. In contrast, for many sequential decision making and control problems, it is very difficult to provide this type of explicit supervision to a learning algorithm. For example, if we have just built a four-legged robot and are trying to program it to walk, then initially we have no idea what the "correct" actions to take are to make it walk, and so do not know how to provide explicit supervision for a learning algorithm to try to mimic.

In the reinforcement learning framework, we will instead provide our algorithms only a reward function, which indicates to the learning agent when it is doing well, and when it is doing poorly. In the four-legged walking example, the reward function might give the robot positive rewards for moving forwards, and negative rewards for either moving backwards or falling over. It will then be the learning algorithm's job to figure out how to choose actions over time so as to obtain large rewards.

Reinforcement learning has been successful in applications as diverse as autonomous helicopter flight, robot legged locomotion, cell-phone network routing, marketing strategy selection, factory control, and efficient web-page indexing. Our study of reinforcement learning will begin with a definition of the **Markov decision processes (MDP)**, which provides the formalism in which RL problems are usually posed.

1 Markov decision processes

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

- S is a set of **states**. (For example, in autonomous helicopter flight, S might be the set of all possible positions and orientations of the helicopter.)
- A is a set of **actions**. (For example, the set of all possible directions in which you can push the helicopter's control sticks.)
- P_{sa} are the state transition probabilities. For each state $s \in S$ and action $a \in A$, P_{sa} is a distribution over the state space. We'll say more about this later, but briefly, P_{sa} gives the distribution over what states we will transition to if we take action a in state s.
- $\gamma \in [0, 1)$ is called the **discount factor**.
- $R : S \times A \mapsto \mathbb{R}$ is the **reward function**. (Rewards are sometimes also written as a function of a state S only, in which case we would have $R : S \mapsto \mathbb{R}$).

The dynamics of an MDP proceeds as follows: We start in some state s_0, and get to choose some action $a_0 \in A$ to take in the MDP. As a result of our choice, the state of the MDP randomly transitions to some successor state s_1, drawn according to $s_1 \sim P_{s_0 a_0}$. Then, we get to pick another action a_1. As a result of this action, the state transitions again, now to some $s_2 \sim P_{s_1 a_1}$. We then pick a_2, and so on.... Pictorially, we can represent this process as follows:

$$s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} s_3 \xrightarrow{a_3} \ldots$$

Upon visiting the sequence of states $s_0, s_1, \ldots$ with actions $a_0, a_1, \ldots$, our total payoff is given by

$$R(s_0, a_0) + \gamma R(s_1, a_1) + \gamma^2 R(s_2, a_2) + \cdots .$$

Or, when we are writing rewards as a function of the states only, this becomes

$$R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \cdots .$$

For most of our development, we will use the simpler state-rewards $R(s)$, though the generalization to state-action rewards $R(s, a)$ offers no special difficulties.

Our goal in reinforcement learning is to choose actions over time so as to maximize the expected value of the total payoff:

$$\mathrm{E}\left[R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \cdots\right]$$

Note that the reward at timestep t is **discounted** by a factor of γ^t. Thus, to make this expectation large, we would like to accrue positive rewards as soon as possible (and postpone negative rewards as long as possible). In economic applications where $R(\cdot)$ is the amount of money made, γ also has a natural interpretation in terms of the interest rate (where a dollar today is worth more than a dollar tomorrow).

A **policy** is any function $\pi : S \mapsto A$ mapping from the states to the actions. We say that we are **executing** some policy π if, whenever we are in state s, we take action $a = \pi(s)$. We also define the **value function** for a policy π according to

$$V^\pi(s) = \mathrm{E}\left[R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \cdots \,\middle|\, s_0 = s, \pi\right].$$

$V^\pi(s)$ is simply the expected sum of discounted rewards upon starting in state s, and taking actions according to π.[1]

Given a fixed policy π, its value function V^π satisfies the **Bellman equations**:

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

This says that the expected sum of discounted rewards $V^\pi(s)$ for starting in s consists of two terms: First, the **immediate reward** $R(s)$ that we get rightaway simply for starting in state s, and second, the expected sum of future discounted rewards. Examining the second term in more detail, we see that the summation term above can be rewritten $\mathrm{E}_{s' \sim P_{s\pi(s)}}[V^\pi(s')]$. This is the expected sum of discounted rewards for starting in state s', where s' is distributed according $P_{s\pi(s)}$, which is the distribution over where we will end up after taking the first action $\pi(s)$ in the MDP from state s. Thus, the second term above gives the expected sum of discounted rewards obtained *after* the first step in the MDP.

Bellman's equations can be used to efficiently solve for V^π. Specifically, in a finite-state MDP ($|S| < \infty$), we can write down one such equation for $V^\pi(s)$ for every state s. This gives us a set of $|S|$ linear equations in $|S|$ variables (the unknown $V^\pi(s)$'s, one for each state), which can be efficiently solved for the $V^\pi(s)$'s.

[1] This notation in which we condition on π isn't technically correct because π isn't a random variable, but this is quite standard in the literature.

We also define the **optimal value function** according to

$$V^*(s) = \max_\pi V^\pi(s). \tag{1}$$

In other words, this is the best possible expected sum of discounted rewards that can be attained using any policy. There is also a version of Bellman's equations for the optimal value function:

$$V^*(s) = R(s) + \max_{a \in A} \gamma \sum_{s' \in S} P_{sa}(s') V^*(s'). \tag{2}$$

The first term above is the immediate reward as before. The second term is the maximum over all actions a of the expected future sum of discounted rewards we'll get upon after action a. You should make sure you understand this equation and see why it makes sense.

We also define a policy $\pi^* : S \mapsto A$ as follows:

$$\pi^*(s) = \arg\max_{a \in A} \sum_{s' \in S} P_{sa}(s') V^*(s'). \tag{3}$$

Note that $\pi^*(s)$ gives the action a that attains the maximum in the "max" in Equation (2).

It is a fact that for every state s and every policy π, we have

$$V^*(s) = V^{\pi^*}(s) \geq V^\pi(s).$$

The first equality says that the V^{π^*}, the value function for π^*, is equal to the optimal value function V^* for every state s. Further, the inequality above says that π^*'s value is at least a large as the value of any other other policy. In other words, π^* as defined in Equation (3) is the optimal policy.

Note that π^* has the interesting property that it is the optimal policy for *all* states s. Specifically, it is not the case that if we were starting in some state s then there'd be some optimal policy for that state, and if we were starting in some other state s' then there'd be some other policy that's optimal policy for s'. Specifically, the same policy π^* attains the maximum in Equation (1) for *all* states s. This means that we can use the same policy π^* no matter what the initial state of our MDP is.

2 Value iteration and policy iteration

We now describe two efficient algorithms for solving finite-state MDPs. For now, we will consider only MDPs with finite state and action spaces ($|S| < \infty$, $|A| < \infty$).

The first algorithm, **value iteration**, is as follows:

1. For each state s, initialize $V(s) := 0$.

2. Repeat until convergence {

   For every state, update $V(s) := R(s) + \max_{a \in A} \gamma \sum_{s'} P_{sa}(s')V(s')$.

   }

This algorithm can be thought of as repeatedly trying to update the estimated value function using Bellman Equations (2).

There are two possible ways of performing the updates in the inner loop of the algorithm. In the first, we can first compute the new values for $V(s)$ for every state s, and then overwrite all the old values with the new values. This is called a **synchronous** update. In this case, the algorithm can be viewed as implementing a "Bellman backup operator" that takes a current estimate of the value function, and maps it to a new estimate. (See homework problem for details.) Alternatively, we can also perform **asynchronous** updates. Here, we would loop over the states (in some order), updating the values one at a time.

Under either synchronous or asynchronous updates, it can be shown that value iteration will cause V to converge to V^*. Having found V^*, we can then use Equation (3) to find the optimal policy.

Apart from value iteration, there is a second standard algorithm for finding an optimal policy for an MDP. The **policy iteration** algorithm proceeds as follows:

1. Initialize π randomly.

2. Repeat until convergence {

   (a) Let $V := V^{\pi}$.

   (b) For each state s, let $\pi(s) := \arg\max_{a \in A} \sum_{s'} P_{sa}(s')V(s')$.

   }

Thus, the inner-loop repeatedly computes the value function for the current policy, and then updates the policy using the current value function. (The policy π found in step (b) is also called the policy that is **greedy with respect to** V.) Note that step (a) can be done via solving Bellman's equations as described earlier, which in the case of a fixed policy, is just a set of $|S|$ linear equations in $|S|$ variables.

After at most a finite number of iterations of this algorithm, V will converge to V^*, and π will converge to π^*.

Both value iteration and policy iteration are standard algorithms for solving MDPs, and there isn't currently universal agreement over which algorithm is better. For small MDPs, policy iteration is often very fast and converges with very few iterations. However, for MDPs with large state spaces, solving for V^π explicitly would involve solving a large system of linear equations, and could be difficult. In these problems, value iteration may be preferred. For this reason, in practice value iteration seems to be used more often than policy iteration.

3 Learning a model for an MDP

So far, we have discussed MDPs and algorithms for MDPs assuming that the state transition probabilities and rewards are known. In many realistic problems, we are not given state transition probabilities and rewards explicitly, but must instead estimate them from data. (Usually, S, A and γ are known.)

For example, suppose that, for the inverted pendulum problem (see problem set 4), we had a number of trials in the MDP, that proceeded as follows:

$$\begin{aligned}
&s_0^{(1)} \xrightarrow{a_0^{(1)}} s_1^{(1)} \xrightarrow{a_1^{(1)}} s_2^{(1)} \xrightarrow{a_2^{(1)}} s_3^{(1)} \xrightarrow{a_3^{(1)}} \ldots \\
&s_0^{(2)} \xrightarrow{a_0^{(2)}} s_1^{(2)} \xrightarrow{a_1^{(2)}} s_2^{(2)} \xrightarrow{a_2^{(2)}} s_3^{(2)} \xrightarrow{a_3^{(2)}} \ldots \\
&\ldots
\end{aligned}$$

Here, $s_i^{(j)}$ is the state we were at time i of trial j, and $a_i^{(j)}$ is the corresponding action that was taken from that state. In practice, each of the trials above might be run until the MDP terminates (such as if the pole falls over in the inverted pendulum problem), or it might be run for some large but finite number of timesteps.

Given this "experience" in the MDP consisting of a number of trials, we can then easily derive the maximum likelihood estimates for the state transition probabilities:

$$P_{sa}(s') = \frac{\#\text{times took we action } a \text{ in state } s \text{ and got to } s'}{\#\text{times we took action a in state } s} \tag{4}$$

Or, if the ratio above is "0/0"—corresponding to the case of never having taken action a in state s before—the we might simply estimate $P_{sa}(s')$ to be $1/|S|$. (I.e., estimate P_{sa} to be the uniform distribution over all states.)

Note that, if we gain more experience (observe more trials) in the MDP, there is an efficient way to update our estimated state transition probabilities

using the new experience. Specifically, if we keep around the counts for both the numerator and denominator terms of (4), then as we observe more trials, we can simply keep accumulating those counts. Computing the ratio of these counts then given our estimate of P_{sa}.

Using a similar procedure, if R is unknown, we can also pick our estimate of the expected immediate reward $R(s)$ in state s to be the average reward observed in state s.

Having learned a model for the MDP, we can then use either value iteration or policy iteration to solve the MDP using the estimated transition probabilities and rewards. For example, putting together model learning and value iteration, here is one possible algorithm for learning in an MDP with unknown state transition probabilities:

1. Initialize π randomly.
2. Repeat {
   (a) Execute π in the MDP for some number of trials.
   (b) Using the accumulated experience in the MDP, update our estimates for P_{sa} (and R, if applicable).
   (c) Apply value iteration with the estimated state transition probabilities and rewards to get a new estimated value function V.
   (d) Update π to be the greedy policy with respect to V.
   }

We note that, for this particular algorithm, there is one simple optimization that can make it run much more quickly. Specifically, in the inner loop of the algorithm where we apply value iteration, if instead of initializing value iteration with $V = 0$, we initialize it with the solution found during the previous iteration of our algorithm, then that will provide value iteration with a much better initial starting point and make it converge more quickly.

Linear Algebra Review and Reference

Zico Kolter

October 16, 2007

1 Basic Concepts and Notation

Linear algebra provides a way of compactly representing and operating on sets of linear equations. For example, consider the following system of equations:

$$\begin{aligned} 4x_1 &- 5x_2 &&= -13 \\ -2x_1 &+ 3x_2 &&= 9 \ . \end{aligned}$$

This is two equations and two variables, so as you know from high school algebra, you can find a unique solution for x_1 and x_2 (unless the equations are somehow degenerate, for example if the second equation is simply a multiple of the first, but in the case above there is in fact a unique solution). In matrix notation, we can write the system more compactly as:

$$Ax = b$$
$$\text{with } A = \begin{bmatrix} 4 & -5 \\ -2 & 3 \end{bmatrix}, b = \begin{bmatrix} 13 \\ -9 \end{bmatrix}.$$

As we will see shortly, there are many advantages (including the obvious space savings) to analyzing linear equations in this form.

1.1 Basic Notation

We use the following notation:

- By $A \in \mathbb{R}^{m \times n}$ we denote a matrix with m rows and n columns, where the entries of A are real numbers.
- By $x \in \mathbb{R}^n$, we denote a vector with n entries. Usually a vector x will denote a ***column vector*** — i.e., a matrix with n rows and 1 column. If we want to explicitly represent a ***row vector*** — a matrix with 1 row and n columns — we typically write x^T (here x^T denotes the transpose of x, which we will define shortly).

- The ith element of a vector x is denoted x_i:

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}.$$

- We use the notation a_{ij} (or A_{ij}, $A_{i,j}$, etc) to denote the entry of A in the ith row and jth column:

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}.$$

- We denote the jth column of A by a_j or $A_{:,j}$:

$$A = \begin{bmatrix} | & | & & | \\ a_1 & a_2 & \cdots & a_n \\ | & | & & | \end{bmatrix}.$$

- We denote the ith row of A by a_i^T or $A_{i,:}$:

$$A = \begin{bmatrix} — & a_1^T & — \\ — & a_2^T & — \\ & \vdots & \\ — & a_m^T & — \end{bmatrix}.$$

- Note that these definitions are ambiguous (for example, the a_1 and a_1^T in the previous two definitions are *not* the same vector). Usually the meaning of the notation should be obvious from its use.

2 Matrix Multiplication

The product of two matrices $A \in \mathbb{R}^{m \times n}$ and $B \in \mathbb{R}^{n \times p}$ is the matrix

$$C = AB \in \mathbb{R}^{m \times p},$$

where

$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}.$$

Note that in order for the matrix product to exist, the number of columns in A must equal the number of rows in B. There are many ways of looking at matrix multiplication, and we'll start by examining a few special cases.

2.1 Vector-Vector Products

Given two vectors $x, y \in \mathbb{R}^n$, the quantity $x^T y$, sometimes called the ***inner product*** or ***dot product*** of the vectors, is a real number given by

$$x^T y \in \mathbb{R} = \sum_{i=1}^{n} x_i y_i.$$

Note that it is always the case that $x^T y = y^T x$.

Given vectors $x \in \mathbb{R}^m$, $y \in \mathbb{R}^n$ (they no longer have to be the same size), xy^T is called the ***outer product*** of the vectors. It is a matrix whose entries are given by $(xy^T)_{ij} = x_i y_j$, i.e.,

$$xy^T \in \mathbb{R}^{m \times n} = \begin{bmatrix} x_1 y_1 & x_1 y_2 & \cdots & x_1 y_n \\ x_2 y_1 & x_2 y_2 & \cdots & x_2 y_n \\ \vdots & \vdots & \ddots & \vdots \\ x_m y_1 & x_m y_2 & \cdots & x_m y_n \end{bmatrix}.$$

2.2 Matrix-Vector Products

Given a matrix $A \in \mathbb{R}^{m \times n}$ and a vector $x \in \mathbb{R}^n$, their product is a vector $y = Ax \in \mathbb{R}^m$. There are a couple ways of looking at matrix-vector multiplication, and we will look at them both.

If we write A by rows, then we can express Ax as,

$$y = \begin{bmatrix} — & a_1^T & — \\ — & a_2^T & — \\ & \vdots & \\ — & a_m^T & — \end{bmatrix} x = \begin{bmatrix} a_1^T x \\ a_2^T x \\ \vdots \\ a_m^T x \end{bmatrix}.$$

In other words, the ith entry of y is equal to the inner product of the ith *row* of A and x, $y_i = a_i^T x$.

Alternatively, lets write A in column form. In this case we see that,

$$y = \begin{bmatrix} | & | & & | \\ a_1 & a_2 & \cdots & a_n \\ | & | & & | \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} \\ a_1 \\ \\ \end{bmatrix} x_1 + \begin{bmatrix} \\ a_2 \\ \\ \end{bmatrix} x_2 + \ldots + \begin{bmatrix} \\ a_n \\ \\ \end{bmatrix} x_n \ .$$

In other words, y is a ***linear combination*** of the *columns* of A, where the coefficients of the linear combination are given by the entries of x.

So far we have been multiplying on the right by a column vector, but it is also possible to multiply on the left by a row vector. This is written, $y^T = x^T A$ for $A \in \mathbb{R}^{m \times n}$, $x \in \mathbb{R}^m$, and $y \in \mathbb{R}^n$. As before, we can express y^T in two obvious ways, depending on whether we

express A in terms on its rows or columns. In the first case we express A in terms of its columns, which gives

$$y^T = x^T \begin{bmatrix} | & | & & | \\ a_1 & a_2 & \cdots & a_n \\ | & | & & | \end{bmatrix} = \begin{bmatrix} x^T a_1 & x^T a_2 & \cdots & x^T a_n \end{bmatrix}$$

which demonstrates that the ith entry of y^T is equal to the inner product of x and the ith *column* of A.

Finally, expressing A in terms of rows we get the final representation of the vector-matrix product,

$$\begin{aligned} y^T &= \begin{bmatrix} x_1 & x_2 & \cdots & x_n \end{bmatrix} \begin{bmatrix} — & a_1^T & — \\ — & a_2^T & — \\ & \vdots & \\ — & a_m^T & — \end{bmatrix} \\ &= x_1 \begin{bmatrix} — & a_1^T & — \end{bmatrix} + x_2 \begin{bmatrix} — & a_2^T & — \end{bmatrix} + ... + x_n \begin{bmatrix} — & a_n^T & — \end{bmatrix} \end{aligned}$$

so we see that y^T is a linear combination of the *rows* of A, where the coefficients for the linear combination are given by the entries of x.

2.3 Matrix-Matrix Products

Armed with this knowledge, we can now look at four different (but, of course, equivalent) ways of viewing the matrix-matrix multiplication $C = AB$ as defined at the beginning of this section. First we can view matrix-matrix multiplication as a set of vector-vector products. The most obvious viewpoint, which follows immediately from the definition, is that the i, j entry of C is equal to the inner product of the ith row of A and the jth row of B. Symbolically, this looks like the following,

$$C = AB = \begin{bmatrix} — & a_1^T & — \\ — & a_2^T & — \\ & \vdots & \\ — & a_m^T & — \end{bmatrix} \begin{bmatrix} | & | & & | \\ b_1 & b_2 & \cdots & b_p \\ | & | & & | \end{bmatrix} = \begin{bmatrix} a_1^T b_1 & a_1^T b_2 & \cdots & a_1^T b_p \\ a_2^T b_1 & a_2^T b_2 & \cdots & a_2^T b_p \\ \vdots & \vdots & \ddots & \vdots \\ a_m^T b_1 & a_m^T b_2 & \cdots & a_m^T b_p \end{bmatrix}.$$

Remember that since $A \in \mathbb{R}^{m \times n}$ and $B \in \mathbb{R}^{n \times p}$, $a_i \in \mathbb{R}^n$ and $b_j \in \mathbb{R}^n$, so these inner products all make sense. This is the most "natural" representation when we represent A by rows and B by columns. Alternatively, we can represent A by columns, and B by rows, which leads to the interpretation of AB as a sum of outer products. Symbolically,

$$C = AB = \begin{bmatrix} | & | & & | \\ a_1 & a_2 & \cdots & a_n \\ | & | & & | \end{bmatrix} \begin{bmatrix} — & b_1^T & — \\ — & b_2^T & — \\ & \vdots & \\ — & b_n^T & — \end{bmatrix} = \sum_{i=1}^{n} a_i b_i^T \ .$$

Put another way, AB is equal to the sum, over all i, of the outer product of the ith column of A and the ith row of B. Since, in this case, $a_i \in \mathbb{R}^m$ and $b_i \in \mathbb{R}^p$, the dimension of the outer product $a_i b_i^T$ is $m \times p$, which coincides with the dimension of C.

Second, we can also view matrix-matrix multiplication as a set of matrix-vector products. Specifically, if we represent B by columns, we can view the columns of C as matrix-vector products between A and the columns of B. Symbolically,

$$C = AB = A \begin{bmatrix} | & | & & | \\ b_1 & b_2 & \cdots & b_p \\ | & | & & | \end{bmatrix} = \begin{bmatrix} | & | & & | \\ Ab_1 & Ab_2 & \cdots & Ab_p \\ | & | & & | \end{bmatrix}.$$

Here the ith column of C is given by the matrix-vector product with the vector on the right, $c_i = Ab_i$. These matrix-vector products can in turn be interpreted using both viewpoints given in the previous subsection. Finally, we have the analogous viewpoint, where we represent A by rows, and view the rows of C as the matrix-vector product between the rows of A and C. Symbolically,

$$C = AB = \begin{bmatrix} — & a_1^T & — \\ — & a_2^T & — \\ & \vdots & \\ — & a_m^T & — \end{bmatrix} B = \begin{bmatrix} — & a_1^T B & — \\ — & a_2^T B & — \\ & \vdots & \\ — & a_m^T B & — \end{bmatrix}.$$

Here the ith row of C is given by the matrix-vector product with the vector on the left, $c_i^T = a_i^T B$.

It may seem like overkill to dissect matrix multiplication to such a large degree, especially when all these viewpoints follow immediately from the initial definition we gave (in about a line of math) at the beginning of this section. However, virtually all of linear algebra deals with matrix multiplications of some kind, and it is worthwhile to spend some time trying to develop an intuitive understanding of the viewpoints presented here.

In addition to this, it is useful to know a few basic properties of matrix multiplication at a higher level:

- Matrix multiplication is associative: $(AB)C = A(BC)$.
- Matrix multiplication is distributive: $A(B + C) = AB + AC$.
- Matrix multiplication is, in general, *not* commutative; that is, it can be the case that $AB \neq BA$.

3 Operations and Properties

In this section we present several operations and properties of matrices and vectors. Hopefully a great deal of this will be review for you, so the notes can just serve as a reference for these topics.

3.1 The Identity Matrix and Diagonal Matrices

The ***identity matrix***, denoted $I \in \mathbb{R}^{n\times n}$, is a square matrix with ones on the diagonal and zeros everywhere else. That is,

$$I_{ij} = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases}$$

It has the property that for all $A \in \mathbb{R}^{m\times n}$,

$$AI = A = IA$$

where the size of I is determined by the dimensions of A so that matrix multiplication is possible.

A ***diagonal matrix*** is a matrix where all non-diagonal elements are 0. This is typically denoted $D = \text{diag}(d_1, d_2, \ldots, d_n)$, with

$$D_{ij} = \begin{cases} d_i & i = j \\ 0 & i \neq j \end{cases}$$

Clearly, $I = \text{diag}(1, 1, \ldots, 1)$.

3.2 The Transpose

The ***transpose*** of a matrix results from "flipping" the rows and columns. Given a matrix $A \in \mathbb{R}^{m\times n}$, is transpose, written A^T, is defined as

$$A^T \in \mathbb{R}^{n\times m},\ (A^T)_{ij} = A_{ji} \quad .$$

We have in fact already been using the transpose when describing row vectors, since the transpose of a column vector is naturally a row vector.

The following properties of transposes are easily verified:

- $(A^T)^T = A$
- $(AB)^T = B^T A^T$
- $(A + B)^T = A^T + B^T$

3.3 Symmetric Matrices

A square matrix $A \in \mathbb{R}^{n\times n}$ is ***symmetric*** if $A = A^T$. It is ***anti-symmetric*** if $A = -A^T$. It is easy to show that for any matrix $A \in \mathbb{R}^{n\times n}$, the matrix $A + A^T$ is symmetric and the matrix $A - A^T$ is anti-symmetric. From this it follows that any square matrix $A \in \mathbb{R}^{n\times n}$ can be represented as a sum of a symmetric matrix and an anti-symmetric matrix, since

$$A = \frac{1}{2}(A + A^T) + \frac{1}{2}(A - A^T)$$

and the first matrix on the right is symmetric, while the second is anti-symmetric. It turns out that symmetric matrices occur a great deal in practice, and they have many nice properties which we will look at shortly. It is common to denote the set of all symmetric matrices of size n as $\mathbb{S}^n$, so that $A \in \mathbb{S}^n$ means that A is a symmetric $n \times n$ matrix;

3.4 The Trace

The ***trace*** of a square matrix $A \in \mathbb{R}^{n \times n}$, denoted $\text{tr}(A)$ (or just $\text{tr}A$ if the parentheses are obviously implied), is the sum of diagonal elements in the matrix:

$$\text{tr}A = \sum_{i=1}^{n} A_{ii}.$$

As described in the CS229 lecture notes, the trace has the following properties (included here for the sake of completeness):

- For $A \in \mathbb{R}^{n \times n}$, $\text{tr}A = \text{tr}A^T$.
- For $A, B \in \mathbb{R}^{n \times n}$, $\text{tr}(A + B) = \text{tr}A + \text{tr}B$.
- For $A \in \mathbb{R}^{n \times n}$, $t \in \mathbb{R}$, $\text{tr}(tA) = t\ \text{tr}A$.
- For A, B such that AB is square, $\text{tr}AB = \text{tr}BA$.
- For A, B, C such that ABC is square, $\text{tr}ABC = \text{tr}BCA = \text{tr}CAB$, and so on for the product of more matrices.

3.5 Norms

A ***norm*** of a vector $\|x\|$ is informally measure of the "length" of the vector. For example, we have the commonly-used Euclidean or ℓ_2 norm,

$$\|x\|_2 = \sqrt{\sum_{i=1}^{n} x_i^2}.$$

Note that $\|x\|_2^2 = x^T x$.

More formally, a norm is any function $f : \mathbb{R}^n \to \mathbb{R}$ that satisfies 4 properties:

1. For all $x \in \mathbb{R}^n$, $f(x) \geq 0$ (non-negativity).
2. $f(x) = 0$ if and only if $x = 0$ (definiteness).
3. For all $x \in \mathbb{R}^n$, $t \in \mathbb{R}$, $f(tx) = |t|f(x)$ (homogeneity).
4. For all $x, y \in \mathbb{R}^n$, $f(x + y) \leq f(x) + f(y)$ (triangle inequality).

Other examples of norms are the ℓ_1 norm,

$$\|x\|_1 = \sum_{i=1}^{n} |x_i|$$

and the ℓ_∞ norm,

$$\|x\|_\infty = \max_i |x_i|.$$

In fact, all three norms presented so far are examples of the family of ℓ_p norms, which are parameterized by a real number $p \geq 1$, and defined as

$$\|x\|_p = \left(\sum_{i=1}^{n} |x_i|^p \right)^{1/p}.$$

Norms can also be defined for matrices, such as the Frobenius norm,

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

Many other norms exist, but they are beyond the scope of this review.

3.6 Linear Independence and Rank

A set of vectors $\{x_1, x_2, \ldots x_n\}$ is said to be ***(linearly) independent*** if no vector can be represented as a linear combination of the remaining vectors. Conversely, a vector which *can* be represented as a linear combination of the remaining vectors is said to be ***(linearly) dependent***. For example, if

$$x_n = \sum_{i=1}^{n-1} \alpha_i x_i$$

for some $\{\alpha_1, \ldots, \alpha_{n-1}\}$ then x_n is dependent on $\{x_1, \ldots, x_{n-1}\}$; otherwise, it is independent of $\{x_1, \ldots, x_{n-1}\}$.

The ***column rank*** of a matrix A is the largest number of columns of A that constitute linearly independent set. This is often referred to simply as the number of linearly independent columns, but this terminology is a little sloppy, since it is possible that any vector in some set $\{x_1, \ldots x_n\}$ can be expressed as a linear combination of the remaining vectors, even though some subset of the vectors might be independent. In the same way, the ***row rank*** is the largest number of rows of A that constitute a linearly independent set.

It is a basic fact of linear algebra, that for any matrix A, $\mathrm{columnrank}(A) = \mathrm{rowrank}(A)$, and so this quantity is simply refereed to as the ***rank*** of A, denoted as $\mathrm{rank}(A)$. The following are some basic properties of the rank:

- For $A \in \mathbb{R}^{m \times n}$, $\mathrm{rank}(A) \leq \min(m, n)$. If $\mathrm{rank}(A) = \min(m, n)$, then A is said to be ***full rank***.

- For $A \in \mathbb{R}^{m \times n}$, $\text{rank}(A) = \text{rank}(A^T)$.
- For $A \in \mathbb{R}^{m \times n}$, $B \in \mathbb{R}^{n \times p}$, $\text{rank}(AB) \leq \min(\text{rank}(A), \text{rank}(B))$.
- For $A, B \in \mathbb{R}^{m \times n}$, $\text{rank}(A + B) \leq \text{rank}(A) + \text{rank}(B)$.

3.7 The Inverse

The ***inverse*** of a square matrix $A \in \mathbb{R}^{n \times n}$ is denoted A^{-1}, and is the unique matrix such that

$$A^{-1}A = I = AA^{-1}.$$

It turns out that A^{-1} may not exist for some matrices A; we say A is ***invertible*** or ***non-singular*** if A^{-1} exists and ***non-invertible*** or ***singular*** otherwise. One condition for invertibility we already know: it is possible to show that A^{-1} exists if and only if A is full rank. We will soon see that there are many alternative sufficient and necessary conditions, in addition to full rank, for invertibility. The following are properties of the inverse; all assume that $A, B \in \mathbb{R}^{n \times n}$ are non-singular:

- $(A^{-1})^{-1} = A$
- If $Ax = b$, we can multiply by A^{-1} on both sides to obtain $x = A^{-1}b$. This demonstrates the inverse with respect to the original system of linear equalities we began this review with.
- $(AB)^{-1} = B^{-1}A^{-1}$
- $(A^{-1})^T = (A^T)^{-1}$. For this reason this matrix is often denoted A^{-T}.

3.8 Orthogonal Matrices

Two vectors $x, y \in \mathbb{R}^n$ are ***orthogonal*** if $x^T y = 0$. A vector $x \in \mathbb{R}^n$ is ***normalized*** if $\|x\|_2 = 1$. A square matrix $U \in \mathbb{R}^{n \times n}$ is ***orthogonal*** (note the different meanings when talking about vectors versus matrices) if all its columns are orthogonal to each other and are normalized (the columns are then referred to as being ***orthonormal***).

It follows immediately from the definition of orthogonality and normality that

$$U^T U = I = UU^T.$$

In other words, the inverse of an orthogonal matrix is its transpose. Note that if U is not square — i.e., $U \in \mathbb{R}^{m \times n}$, $n < m$ — but its columns are still orthonormal, then $U^T U = I$, but $UU^T \neq I$. We generally only use the term orthogonal to describe the previous case, where U is square.

Another nice property of orthogonal matrices is that operating on a vector with an orthogonal matrix will not change its Euclidean norm, i.e.,

$$\|Ux\|_2 = \|x\|_2$$

for any $x \in \mathbb{R}^n$, $U \in \mathbb{R}^{n \times n}$ orthogonal.

3.9 Range and Nullspace of a Matrix

The ***span*** of a set of vectors $\{x_1, x_2, \ldots x_n\}$ is the set of all vectors that can be expressed as a linear combination of $\{x_1, \ldots, x_n\}$. That is,

$$\text{span}(\{x_1, \ldots x_n\}) = \left\{ v : v = \sum_{i=1}^{n} \alpha_i x_i, \quad \alpha_i \in \mathbb{R} \right\}.$$

It can be shown that if $\{x_1, \ldots, x_n\}$ is a set of n linearly independent vectors, where each $x_i \in \mathbb{R}^n$, then $\text{span}(\{x_1, \ldots x_n\}) = \mathbb{R}^n$. In other words, *any* vector $v \in \mathbb{R}^n$ can be written as a linear combination of x_1 through x_n. The ***projection*** of a vector $y \in \mathbb{R}^m$ onto the span of $\{x_1, \ldots, x_n\}$ (here we assume $x_i \in \mathbb{R}^m$) is the vector $v \in \text{span}(\{x_1, \ldots x_n\})$, such that v as close as possible to y, as measured by the Euclidean norm $\|v - y\|_2$. We denote the projection as $\text{Proj}(y; \{x_1, \ldots, x_n\})$ and can define it formally as,

$$\text{Proj}(y; \{x_1, \ldots x_n\}) = \text{argmin}_{v \in \text{span}(\{x_1, \ldots, x_n\})} \|y - v\|_2.$$

The ***range*** (sometimes also called the columnspace) of a matrix $A \in \mathbb{R}^{m \times n}$, denoted $\mathcal{R}(A)$, is the the span of the columns of A. In other words,

$$\mathcal{R}(A) = \{v \in \mathbb{R}^m : v = Ax, x \in \mathbb{R}^n\}.$$

Making a few technical assumptions (namely that A is full rank and that $n < m$), the projection of a vector $y \in \mathbb{R}^m$ onto the range of A is given by,

$$\text{Proj}(y; A) = \text{argmin}_{v \in \mathcal{R}(A)} \|v - y\|_2 = A(A^T A)^{-1} A^T y \ .$$

This last equation should look extremely familiar, since it is almost the same formula we derived in class (and which we will soon derive again) for the least squares estimation of parameters. Looking at the definition for the projection, it should not be too hard to convince yourself that this is in fact the same objective that we minimized in our least squares problem (except for a squaring of the norm, which doesn't affect the optimal point) and so these problems are naturally very connected. When A contains only a single column, $a \in \mathbb{R}^m$, this gives the special case for a projection of a vector on to a line:

$$\text{Proj}(y; a) = \frac{aa^T}{a^T a} y \ .$$

The ***nullspace*** of a matrix $A \in \mathbb{R}^{m \times n}$, denoted $\mathcal{N}(A)$ is the set of all vectors that equal 0 when multiplied by A, i.e.,

$$\mathcal{N}(A) = \{x \in \mathbb{R}^n : Ax = 0\}.$$

Note that vectors in $\mathcal{R}(A)$ are of size m, while vectors in the $\mathcal{N}(A)$ are of size n, so vectors in $\mathcal{R}(A^T)$ and $\mathcal{N}(A)$ are both in $\mathbb{R}^n$. In fact, we can say much more. It turns out that

$$\left\{w : w = u + v, u \in \mathcal{R}(A^T), v \in \mathcal{N}(A)\right\} = \mathbb{R}^n \text{ and } \mathcal{R}(A^T) \cap \mathcal{N}(A) = \emptyset \ .$$

In other words, $\mathcal{R}(A^T)$ and $\mathcal{N}(A)$ are disjoint subsets that together span the entire space of $\mathbb{R}^n$. Sets of this type are called ***orthogonal complements***, and we denote this $\mathcal{R}(A^T) = \mathcal{N}(A)^\perp$.

3.10 The Determinant

The ***determinant*** of a square matrix $A \in \mathbb{R}^{n\times n}$, is a function $\det : \mathbb{R}^{n\times n} \to \mathbb{R}$, and is denoted $|A|$ or $\det A$ (like the trace operator, we usually omit parentheses). The full formula for the determinant gives little intuition about its meaning, so we instead first give three defining properties of the determinant, from which all the rest follow (including the general formula):

1. The determinant of the identity is 1, $|I| = 1$.
2. Given a matrix $A \in \mathbb{R}^{n\times n}$, if we multiply a single row in A by a scalar $t \in \mathbb{R}$, then the determinant of the new matrix is $t|A|$,
$$\begin{vmatrix} \begin{bmatrix} — & t\, a_1^T & — \\ — & a_2^T & — \\ & \vdots & \\ — & a_m^T & — \end{bmatrix} \end{vmatrix} = t|A| \ .$$
3. If we exchange any two rows a_i^T and a_j^T of A, then the determinant of the new matrix is $-|A|$, for example
$$\begin{vmatrix} \begin{bmatrix} — & a_2^T & — \\ — & a_1^T & — \\ & \vdots & \\ — & a_m^T & — \end{bmatrix} \end{vmatrix} = -|A| \ .$$

These properties, however, also give very little intuition about the nature of the determinant, so we now list several properties that follow from the three properties above:

- For $A \in \mathbb{R}^{n\times n}$, $|A| = |A^T|$.
- For $A, B \in \mathbb{R}^{n\times n}$, $|AB| = |A||B|$.
- For $A \in \mathbb{R}^{n\times n}$, $|A| = 0$ if and only if A is singular (i.e., non-invertible).
- For $A \in \mathbb{R}^{n\times n}$ and A non-singular, $|A|^{-1} = 1/|A|$.

Before given the general definition for the determinant, we define, for $A \in \mathbb{R}^{n\times n}$, $A_{\setminus i,\setminus j} \in \mathbb{R}^{(n-1)\times(n-1)}$ to be the *matrix* that results from deleting the ith row and jth column from A. The general (recursive) formula for the determinant is

$$\begin{aligned} |A| &= \sum_{i=1}^{n} (-1)^{i+j} a_{ij} |A_{\setminus i,\setminus j}| \quad \text{(for any } j \in 1,\dots,n) \\ &= \sum_{j=1}^{n} (-1)^{i+j} a_{ij} |A_{\setminus i,\setminus j}| \quad \text{(for any } i \in 1,\dots,n) \end{aligned}$$

with the initial case that $|A| = a_{11}$ for $A \in \mathbb{R}^{1\times 1}$. If we were to expand this formula completely for $A \in \mathbb{R}^{n\times n}$, there would be a total of $n!$ (n factorial) different terms. For this reason, we hardly even explicitly write the complete equation of the determinant for matrices bigger than 3×3. However, the equations for determinants of matrices up to size 3×3 are fairly common, and it is good to know them:

$$\begin{aligned}
|[a_{11}]| &= a_{11} \\
\left|\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}\right| &= a_{11}a_{22} - a_{12}a_{21} \\
\left|\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}\right| &= \begin{array}{l} a_{11}a_{22}a_{33} + a_{12}a_{23}a_{31} + a_{13}a_{21}a_{32} \\ \quad -a_{11}a_{23}a_{32} - a_{12}a_{21}a_{33} - a_{13}a_{22}a_{31} \end{array}
\end{aligned}$$

The ***classical adjoint*** (often just called the adjoint) of a matrix $A \in \mathbb{R}^{n\times n}$, is denoted $\text{adj}(A)$, and defined as

$$\text{adj}(A) \in \mathbb{R}^{n\times n}, \quad (\text{adj}(A))_{ij} = (-1)^{i+j}|A_{\setminus j,\setminus i}|$$

(note the switch in the indices $A_{\setminus j,\setminus i}$). It can be shown that for any nonsingular $A \in \mathbb{R}^{n\times n}$,

$$A^{-1} = \frac{1}{|A|}\text{adj}(A) \ .$$

While this is a nice "explicit" formula for the inverse of matrix, we should note that, numerically, there are in fact much more efficient ways of computing the inverse.

3.11 Quadratic Forms and Positive Semidefinite Matrices

Given a matrix square $A \in \mathbb{R}^{n\times n}$ and a vector $x \in \mathbb{R}$, the scalar value x^TAx is called a ***quadratic form***. Written explicitly, we see that

$$x^TAx = \sum_{i=1}^{n}\sum_{j=1}^{n} A_{ij}x_ix_j \ .$$

Note that,

$$x^TAx = (x^TAx)^T = x^TA^Tx = x^T(\frac{1}{2}A + \frac{1}{2}A^T)x$$

i.e., only the symmetric part of A contributes to the quadratic form. For this reason, we often implicitly assume that the matrices appearing in a quadratic form are symmetric.

We give the following definitions:

- A symmetric matrix $A \in \mathbb{S}^n$ is ***positive definite*** (PD) if for all non-zero vectors $x \in \mathbb{R}^n$, $x^TAx > 0$. This is usually denoted $A \succ 0$ (or just $A > 0$), and often times the set of all positive definite matrices is denoted $\mathbb{S}^n_{++}$.

- A symmetric matrix $A \in \mathbb{S}^n$ is ***position semidefinite*** (PSD) if for all vectors $x^T Ax \geq 0$. This is written $A \succeq 0$ (or just $A \geq 0$), and the set of all positive semidefinite matrices is often denoted $\mathbb{S}^n_+$.

- Likewise, a symmetric matrix $A \in \mathbb{S}^n$ is ***negative definite*** (ND), denoted $A \prec 0$ (or just $A < 0$) if for all non-zero $x \in \mathbb{R}^n$, $x^T Ax < 0$.

- Similarly, a symmetric matrix $A \in \mathbb{S}^n$ is ***negative semidefinite*** (NSD), denoted $A \succeq 0$ (or just $A \leq 0$) if for all $x \in \mathbb{R}^n$, $x^T Ax \leq 0$.

- Finally, a symmetric matrix $A \in \mathbb{S}^n$ is ***indefinite***, if it is neither positive semidefinite nor negative semidefinite — i.e., if there exists $x_1, x_2 \in \mathbb{R}^n$ such that $x_1^T Ax_1 > 0$ and $x_2^T Ax_2 < 0$.

It should be obvious that if A is positive definite, then $-A$ is negative definite and vice versa. Likewise, if A is positive semidefinite then $-A$ is negative semidefinite and vice versa. If A is indefinite, then so is $-A$. It can also be shown that positive definite and negative definite matrices are always invertible.

Finally, there is one type of positive definite matrix that comes up frequently, and so deserves some special mention. Given any matrix $A \in \mathbb{R}^{m \times n}$ (not necessarily symmetric or even square), the matrix $G = A^T A$ (sometimes called a ***Gram matrix***) is always positive semidefinite. Further, if $m \geq n$ (and we assume for convenience that A is full rank), then $G = A^T A$ is positive definite.

3.12 Eigenvalues and Eigenvectors

Given a square matrix $A \in \mathbb{R}^{n \times n}$, we say that $\lambda \in \mathbb{C}$ is an ***eigenvalue*** of A and $x \in \mathbb{C}^n$ is the corresponding ***eigenvector***[1] if

$$Ax = \lambda x, \quad x \neq 0 \ .$$

Intuitively, this definition means that multiplying A by the vector x results in a new vector that points in the same direction as x, but scaled by a factor λ. Also note that for any eigenvector $x \in \mathbb{C}^n$, and scalar $t \in \mathbb{C}$, $A(cx) = cAx = c\lambda x = \lambda(cx)$, so cx is also an eigenvector. For this reason when we talk about "the" eigenvector associated with λ, we usually assume that the eigenvector is normalized to have length 1 (this still creates some ambiguity, since x and $-x$ will both be eigenvectors, but we will have to live with this).

We can rewrite the equation above to state that (λ, x) is an eigenvalue-eigenvector pair of A if,

$$(\lambda I - A)x = 0, \quad x \neq 0 \ .$$

[1]Note that λ and the entries of x are actually in $\mathbb{C}$, the set of complex numbers, not just the reals; we will see shortly why this is necessary. Don't worry about this technicality for now, you can think of complex vectors in the same way as real vectors.

But $(\lambda I - A)x = 0$ has a non-zero solution to x if and only if $(\lambda I - A)$ has a non-empty nullspace, which is only the case if $(\lambda I - A)$ is singular, i.e.,

$$|(\lambda I - A)| = 0 \ .$$

We can now use the previous definition of the determinant to expand this expression into a (very large) polynomial in λ, where λ will have maximum degree n. We then find the n (possibly complex) roots of this polynomial to find the n eigenvalues $\lambda_1, \ldots, \lambda_n$. To find the eigenvector corresponding to the eigenvalue λ_i, we simply solve the linear equation $(\lambda_i I - A)x = 0$. It should be noted that this is not the method which is actually used in practice to numerically compute the eigenvalues and eigenvectors (remember that the complete expansion of the determinant has $n!$ terms); it is rather a mathematical argument.

The following are properties of eigenvalues and eigenvectors (in all cases assume $A \in \mathbb{R}^{n \times n}$ has eigenvalues $\lambda_i, \ldots, \lambda_n$ and associated eigenvectors $x_1, \ldots x_n$):

- The trace of a A is equal to the sum of its eigenvalues,

$$\mathrm{tr}A = \sum_{i=1}^{n} \lambda_i \ .$$

- The determinant of A is equal to the product of its eigenvalues,

$$|A| = \prod_{i=1}^{n} \lambda_i \ .$$

- The rank of A is equal to the number of non-zero eigenvalues of A.
- If A is non-singular then $1/\lambda_i$ is an eigenvalue of A^{-1} with associated eigenvector x_i, i.e., $A^{-1}x_i = (1/\lambda_i)x_i$.
- The eigenvalues of a diagonal matrix $D = \mathrm{diag}(d_1, \ldots d_n)$ are just the diagonal entries $d_1, \ldots d_n$.

We can write all the eigenvector equations simultaneously as

$$AX = X\Lambda$$

where the columns of $X \in \mathbb{R}^{n \times n}$ are the eigenvectors of A and Λ is a diagonal matrix whose entries are the eigenvalues of A, i.e.,

$$X \in \mathbb{R}^{n \times n} = \begin{bmatrix} | & | & & | \\ x_1 & x_2 & \cdots & x_n \\ | & | & & | \end{bmatrix}, \quad \Lambda = \mathrm{diag}(\lambda_1, \ldots, \lambda_n) \ .$$

If the eigenvectors of A are linearly independent, then the matrix X will be invertible, so $A = X\Lambda X^{-1}$. A matrix that can be written in this form is called ***diagonalizable***.

3.13 Eigenvalues and Eigenvectors of Symmetric Matrices

Two remarkable properties come about when we look at the eigenvalues and eigenvectors of a symmetric matrix $A \in \mathbb{S}^n$. First, it can be shown that all the eigenvalues of A are real. Secondly, the eigenvectors of A are orthonormal, i.e., the matrix X defined above is an orthogonal matrix (for this reason, we denote the matrix of eigenvectors as U in this case). We can therefore represent A as $A = U\Lambda U^T$, remembering from above that the inverse of an orthogonal matrix is just its transpose.

Using this, we can show that the definiteness of a matrix depends entirely on the sign of its eigenvalues. Suppose $A \in \mathbb{S}^n = U\Lambda U^T$. Then

$$x^T Ax = x^T U\Lambda U^T x = y^T \Lambda y = \sum_{i=1}^{n} \lambda_i y_i^2$$

where $y = U^T x$ (and since U is full rank, any vector $y \in \mathbb{R}^n$ can be represented in this form). Because y_i^2 is always positive, the sign of this expression depends entirely on the λ_i's. If all $\lambda_i > 0$, then the matrix is positive definite; if all $\lambda_i \geq 0$, it is positive semidefinite. Likewise, if all $\lambda_i < 0$ or $\lambda_i \leq 0$, then A is negative definite or negative semidefinite respectively. Finally, if A has both positive and negative eigenvalues, it is indefinite.

An application where eigenvalues and eigenvectors come up frequently is in maximizing some function of a matrix. In particular, for a matrix $A \in \mathbb{S}^n$, consider the following maximization problem,

$$\max_{x \in \mathbb{R}^n} \; x^T Ax \quad \text{subject to } \|x\|_2^2 = 1$$

i.e., we want to find the vector (of norm 1) which maximizes the quadratic form. Assuming the eigenvalues are ordered as $\lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_n$, the optimal x for this optimization problem is x_1, the eigenvector corresponding to λ_1. In this case the maximal value of the quadratic form is λ_1. Similarly, the optimal solution to the minimization problem,

$$\min_{x \in \mathbb{R}^n} \; x^T Ax \quad \text{subject to } \|x\|_2^2 = 1$$

is x_n, the eigenvector corresponding to λ_n, and the minimal value is λ_n. This can be proved by appealing to the eigenvector-eigenvalue form of A and the properties of orthogonal matrices. However, in the next section we will see a way of showing it directly using matrix calculus.

4 Matrix Calculus

While the topics in the previous sections are typically covered in a standard course on linear algebra, one topic that does not seem to be covered very often (and which we will use extensively) is the extension of calculus to the vector setting. Despite the fact that all the actual calculus we use is relatively trivial, the notation can often make things look much more difficult than they are. In this section we present some basic definitions of matrix calculus and provide a few examples.

4.1 The Gradient

Suppose that $f : \mathbb{R}^{m \times n} \to \mathbb{R}$ is a function that takes as input a matrix A of size $m \times n$ and returns a real value. Then the ***gradient*** of f (with respect to $A \in \mathbb{R}^{m \times n}$) is the matrix of partial derivatives, defined as:

$$\nabla_A f(A) \in \mathbb{R}^{m \times n} = \begin{bmatrix} \frac{\partial f(A)}{\partial A_{11}} & \frac{\partial f(A)}{\partial A_{12}} & \cdots & \frac{\partial f(A)}{\partial A_{1n}} \\ \frac{\partial f(A)}{\partial A_{21}} & \frac{\partial f(A)}{\partial A_{22}} & \cdots & \frac{\partial f(A)}{\partial A_{2n}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f(A)}{\partial A_{m1}} & \frac{\partial f(A)}{\partial A_{m2}} & \cdots & \frac{\partial f(A)}{\partial A_{mn}} \end{bmatrix}$$

i.e., an $m \times n$ matrix with

$$(\nabla_A f(A))_{ij} = \frac{\partial f(A)}{\partial A_{ij}}.$$

Note that the size of $\nabla_A f(A)$ is always the same as the size of A. So if, in particular, A is just a vector $x \in \mathbb{R}^n$,

$$\nabla_x f(x) = \begin{bmatrix} \frac{\partial f(x)}{\partial x_1} \\ \frac{\partial f(x)}{\partial x_2} \\ \vdots \\ \frac{\partial f(x)}{\partial x_n} \end{bmatrix}.$$

It is very important to remember that the gradient of a function is *only* defined if the function is real-valued, that is, if it returns a scalar value. We can not, for example, take the gradient of $Ax, A \in \mathbb{R}^{n \times n}$ with respect to x, since this quantity is vector-valued.

It follows directly from the equivalent properties of partial derivatives that:

- $\nabla_x(f(x) + g(x)) = \nabla_x f(x) + \nabla_x g(x)$.
- For $t \in \mathbb{R}$, $\nabla_x(t\ f(x)) = t\nabla_x f(x)$.

It is a little bit trickier to determine what the proper expression is for $\nabla_x f(Ax)$, $A \in \mathbb{R}^{n \times n}$, but this is doable as well (if fact, you'll have to work this out for a homework problem).

4.2 The Hessian

Suppose that $f : \mathbb{R}^n \to \mathbb{R}$ is a function that takes a vector in $\mathbb{R}^n$ and returns a real number. Then the ***Hessian*** matrix with respect to x, written $\nabla_x^2 f(x)$ or simply as H is the $n \times n$ matrix of partial derivatives,

$$\nabla_x^2 f(x) \in \mathbb{R}^{n \times n} = \begin{bmatrix} \frac{\partial^2 f(x)}{\partial x_1^2} & \frac{\partial^2 f(x)}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f(x)}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f(x)}{\partial x_2 \partial x_1} & \frac{\partial^2 f(x)}{\partial x_2^2} & \cdots & \frac{\partial^2 f(x)}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f(x)}{\partial x_n \partial x_1} & \frac{\partial^2 f(x)}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f(x)}{\partial x_n^2} \end{bmatrix}.$$

In other words, $\nabla_x^2 f(x) \in \mathbb{R}^{n \times n}$, with

$$(\nabla_x^2 f(x))_{ij} = \frac{\partial^2 f(x)}{\partial x_i \partial x_j}.$$

Note that the Hessian is always symmetric, since

$$\frac{\partial^2 f(x)}{\partial x_i \partial x_j} = \frac{\partial^2 f(x)}{\partial x_j \partial x_i}.$$

Similar to the gradient, the Hessian is defined only when $f(x)$ is real-valued.

It is natural to think of the gradient as the analogue of the first derivative for functions of vectors, and the Hessian as the analogue of the second derivative (and the symbols we use also suggest this relation). This intuition is generally correct, but there a few caveats to keep in mind.

First, for real-valued functions of one variable $f : \mathbb{R} \to \mathbb{R}$, it is a basic definition that the second derivative is the derivative of the first derivative, i.e.,

$$\frac{\partial^2 f(x)}{\partial x^2} = \frac{\partial}{\partial x}\frac{\partial}{\partial x} f(x).$$

However, for functions of a vector, the gradient of the function is a vector, and we cannot take the gradient of a vector — i.e.,

$$\nabla_x \nabla_x f(x) = \nabla_x \begin{bmatrix} \frac{\partial f(x)}{\partial x_1} \\ \frac{\partial f(x)}{\partial x_2} \\ \vdots \\ \frac{\partial f(x)}{\partial x_1} \end{bmatrix}$$

and this expression is not defined. Therefore, it is *not* the case that the Hessian is the gradient of the gradient. However, this is *almost* true, in the following sense: If we look at the ith entry of the gradient $(\nabla_x f(x))_i = \partial f(x)/\partial x_i$, and take the gradient with respect to x we get

$$\nabla_x \frac{\partial f(x)}{\partial x_i} = \begin{bmatrix} \frac{\partial^2 f(x)}{\partial x_i \partial x_1} \\ \frac{\partial^2 f(x)}{\partial x_i \partial x_2} \\ \vdots \\ \frac{\partial f(x)}{\partial x_i \partial x_n} \end{bmatrix}$$

which is the ith column (or row) of the Hessian. Therefore,

$$\nabla_x^2 f(x) = \begin{bmatrix} \nabla_x(\nabla_x f(x))_1 & \nabla_x(\nabla_x f(x))_2 & \cdots & \nabla_x(\nabla_x f(x))_n \end{bmatrix}.$$

If we don't mind being a little bit sloppy we can say that (essentially) $\nabla_x^2 f(x) = \nabla_x(\nabla_x f(x))^T$, so long as we understand that this really means taking the gradient of each entry of $(\nabla_x f(x))^T$, not the gradient of the whole vector.

Finally, note that while we can take the gradient with respect to a matrix $A \in \mathbb{R}^n$, for the purposes of this class we will only consider taking the Hessian with respect to a vector $x \in \mathbb{R}^n$. This is simply a matter of convenience (and the fact that none of the calculations we do require us to find the Hessian with respect to a matrix), since the Hessian with respect to a matrix would have to represent all the partial derivatives $\partial^2 f(A)/(\partial A_{ij} \partial A_{k\ell})$, and it is rather cumbersome to represent this as a matrix.

4.3 Gradients and Hessians of Quadratic and Linear Functions

Now let's try to determine the gradient and Hessian matrices for a few simple functions. It should be noted that all the gradients given here are special cases of the gradients given in the CS229 lecture notes.

For $x \in \mathbb{R}^n$, let $f(x) = b^T x$ for some known vector $b \in \mathbb{R}^n$. Then

$$f(x) = \sum_{i=1}^{n} b_i x_i$$

so

$$\frac{\partial f(x)}{\partial x_k} = \frac{\partial}{\partial x_k} \sum_{i=1}^{n} b_i x_i = b_k.$$

From this we can easily see that $\nabla_x b^T x = b$. This should be compared to the analogous situation in single variable calculus, where $\partial/(\partial x)\, ax = a$.

Now consider the quadratic function $f(x) = x^T A x$ for $A \in \mathbb{S}^n$. Remember that

$$f(x) = \sum_{i=1}^{n} \sum_{j=1}^{n} A_{ij} x_i x_j$$

so

$$\frac{\partial f(x)}{\partial x_k} = \frac{\partial}{\partial x_k} \sum_{i=1}^{n} \sum_{j=1}^{n} A_{ij} x_i x_j = \sum_{i=1}^{n} A_{ik} x_i + \sum_{j=1}^{n} A_{kj} x_j = 2 \sum_{i=1}^{n} A_{ki} x_i$$

where the last equality follows since A is symmetric (which we can safely assume, since it is appearing in a quadratic form). Note that the kth entry of $\nabla_x f(x)$ is just the inner product of the kth row of A and x. Therefore, $\nabla_x x^T A x = 2Ax$. Again, this should remind you of the analogous fact in single-variable calculus, that $\partial/(\partial x)\, ax^2 = 2ax$.

Finally, lets look at the Hessian of the quadratic function $f(x) = x^T A x$ (it should be obvious that the Hessian of a linear function $b^T x$ is zero). This is even easier than determining the gradient of the function, since

$$\frac{\partial^2 f(x)}{\partial x_k \partial x_\ell} = \frac{\partial^2}{\partial x_k \partial x_\ell} \sum_{i=1}^{n} \sum_{j=1}^{n} A_{ij} x_i x_j = A_{k\ell} + A_{\ell k} = 2A_{k\ell}.$$

Therefore, it should be clear that $\nabla_x^2 x^T A x = 2A$, which should be entirely expected (and again analogous to the single-variable fact that $\partial^2/(\partial x^2)\, ax^2 = 2a$).

To recap,

- $\nabla_x b^T x = b$
- $\nabla_x x^T A x = 2Ax$ (if A symmetric)
- $\nabla_x^2 x^T A x = 2A$ (if A symmetric)

4.4 Least Squares

Lets apply the equations we obtained in the last section to derive the least squares equations. Suppose we are given matrices $A \in \mathbb{R}^{m \times n}$ (for simplicity we assume A is full rank) and a vector $b \in \mathbb{R}^m$ such that $b \notin \mathcal{R}(A)$. In this situation we will not be able to find a vector $x \in \mathbb{R}^n$, such that $Ax = b$, so instead we want to find a vector x such that Ax is as close as possible to b, as measured by the square of the Euclidean norm $\|Ax - b\|_2^2$.

Using the fact that $\|x\|_2^2 = x^T x$, we have

$$\begin{aligned} \|Ax - b\|_2^2 &= (Ax - b)^T (Ax - b) \\ &= x^T A^T A x - 2b^T A x + b^T b \end{aligned}$$

Taking the gradient with respect to x we have, and using the properties we derived in the previous section

$$\begin{aligned} \nabla_x (x^T A^T A x - 2b^T A x + b^T b) &= \nabla_x x^T A^T A x - \nabla_x 2b^T A x + \nabla_x b^T b \\ &= 2A^T A x - 2A^T b \end{aligned}$$

Setting this last expression equal to zero and solving for x gives the normal equations

$$x = (A^T A)^{-1} A^T b$$

which is the same as what we derived in class.

4.5 Gradients of the Determinant

Now lets consider a situation where we find the gradient of a function with respect to a matrix, namely for $A \in \mathbb{R}^{n \times n}$, we want to find $\nabla_A |A|$. Recall from our discussion of determinants that

$$|A| = \sum_{i=1}^{n} (-1)^{i+j} A_{ij} |A_{\setminus i, \setminus j}| \quad \text{(for any } j \in 1, \ldots, n)$$

so

$$\frac{\partial}{\partial A_{k\ell}} |A| = \frac{\partial}{\partial A_{k\ell}} \sum_{i=1}^{n} (-1)^{i+j} A_{ij} |A_{\setminus i, \setminus j}| = (-1)^{k+\ell} |A_{\setminus k, \setminus \ell}| = (\text{adj}(A))_{\ell k}.$$

From this it immediately follows from the properties of the adjoint that

$$\nabla_A |A| = (\text{adj}(A))^T = |A| A^{-T}.$$

Now lets consider the function $f : \mathbb{S}^n_{++} \to \mathbb{R}$, $f(A) = \log |A|$. Note that we have to restrict the domain of f to be the positive definite matrices, since this ensures that $|A| > 0$, so that the log of $|A|$ is a real number. In this case we can use the chain rule (nothing fancy, just the ordinary chain rule from single-variable calculus) to see that

$$\frac{\partial \log |A|}{\partial A_{ij}} = \frac{\partial \log |A|}{\partial |A|} \frac{\partial |A|}{\partial A_{ij}} = \frac{1}{|A|} \frac{\partial |A|}{\partial A_{ij}}.$$

From this is should be obvious that

$$\nabla_A \log |A| = \frac{1}{|A|} \nabla_A |A| = A^{-1},$$

where we can drop the transpose in the last expression because A is symmetric. Note the similarity to the single-valued case, where $\partial/(\partial x)\ \log x = 1/x$.

4.6 Eigenvalues as Optimization

Finally, we use matrix calculus to solve an optimization problem in a way that leads directly to eigenvalue/eigenvector analysis. Consider the following, equality constrained optimization problem:

$$\max_{x \in \mathbb{R}^n}\ x^T A x \quad \text{subject to } \|x\|_2^2 = 1$$

for a symmetric matrix $A \in \mathbb{S}^n$. A standard way of solving optimization problems with equality constraints is by forming the ***Lagrangian***, an objective function that includes the equality constraints.[2] The Lagrangian in this case can be given by

$$\mathcal{L}(x, \lambda) = x^T A x - \lambda x^T x$$

where λ is called the Lagrange multiplier associated with the equality constraint. It can be established that for x^* to be a optimal point to the problem, the gradient of the Lagrangian has to be zero at x^* (this is not the only condition, but it is required). That is,

$$\nabla_x \mathcal{L}(x, \lambda) = \nabla_x (x^T A x - \lambda x^T x) = 2A^T x - 2\lambda x = 0.$$

Notice that this is just the linear equation $Ax = \lambda x$. This shows that the only points which can possibly maximize (or minimize) $x^T A x$ assuming $x^T x = 1$ are the eigenvectors of A.

[2] Don't worry if you haven't seen Lagrangians before, as we will cover them in greater detail later in CS229.

Probability Theory Review for Machine Learning

Samuel Ieong

November 6, 2006

1 Basic Concepts

Broadly speaking, probability theory is the mathematical study of uncertainty. It plays a central role in machine learning, as the design of learning algorithms often relies on probabilistic assumption of the data. This set of notes attempts to cover some basic probability theory that serves as a background for the class.

1.1 Probability Space

When we speak about probability, we often refer to the probability of an *event* of uncertain nature taking place. For example, we speak about the probability of rain next Tuesday. Therefore, in order to discuss probability theory formally, we must first clarify what the possible events are to which we would like to attach probability.

Formally, a *probability space* is defined by the triple $(\Omega, \mathcal{F}, P)$, where

- Ω is the *space of possible outcomes* (or *outcome space*),
- $\mathcal{F} \subseteq 2^\Omega$ (the power set of Ω) is the *space of (measurable) events* (or *event space*),
- P is the *probability measure* (or *probability distribution*) that maps an event $E \in \mathcal{F}$ to a real value between 0 and 1 (think of P as a function).

Given the outcome space Ω, there is some restrictions as to what subset of 2^Ω can be considered an event space $\mathcal{F}$:

- The trivial event Ω and the empty event $\emptyset$ is in $\mathcal{F}$.
- The event space $\mathcal{F}$ is closed under (countable) union, i.e., if $\alpha, \beta \in \mathcal{F}$, then $\alpha \cup \beta \in \mathcal{F}$.
- The even space $\mathcal{F}$ is closed under complement, i.e., if $\alpha \in \mathcal{F}$, then $(\Omega \setminus \alpha) \in \mathcal{F}$.

Example 1. *Suppose we throw a (six-sided) dice. The space of possible outcomes* $\Omega = \{1, 2, 3, 4, 5, 6\}$. *We may decide that the events of interest is whether the dice throw is odd or even. This event space will be given by* $\mathcal{F} = \{\emptyset, \{1, 3, 5\}, \{2, 4, 6\}, \Omega\}$.

Note that when the outcome space Ω is finite, as in the previous example, we often take the event space $\mathcal{F}$ to be 2^Ω. This treatment is not fully general, but it is often sufficient for practical purposes. However, when the outcome space is infinite, we must be careful to define what the event space is.

Given an event space $\mathcal{F}$, the probability measure P must satisfy certain axioms.

- (non-negativity) For all $\alpha \in \mathcal{F}$, $P(\alpha) \geq 0$.
- (trivial event) $P(\Omega) = 1$.
- (additivity) For all $\alpha, \beta \in \mathcal{F}$ and $\alpha \cap \beta = \emptyset$, $P(\alpha \cup \beta) = P(\alpha) + P(\beta)$.

Example 2. *Returning to our dice example, suppose we now take the event space $\mathcal{F}$ to be 2^Ω. Further, we define a probability distribution P over $\mathcal{F}$ such that*

$$P(\{1\}) = P(\{2\}) = \cdots = P(\{6\}) = 1/6$$

then this distribution P completely specifies the probability of any given event happening (through the additivity axiom). For example, the probability of an even dice throw will be

$$P(\{2, 4, 6\}) = P(\{2\}) + P(\{4\}) + P(\{6\}) = 1/6 + 1/6 + 1/6 = 1/2$$

since each of these events are disjoint.

1.2 Random Variables

Random variables play an important role in probability theory. The most important fact about random variables is that they are **not** variables. They are actually **functions** that map outcomes (in the outcome space) to real values. In terms of notation, we usually denote random variables by a capital letter. Let's see an example.

Example 3. *Again, consider the process of throwing a dice. Let X be a random variable that depends on the outcome of the throw. A natural choice for X would be to map the outcome i to the value i, i.e., mapping the event of throwing an "one" to the value of* 1*. Note that we could have chosen some strange mappings too. For example, we could have a random variable Y that maps all outcomes to* 0*, which would be a very boring function, or a random variable Z that maps the outcome i to the value of 2^i if i is odd and the value of $-i$ if i is even, which would be quite strange indeed.*

In a sense, random variables allow us to abstract away from the formal notion of event space, as we can define random variables that capture the appropriate events. For example, consider the event space of odd or even dice throw in Example 1. We could have defined a random variable that takes on value 1 if outcome i is odd and 0 otherwise. These type of binary random variables are very common in practice, and are known as *indicator variables*, taking its name from its use to indicate whether a certain event has happened. So why did we introduce event space? That is because when one studies probability theory (more

rigorously) using measure theory, the distinction between outcome space and event space will be very important. This topic is too advanced to be covered in this short review note. In any case, it is good to keep in mind that event space is not always simply the power set of the outcome space.

From here onwards, we will talk mostly about probability with respect to random variables. While some probability concepts can be defined meaningfully without using them, random variables allow us to provide a more uniform treatment of probability theory. For notations, the probability of a random variable X taking on the value of a will be denoted by either

$$P(X = a) \qquad \text{or} \qquad P_X(a)$$

We will also denote the range of a random variable X by $Val(X)$.

1.3 Distributions, Joint Distributions, and Marginal Distributions

We often speak about the *distribution* of a variable. This formally refers to the probability of a random variable taking on certain values. For example,

Example 4. *Let random variable X be defined on the outcome space Ω of a dice throw (again!). If the dice is fair, then the distribution of X would be*

$$P_X(1) = P_X(2) = \cdots = P_X(6) = 1/6$$

Note that while this example resembles that of Example 2, they have different semantic meaning. The probability distribution defined in Example 2 is over **events**, whereas the one here is defined over **random variables**.

For notation, we will use $P(X)$ to denote the distribution of the random variable X.

Sometimes, we speak about the distribution of more than one variables at a time. We call these distributions *joint distributions*, as the probability is determined jointly by all the variables involved. This is best clarified by an example.

Example 5. *Let X be a random variable defined on the outcome space of a dice throw. Let Y be an indicator variable that takes on value 1 if a coin flip turns up head and 0 if tail. Assuming both the dice and the coin are fair, the joint distribution of X and Y is given by*

P	$X=1$	$X=2$	$X=3$	$X=4$	$X=5$	$X=6$
$Y=0$	1/12	1/12	1/12	1/12	1/12	1/12
$Y=1$	1/12	1/12	1/12	1/12	1/12	1/12

As before, we will denote the probability of X taking value a and Y taking value b by either the long hand of $P(X = a, Y = b)$, or the short hand of $P_{X,Y}(a, b)$. We refer to their joint distribution by $P(X, Y)$.

Given a joint distribution, say over random variables X and Y, we can talk about the *marginal distribution* of X or that of Y. The marginal distribution refers to the probability distribution of a random variable on its own. To find out the marginal distribution of a

random variable, we *sum out* all the other random variables from the distribution. Formally, we mean

$$P(X) = \sum_{b \in Val(Y)} P(X, Y = b) \tag{1}$$

The name of marginal distribution comes from the fact that if we add up all the entries of a row (or a column) of a joint distribution, and write the answer at the end (i.e., margin) of it, this will be the probability of the random variable taking on that value. Of course, thinking in this way only helps when the joint distribution involves two variables.

1.4 Conditional Distributions

Conditional distributions are one of the key tools in probability theory for reasoning about uncertainty. They specify the distribution of a random variable when the value of another random variable is known (or more generally, when some event is known to be true).

Formally, conditional probability of $X = a$ *given* $Y = b$ is defined as

$$P(X = a|Y = b) = \frac{P(X = a, Y = b)}{P(Y = b)} \tag{2}$$

Note that this is not defined when the probability of $Y = b$ is 0.

Example 6. *Suppose we know that a dice throw was odd, and want to know the probability of an "one" has been thrown. Let X be the random variable of the dice throw, and Y be an indicator variable that takes on the value of 1 if the dice throw turns up odd, then we write our desired probability as follows:*

$$P(X = 1|Y = 1) = \frac{P(X = 1, Y = 1)}{P(Y = 1)} = \frac{1/6}{1/2} = 1/3$$

The idea of conditional probability extends naturally to the case when the distribution of a random variable is conditioned on several variables, namely

$$P(X = a|Y = b, Z = c) = \frac{P(X = a, Y = b, Z = c)}{P(Y = b, Z = c)}$$

As for notations, we write $P(X|Y = b)$ to denote the distribution of random variable X when $Y = b$. We may also write $P(X|Y)$ to denote a set of distributions of X, one for each of the different values that Y can take.

1.5 Independence

In probability theory, *independence* means that the distribution of a random variable does *not* change on learning the value of another random variable. In machine learning, we often make such assumptions about our data. For example, the training samples are assumed to

be drawn independently from some underlying space; the label of sample i is assumed to be independent of the features of sample j ($i \neq j$).

Mathematically, a random variable X is independent of Y when

$$P(X) = P(X|Y)$$

(Note that we have dropped what values X and Y are taking. This means the statement holds true for any values X and Y may take.)

Using Equation (2), it is easy to verify that if X is independent of Y, then Y is also independent of X. As a notation, we write $X \perp Y$ if X and Y are independent.

An equivalent mathematical statement about the independence of random variables X and Y is

$$P(X, Y) = P(X)P(Y)$$

Sometimes we also talk about *conditional independence*, meaning that if we know the value of a random variable (or more generally, a set of random variables), then some other random variables will be independent of each other. Formally, we say "X and Y are *conditionally* independent given Z" if

$$P(X|Z) = P(X|Y, Z)$$

or, equivalently,

$$P(X, Y|Z) = P(X|Z)P(Y|Z)$$

An example of conditional independence that we will se in class is the *Naïve Bayes* assumption. This assumption is made in the context of a learning algorithm for learning to classify emails as spams or non-spams. It assumes that the probability of a word x appearing in the email is conditionally independent of a word y appearing given whether the email is spam or not. This clearly is not without loss of generality, as some words almost invariably comes in pair. However, as it turns out, making this simplifying assumption does not hurt the performance much, and in any case allow us to learn to classify spams rapidly. Details can be found in Lecture Notes 2.

1.6 Chain Rule and Bayes Rule

We now present two basic yet important rules for manipulating that relates joint distributions and conditional distributions. The first is known as the *Chain Rule*. It can be seen as a generalization of Equation (2) to multiple random variables.

Theorem 1 (Chain Rule)**.**

$$P(X_1, X_2, \ldots, X_n) = P(X_1)P(X_2|X_1) \cdots P(X_n|X_1, X_2, \ldots, X_{n-1}) \tag{3}$$

The Chain Rule is often used to evaluate the joint probability of some random variables, and is especially useful when there are (conditional) independence across variables. Notice

there is a choice in the order we unravel the random variables when applying the Chain Rule; picking the right order can often make evaluating the probability much easier.

The second rule we are going to introduce is the *Bayes Rule*. The Bayes Rule allows us to compute the conditional probability $P(X|Y)$ from $P(Y|X)$, in a sense "inverting" the conditions. It can be derived simply from Equation (2) as well.

Theorem 2 (Bayes Rule)**.**

$$P(X|Y) = \frac{P(Y|X)P(X)}{P(Y)} \tag{4}$$

And recall that if $P(Y)$ is not given, we can always apply Equation (1) to find it.

$$P(Y) = \sum_{a \in Val(X)} P(X = a, Y) = \sum_{a \in Val(X)} P(Y|X = a)P(X = a)$$

This application of Equation (1) is sometimes referred to as the *law of total probability*.

Extending the Bayes Rule to the case of multiple random variables can sometimes be tricky. Just to be clear, we would give a few examples. When in doubt, one can always refer to how conditional probabilities are defined and work out the details.

Example 7. *Let's consider the following conditional probabilities:* $P(X, Y|Z)$ *and* $(X|Y, Z)$*.*

$$P(X, Y|Z) = \frac{P(Z|X, Y)P(X, Y)}{P(Z)} = \frac{P(Y, Z|X)P(X)}{P(Z)}$$
$$P(X|Y, Z) = \frac{P(Y|X, Z)P(X, Z)}{P(Y, Z)} = \frac{P(Y|X, Z)P(X|Z)P(Z)}{P(Y|Z)P(Z)} = \frac{P(Y|X, Z)P(X|Z)}{P(Y|Z)}$$

2 Defining a Probability Distribution

We have been talking about probability distributions for a while. But how do we define a distribution? In a broad sense, there are two classes of distribution that require seemingly different treatments (these can be unified using measure theory). Namely, *discrete* distributions and *continuous* distributions. We will discuss how distributions are specified next.

Note that this discussion is distinct from how we can efficiently *represent* a distribution. The topic of efficient representation of probability distribution is in fact a very important and active research area that deserves its own course. If you are interested to learn more about how to efficiently represent, reason, and perform learning on distributions, you are advised to take CS228: Probabilistic Models in Artificial Intelligence.

2.1 Discrete Distribution: Probability Mass Function

By a discrete distribution, we mean that the random variable of the underlying distribution can take on only *finitely many* different values (or that the outcome space is finite).

To define a discrete distribution, we can simply enumerate the probability of the random variable taking on each of the possible values. This enumeration is known as the *probability mass function*, as it divides up a unit mass (the total probability) and places them on the different values a random variable can take. This can be extended analogously to joint distributions and conditional distributions.

2.2 Continuous Distribution: Probability Density Function

By a continuous distribution, we mean that the random variable of the underlying distribution can take on *infinitely many* different values (or that the outcome space is infinite).

This is arguably a trickier situation than the discrete case, since if we place a non-zero amount of mass on each of the values, the total mass will add up to infinity, which violates the requirement that the total probaiblity must sum up to one.

To define a continuous distribution, we will make use of *probability density function* (PDF). A probability density function, f, is a *non-negative*, *integrable* function such that

$$\int_{Val(X)} f(x)dx = 1$$

The probability of a random variable X distributed according to a PDF f is computed as follows

$$P(a \leq X \leq b) = \int_a^b f(x)dx$$

Note that this, in particular, implies that the probability of a continuously distributed random variable taking on any given single value is zero.

Example 8 (Uniform distribution)**.** *Let's consider a random variable X that is uniformly distributed in the range $[0, 1]$. The corresponding PDF would be*

$$f(x) = \begin{cases} 1 & \text{if } 0 \leq x \leq 1 \\ 0 & \text{otherwise} \end{cases}$$

We can verify that $\int_0^1 1\ dx$ is indeed 1, and therefore f is a PDF. To compute the probability of X smaller than a half,

$$P(X \leq 1/2) = \int_0^{1/2} 1\ dx = [x]_0^{1/2} = 1/2$$

More generally, suppose X is distributed uniformly over the range $[a, b]$, then the PDF would be

$$f(x) = \begin{cases} \frac{1}{b-a} & \text{if } a \leq x \leq b \\ 0 & \text{otherwise} \end{cases}$$

Sometimes we will also speak about *cumulative distribution function*. It is a function that gives the probability of a random variable being smaller than some value. A cumulative distribution function F is related to the underlying probability density function f as follows:

$$F(b) = P(X \leq b) = \int_{-\infty}^{b} f(x)dx$$

and hence $F(x) = \int f(x)dx$ (in the sense of indefinite integral).

To extend the definition of continuous distribution to joint distribution, the probability density function is extended to take multiple arguments, namely,

$$P(a_1 \leq X_1 \leq b_1, a_2 \leq X_2 \leq b_2, \ldots, a_n \leq X_n \leq n_1) = \int_{a_1}^{b_1} \int_{a_2}^{b_2} \cdots \int_{a_n}^{b_n} f(x_1, x_2, \ldots, x_n)dx_1dx_2 \ldots dx_n$$

To extend the definition of conditional distribution to continuous random variables, we ran into the problem that the probability of a continuous random variable taking on a single value is 0, so Equation (2) is not well defined, since the denominator equals 0. To define the conditional distribution of a continuous variable, let $f(x, y)$ be the joint distribution of X and Y. Through application of analysis, we can show that the PDF, $f(y|x)$, underlying the distribution $P(Y|X)$ is given by

$$f(y|x) = \frac{f(x, y)}{f(x)}$$

For example,

$$P(a \leq Y \leq b|X = c) = \int_{a}^{b} f(y|c)dy = \int_{a}^{b} \frac{f(c, y)}{f(c)}dy$$

3 Expectations and Variance

3.1 Expectations

One of the most common operations we perform on a random variable is to compute its *expectation*, also known as its *mean*, *expected value*, or *first moment*. The expectation of a random variable, denoted by $E(X)$, is given by

$$E(X) = \sum_{a \in Val(X)} aP(X = a) \qquad \text{or} \qquad E(X) = \int_{a \in Val(X)} xf(x)\, dx \tag{5}$$

Example 9. *Let X be the outcome of rolling a fair dice. The expectation of X is*

$$E(X) = (1)\frac{1}{6} + (2)\frac{1}{6} + \cdots + 6\frac{1}{6} = 3\frac{1}{2}$$

We may sometimes be interested in computing the expected value of some function f of a random variable X. Recall, however, that a random variable is also a function itself, so

the easiest way to think about this is that we define a new random variable $Y = f(X)$, and compute the expected value of Y instead.

When working with indicator variables, a useful identify is the following:

$$E(X) = P(X = 1) \qquad \text{for indicator variable } X$$

When working with the sums of random variables, one of the most important rule is the *linearity of expectations*.

Theorem 3 (Linearity of Expectations)**.** *Let X_1, X_2, ..., X_n be (possibly dependent) random variables,*

$$E(X_1 + X_2 + \cdots + X_n) = E(X_1) + E(X_2) + \cdots + E(X_n) \tag{6}$$

The linearity of expectations is very powerful because there are no restrictions on whether the random variables are independent or not. When we work on products of random variables, however, there is very little we can say in general. However, when the random variables are independent, then

Theorem 4. *Let X and Y be independent random variables,*

$$E(XY) = E(X)E(Y)$$

3.2 Variance

The *variance* of a distribution is a measure of the "spread" of a distribution. Sometimes it is also referred to as the *second moment*. It is defined as follows:

$$Var(X) = E\left((X - E(X))^2\right) \tag{7}$$

The variance of a random variable is often denoted by σ^2. The reason that this is squared is because we often want to find out σ, known as the *standard deviation*. The variance and the standard deviation is related (obviously) by $\sigma = \sqrt{Var(X)}$.

To find out the variance of a random variable X, it's often easier to compute the following instead

$$Var(X) = E(X^2) - (E(X))^2$$

Note that unlike expectation, variance is not a linear function of a random variable X. In fact, we can verify that the variance of $(aX + b)$ is

$$Var(aX + b) = a^2 Var(X)$$

If random variables X and Y are independent, then

$$Var(X + Y) = Var(X)Var(Y) \qquad \text{if } X \perp Y$$

Sometimes we also talk about the *covariance* of two random variables. This is a measure of how "closely related" two random variables are. Its definition is as follows.

$$Cov(X, Y) = E((X - E(X))(Y - E(Y)))$$

4 Some Important Distributions

In this section, we will review some of the probability distributions that we will see in this class. This is by no means a comprehensive list of distribution that one should know. In particular, distributions such as the geoemtric, hypergeometric, and binomial distributions, which are very useful in their own right and studied in introductory probability theory, are not reviewed here.

4.1 Bernoulli

The *Bernoulli distribution* is one of the most basic distribution. A random variable distributed according to the Bernoulli distribution can take on two possible values, $\{0, 1\}$. It can be specified by a single parameter p, and by convention we take p to be $P(X = 1)$. It is often used to indicate whether a trail is successful or not.

Sometimes it is useful to write the probability distribution of a Bernoulli random variable X as follows

$$P(X) = p^x(1-p)^{1-x}$$

An example of the Bernoulli distribution in action is the classification task in Lecture Notes 1. To develop the logistic regression algorithm for the task, we assume that the labels are distributed according to the Bernoulli distribution given the features.

4.2 Poisson

The *Poisson distribution* is a very useful distribution that deals with the arrival of events. It measures probaiblity of the number of events happening over a fixed period of time, given a fixed average rate of occurrence, and that the events take place independently of the time since the last event. It is parametrized by the average arrival rate λ. The probability mass function is given by:

$$P(X = k) = \frac{\exp(-\lambda)\lambda^k}{k!}$$

The mean value of a Poisson random variable is λ, and its variance is also λ.

We will get to work on a learning algorithm that deals with Poisson random variables in Homework 1, Problem 3.

4.3 Gaussian

The *Gaussian distribution*, also known as the *normal distribution*, is one of the most "versatile" distributions in probability theory, and appears in a wide variety of contexts. For example, it can be used to approximate the binomial distribution when the number of experiments is large, or the Poisson distribution when the average arrival rate is high. It is also related to the Law of Large Numbers. For many problems, we will also often assume that when noise in the system is Gaussian distributed. The list of applications is endless.

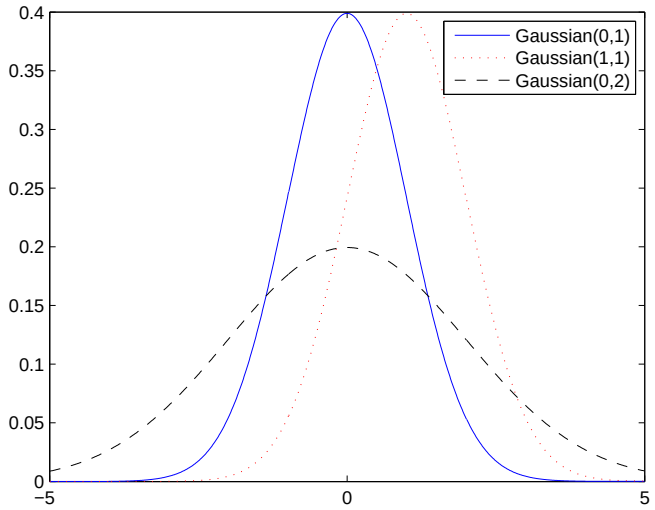


Figure 1: Gaussian distributions under different mean and variance

The Gaussian distribution is determined by two parameters: the mean μ and the variance σ^2. The probability density function is given by

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right) \tag{8}$$

To get a better sense of how the distribution changes with respect to the mean and the variance, we have plotted three different Gaussian distributions in Figure 1.

In our class, we will sometimes work with multi-variate Gaussian distributions. A k-dimensional multi-variate Gaussian distribution is parametrized by (μ, Σ), where μ is now a *vector* of means in $\mathbb{R}^k$, and Σ is the *covariance matrix* in $\mathbb{R}^{k\times k}$, in other words, $\Sigma_{ii} = Var(X_i)$ and $\Sigma_{ij} = Cov(X_i, X_j)$. The probability density function is now defined over vectors of input, given by

$$f(\mathbf{x}) = \frac{1}{\sqrt{2\pi^k|\Sigma|}} \exp\left(-\frac{1}{2}(\mathbf{x}-\mu)^T\Sigma^{-1}(\mathbf{x}-\mu)\right) \tag{9}$$

(Recall that we denote the determinant of a matrix A by $|A|$, and its inverse by A^{-1})

To get a better sense of how a multi-variate Gaussian distribution depends on the covariance matrix, we can look at the figures in Lecture Notes 2, Pages 3—4.

Working with a multi-variate Gaussian distribution can be tricky and daunting at times. One way to make our lives easier, at least as a way to get intuition on a problem, is to assume that the covariances are zero when we first attempt a problem. When the covariances are zero, the determinant $|\Sigma|$ will simply be the product of the variances, and the inverse Σ^{-1} can be found by taking the inverse of the diagonal entries of Σ.

5 Working with Probabilities

As we will be working with probabilities and distributions a lot in this class, listed below are a few tips about efficient manipulation of distributions.

5.1 The log trick

In machine learning, we generally assume the independence of different samples. Therefore, we often have to deal with the product of a (large) number of distributions. When our goal is to optimize functions of such products, it is often easier if we first work with the logarithm of such functions. As the logarithmic function is a strictly increasing function, it will not distort where the maximum is located (although, most certainly, the maximum value of the function before and after taking logarithm will be different).

As an example, consider the likelihood function in Lecture Notes 1, Page 17.

$$L(\theta) = \prod_{i=1}^{m} (h_\theta(x^{(i)}))^{y^{(i)}} (1 - h_\theta(x^{(i)}))^{1-y^{(i)}}$$

I dare say this is a pretty mean-looking function. But by taking the logarithm of it, termed log-likelihood function, we have instead

$$\ell(\theta) = \log L(\theta) = \sum_{i=1}^{m} y^{(i)} \log h_\theta(x^{(i)}) + (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)}))$$

Not the world's prettiest function, but at least it's more manageable. We can now work on one term (i.e., one training sample) at a time, because they are summed together rather than multiplied together.

5.2 Delayed Normalization

Because probability has to sum up to one, we often have to deal with normalization, especially with continuous distribution. For example, for Gaussian distributions, the term outside of the exponent is to ensure that the integral of the PDF evaluates to one. When we are sure that the end product of some algebra will be a probability distribution, or when we are finding the optimum of some distributions, it's often easier to simply denote the normalization constant to be Z, and not worry about computing the normalization constant all the time.

5.3 Jenson's Inequality

Sometimes when we are evaluating the expectation of a function of a random variable, we may only need a bound rather than its exact value. In these situations, if the function is convex or concave, Jenson's inequality allows us to derive a bound by evaluating the value of the function at the expectation of the random variable itself.

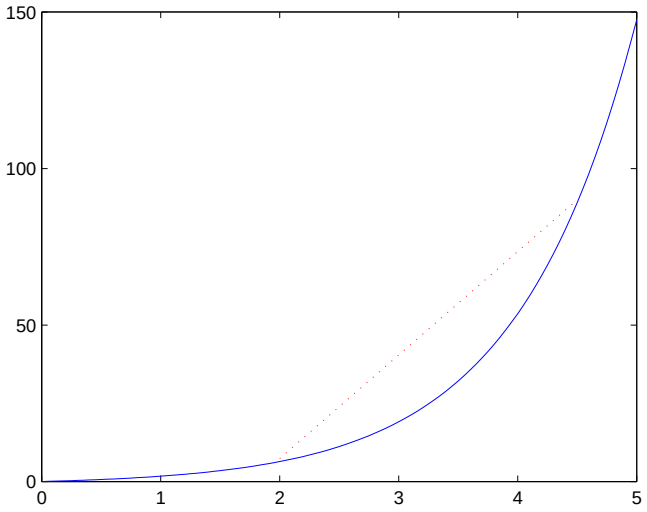


Figure 2: Illustration of Jenson's Inequality

Theorem 5 (Jenson's Inequality)**.** *Let X be a random variable, and f be a convex function. Then*

$$f(E(X)) \leq E(f(X))$$

If f is a concave function, then

$$f(E(X)) \geq E(f(X))$$

While we can show Jenson's inequality by algebra, it's easiest to understand it through a picture. The function in Figure 2 is a convex function. We can see that a straight line between any two points on the function always lie above the function. This shows that if a random variable can take on only two values, then Jenson's inequality holds. It is relatively straight forward to extend this to general random variables.

Convex Optimization Overview

Zico Kolter

October 19, 2007

1 Introduction

Many situations arise in machine learning where we would like to ***optimize*** the value of some function. That is, given a function $f : \mathbb{R}^n \to \mathbb{R}$, we want to find $x \in \mathbb{R}^n$ that minimizes (or maximizes) $f(x)$. We have already seen several examples of optimization problems in class: least-squares, logistic regression, and support vector machines can all be framed as optimization problems.

It turns out that in the general case, finding the global optimum of a function can be a very difficult task. However, for a special class of optimization problems, known as ***convex optimization problems***, we can efficiently find the global solution in many cases. Here, "efficiently" has both practical and theoretical connotations: it means that we can solve many real-world problems in a reasonable amount of time, and it means that theoretically we can solve problems in time that depends only *polynomially* on the problem size.

The goal of these section notes and the accompanying lecture is to give a very brief overview of the field of convex optimization. Much of the material here (including some of the figures) is heavily based on the book *Convex Optimization* [1] by Stephen Boyd and Lieven Vandenberghe (available for free online), and EE364, a class taught here at Stanford by Stephen Boyd. If you are interested in pursuing convex optimization further, these are both excellent resources.

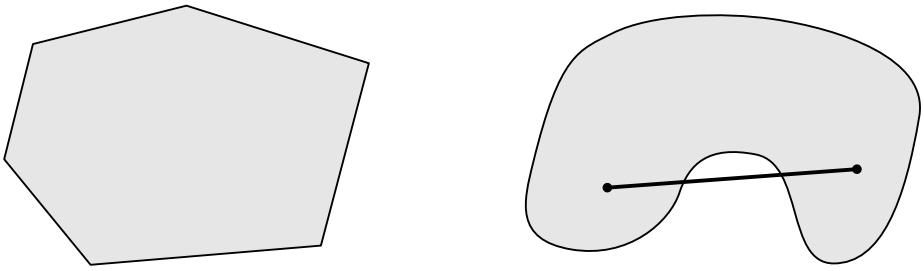
2 Convex Sets

We begin our look at convex optimization with the notion of a ***convex set***.

Definition 2.1 *A set C is convex if, for any $x, y \in C$ and $\theta \in \mathbb{R}$ with $0 \leq \theta \leq 1$,*

$$\theta x + (1 - \theta)y \in C.$$

Intuitively, this means that if we take any two elements in C, and draw a line segment between these two elements, then every point on that line segment also belongs to C. Figure 1 shows an example of one convex and one non-convex set. The point $\theta x + (1 - \theta)y$ is called a ***convex combination*** of the points x and y.



(a) (b)

Figure 1: Examples of a convex set (a) and a non-convex set (b).

2.1 Examples

- **All of $\mathbb{R}^n$.** It should be fairly obvious that given any $x, y \in \mathbb{R}^n$, $\theta x + (1-\theta)y \in \mathbb{R}^n$.

- **The non-negative orthant, $\mathbb{R}^n_+$.** The non-negative orthant consists of all vectors in $\mathbb{R}^n$ whose elements are all non-negative: $\mathbb{R}^n_+ = \{x : x_i \geq 0 \ \ \forall i = 1, \ldots, n\}$. To show that this is a convex set, simply note that given any $x, y \in \mathbb{R}^n_+$ and $0 \leq \theta \leq 1$,

$$(\theta x + (1-\theta)y)_i = \theta x_i + (1-\theta)y_i \geq 0 \ \ \forall i.$$

- **Norm balls.** Let $\|\cdot\|$ be some norm on $\mathbb{R}^n$ (e.g., the Euclidean norm, $\|x\|_2 = \sqrt{\sum_{i=1}^n x_i^2}$). Then the set $\{x : \|x\| \leq 1\}$ is a convex set. To see this, suppose $x, y \in \mathbb{R}^n$, with $\|x\| \leq 1, \|y\| \leq 1$, and $0 \leq \theta \leq 1$. Then

$$\|\theta x + (1-\theta)y\| \leq \|\theta x\| + \|(1-\theta)y\| = \theta\|x\| + (1-\theta)\|y\| \leq 1$$

  where we used the triangle inequality and the positive homogeneity of norms.

- **Affine subspaces and polyhedra.** Given a matrix $A \in \mathbb{R}^{m \times n}$ and a vector $b \in \mathbb{R}^m$, an affine subspace is the set $\{x \in \mathbb{R}^n : Ax = b\}$ (note that this could possibly be empty if b is not in the range of A). Similarly, a polyhedron is the (again, possibly empty) set $\{x \in \mathbb{R}^n : Ax \preceq b\}$, where '$\preceq$' here denotes componentwise inequality (i.e., all the entries of Ax are less than or equal to their corresponding element in b).[1] To prove this, first consider $x, y \in \mathbb{R}^n$ such that $Ax = Ay = b$. Then for $0 \leq \theta \leq 1$,

$$A(\theta x + (1-\theta)y) = \theta Ax + (1-\theta)Ay = \theta b + (1-\theta)b = b.$$

  Similarly, for $x, y \in \mathbb{R}^n$ that satisfy $Ax \leq b$ and $Ay \leq b$ and $0 \leq \theta \leq 1$,

$$A(\theta x + (1-\theta)y) = \theta Ax + (1-\theta)Ay \leq \theta b + (1-\theta)b = b.$$

[1] Similarly, for two vectors $x, y \in \mathbb{R}^n$, $x \succeq y$ denotes that each element of X is greater than or equal to the corresponding element in b. Note that sometimes '$\leq$' and '$\geq$' are used in place of '$\preceq$' and '$\succeq$'; the meaning must be determined contextually (i.e., both sides of the inequality will be vectors).

- **Intersections of convex sets.** Suppose $C_1, C_2, \ldots, C_k$ are convex sets. Then their intersection
$$\bigcap_{i=1}^{k} C_i = \{x : x \in C_i \ \ \forall i = 1, \ldots, k\}$$
is also a convex set. To see this, consider $x, y \in \bigcap_{i=1}^{k} C_i$ and $0 \leq \theta \leq 1$. Then,
$$\theta x + (1-\theta)y \in C_i \ \ \forall i = 1, \ldots, k$$
by the definition of a convex set. Therefore
$$\theta x + (1-\theta)y \in \bigcap_{i=1}^{k} C_i.$$
Note, however, that the *union* of convex sets in general will not be convex.

- **Positive semidefinite matrices.** The set of all symmetric positive semidefinite matrices, often times called the *positive semidefinite cone* and denoted $\mathbb{S}_+^n$, is a convex set (in general, $\mathbb{S}^n \subset \mathbb{R}^{n \times n}$ denotes the set of symmetric $n \times n$ matrices). Recall that a matrix $A \in \mathbb{R}^{n \times n}$ is symmetric positive semidefinite if and only if $A = A^T$ and for all $x \in \mathbb{R}^n$, $x^T A x \geq 0$. Now consider two symmetric positive semidefinite matrices $A, B \in \mathbb{S}_+^n$ and $0 \leq \theta \leq 1$. Then for any $x \in \mathbb{R}^n$,
$$x^T(\theta A + (1-\theta)B)x = \theta x^T A x + (1-\theta) x^T B x \geq 0.$$
The same logic can be used to show that the sets of all positive definite, negative definite, and negative semidefinite matrices are each also convex.

3 Convex Functions

A central element in convex optimization is the notion of a ***convex function***.

Definition 3.1 *A function $f : \mathbb{R}^n \to \mathbb{R}$ is convex if its domain (denoted $\mathcal{D}(f)$) is a convex set, and if, for all $x, y \in \mathcal{D}(f)$ and $\theta \in \mathbb{R}$, $0 \leq \theta \leq 1$,*
$$f(\theta x + (1-\theta)y) \leq \theta f(x) + (1-\theta) f(y).$$

Intuitively, the way to think about this definition is that if we pick any two points on the graph of a convex function and draw a straight line between then, then the portion of the function between these two points will lie below this straight line. This situation is pictured in Figure 2.[2]

We say a function is ***strictly convex*** if Definition 3.1 holds with strict inequality for $x \neq y$ and $0 < \theta < 1$. We say that f is ***concave*** if $-f$ is convex, and likewise that f is ***strictly concave*** if $-f$ is strictly convex.

[2]Don't worry too much about the requirement that the domain of f be a convex set. This is just a technicality to ensure that $f(\theta x + (1-\theta)y)$ is actually defined (if $\mathcal{D}(f)$ were not convex, then it could be that $f(\theta x + (1-\theta)y)$ is undefined even though $x, y \in \mathcal{D}(f)$).

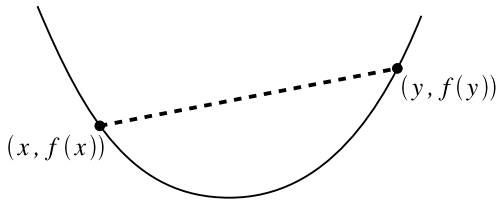


Figure 2: Graph of a convex function. By the definition of convex functions, the line connecting two points on the graph must lie above the function.

3.1 First Order Condition for Convexity

Suppose a function $f : \mathbb{R}^n \to \mathbb{R}$ is differentiable (i.e., the gradient[3] $\nabla_x f(x)$ exists at all points x in the domain of f). Then f is convex if and only if $\mathcal{D}(f)$ is a convex set and for all $x, y \in \mathcal{D}(f)$,

$$f(y) \geq f(x) + \nabla_x f(x)^T (y - x).$$

The function $f(x) + \nabla_x f(x)^T (y - x)$ is called the ***first-order approximation*** to the function f at the point x. Intuitively, this can be thought of as approximating f with its tangent line at the point x. The first order condition for convexity says that f is convex if and only if the tangent line is a global underestimator of the function f. In other words, if we take our function and draw a tangent line at any point, then every point on this line will lie below the corresponding point on f.

Similar to the definition of convexity, f will be strictly convex if this holds with strict inequality, concave if the inequality is reversed, and strictly concave if the reverse inequality is strict.

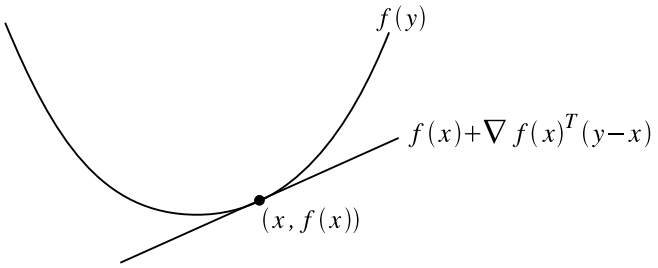


Figure 3: Illustration of the first-order condition for convexity.

[3]Recall that the gradient is defined as $\nabla_x f(x) \in \mathbb{R}^n, (\nabla_x f(x))_i = \frac{\partial f(x)}{\partial x_i}$. For a review on gradients and Hessians, see the previous section notes on linear algebra.

3.2 Second Order Condition for Convexity

Suppose a function $f : \mathbb{R}^n \to \mathbb{R}$ is twice differentiable (i.e., the Hessian[4] $\nabla_x^2 f(x)$ is defined for all points x in the domain of f). Then f is convex if and only if $\mathcal{D}(f)$ is a convex set and its Hessian is positive semidefinite: i.e., for any $x \in \mathcal{D}(f)$,

$$\nabla_x^2 f(x) \succeq 0.$$

Here, the notation '$\succeq$' when used in conjunction with matrices refers to positive semidefiniteness, rather than componentwise inequality. [5] In one dimension, this is equivalent to the condition that the second derivative $f''(x)$ always be positive (i.e., the function always has positive curvature).

Again analogous to both the definition and first order conditions for convexity, f is strictly convex if its Hessian is positive definite, concave if the Hessian is negative semidefinite, and strictly concave if the Hessian is negative definite.

3.3 Jensen's Inequality

Suppose we start with the inequality in the basic definition of a convex function

$$f(\theta x + (1-\theta)y) \leq \theta f(x) + (1-\theta)f(y) \quad \text{for} \quad 0 \leq \theta \leq 1.$$

Using induction, this can be fairly easily extended to convex combinations of more than one point,

$$f\left(\sum_{i=1}^{k} \theta_i x_i\right) \leq \sum_{i=1}^{k} \theta_i f(x_i) \quad \text{for} \quad \sum_{i=1}^{k} \theta_i = 1,\ \theta_i \geq 0\ \ \forall i.$$

In fact, this can also be extended to infinite sums or integrals. In the latter case, the inequality can be written as

$$f\left(\int p(x)x dx\right) \leq \int p(x)f(x)dx \quad \text{for} \quad \int p(x)dx = 1,\ p(x) \geq 0\ \ \forall x.$$

Because $p(x)$ integrates to 1, it is common to consider it as a probability density, in which case the previous equation can be written in terms of expectations,

$$f(\mathbf{E}[x]) \leq \mathbf{E}[f(x)].$$

This last inequality is known as *Jensen's inequality*, and it will come up later in class.[6]

[4] Recall the Hessian is defined as $\nabla_x^2 f(x) \in \mathbb{R}^{n \times n}, (\nabla_x^2 f(x))_{ij} = \frac{\partial^2 f(x)}{\partial x_i \partial x_j}$

[5] Similarly, for a symmetric matrix $X \in \mathbb{S}^n$, $X \preceq 0$ denotes that X is negative semidefinite. As with vector inequalities, '$\leq$' and '$\geq$' are sometimes used in place of '$\preceq$' and '$\succeq$'. Despite their notational similarity to vector inequalities, these concepts are very different; in particular, $X \succeq 0$ does not imply that $X_{ij} \geq 0$ for all i and j.

[6] In fact, all four of these equations are sometimes referred to as Jensen's inequality, due to the fact that they are all equivalent. However, for this class we will use the term to refer specifically to the last inequality presented here.

3.4 Sublevel Sets

Convex functions give rise to a particularly important type of convex set called an ***α-sublevel set***. Given a convex function $f : \mathbb{R}^n \to R$ and a real number $\alpha \in \mathbb{R}$, the α-sublevel set is defined as

$$\{x \in \mathcal{D}(f) : f(x) \leq \alpha\}.$$

In other words, the α-sublevel set is the set of all points x such that $f(x) \leq \alpha$.

To show that this is a convex set, consider any $x, y \in \mathcal{D}(f)$ such that $f(x) \leq \alpha$ and $f(y) \leq \alpha$. Then

$$f(\theta x + (1-\theta)y) \leq \theta f(x) + (1-\theta)f(y) \leq \theta\alpha + (1-\theta)\alpha = \alpha.$$

3.5 Examples

We begin with a few simple examples of convex functions of one variable, then move on to multivariate functions.

- **Exponential.** Let $f : \mathbb{R} \to \mathbb{R}$, $f(x) = e^{ax}$ for any $a \in \mathbb{R}$. To show f is convex, we can simply take the second derivative $f''(x) = a^2 e^{ax}$, which is positive for all x.

- **Negative logarithm.** Let $f : \mathbb{R} \to \mathbb{R}$, $f(x) = -\log x$ with domain $\mathcal{D}(f) = \mathbb{R}_{++}$ (here, $\mathbb{R}_{++}$ denotes the set of strictly positive real numbers, $\{x : x > 0\}$). Then $f''(x) = 1/x^2 > 0$ for all x.

- **Affine functions.** Let $f : \mathbb{R}^n \to \mathbb{R}$, $f(x) = b^T x + c$ for some $b \in \mathbb{R}^n$, $c \in \mathbb{R}$. In this case the Hessian, $\nabla_x^2 f(x) = 0$ for all x. Because the zero matrix is both positive semidefinite and negative semidefinite, f is both convex and concave. In fact, affine functions of this form are the *only* functions that are both convex and concave.

- **Quadratic functions.** Let $f : \mathbb{R}^n \to \mathbb{R}$, $f(x) = \frac{1}{2}x^T A x + b^T x + c$ for a symmetric matrix $A \in \mathbb{S}^n$, $b \in \mathbb{R}^n$ and $c \in \mathbb{R}$. In our previous section notes on linear algebra, we showed the Hessian for this function is given by

  $$\nabla_x^2 f(x) = A.$$

  Therefore, the convexity or non-convexity of f is determined entirely by whether or not A is positive semidefinite: if A is positive semidefinite then the function is convex (and analogously for strictly convex, concave, strictly concave). If A is indefinite then f is neither convex nor concave.

  Note that the squared Euclidean norm $f(x) = \|x\|_2^2 = x^T x$ is a special case of quadratic functions where $A = I$, $b = 0$, $c = 0$, so it is therefore a strictly convex function.

- **Norms.** Let $f : \mathbb{R}^n \to \mathbb{R}$ be some norm on $\mathbb{R}^n$. Then by the triangle inequality and positive homogeneity of norms, for $x, y \in \mathbb{R}^n$, $0 \leq \theta \leq 1$,

$$f(\theta x + (1-\theta)y) \leq f(\theta x) + f((1-\theta)y) = \theta f(x) + (1-\theta)f(y).$$

  This is an example of a convex function where it is *not* possible to prove convexity based on the second or first order conditions, because norms are not generally differentiable everywhere (e.g., the 1-norm, $||x||_1 = \sum_{i=1}^n |x_i|$, is non-differentiable at all points where any x_i is equal to zero).

- **Nonnegative weighted sums of convex functions.** Let $f_1, f_2, \ldots, f_k$ be convex functions and $w_1, w_2, \ldots, w_k$ be nonnegative real numbers. Then

$$f(x) = \sum_{i=1}^k w_i f_i(x)$$

  is a convex function, since

$$\begin{aligned} f(\theta x + (1-\theta)y) &= \sum_{i=1}^k w_i f_i(\theta x + (1-\theta)y) \\ &\leq \sum_{i=1}^k w_i(\theta f_i(x) + (1-\theta)f_i(y)) \\ &= \theta \sum_{i=1}^k w_i f_i(x) + (1-\theta)\sum_{i=1}^k w_i f_i(y) \\ &= \theta f(x) + (1-\theta)f(x). \end{aligned}$$

4 Convex Optimization Problems

Armed with the definitions of convex functions and sets, we are now equipped to consider ***convex optimization problems***. Formally, a convex optimization problem in an optimization problem of the form

$$\begin{array}{ll} \text{minimize} & f(x) \\ \text{subject to} & x \in C \end{array}$$

where f is a convex function, C is a convex set, and x is the optimization variable. However, since this can be a little bit vague, we often write it often written as

$$\begin{array}{lll} \text{minimize} & f(x) & \\ \text{subject to} & g_i(x) \leq 0, & i = 1, \ldots, m \\ & h_i(x) = 0, & i = 1, \ldots, p \end{array}$$

where f is a convex function, g_i are convex functions, and h_i are affine functions, and x is the optimization variable.

Is it imporant to note the direction of these inequalities: a convex function g_i must be *less* than zero. This is because the 0-sublevel set of g_i is a convex set, so the feasible region, which is the intersection of many convex sets, is also convex (recall that affine subspaces are convex sets as well). If we were to require that $g_i \geq 0$ for some convex g_i, the feasible region would no longer be a convex set, and the algorithms we apply for solving these problems would not longer be guaranteed to find the global optimum. Also notice that only affine functions are allowed to be equality constraints. Intuitively, you can think of this as being due to the fact that an equality constraint is equivalent to the two inequalities $h_i \leq 0$ and $h_i \geq 0$. However, these will both be valid constraints if and only if h_i is both convex and concave, i.e., h_i must be affine.

The ***optimal value*** of an optimization problem is denoted $p^\star$ (or sometimes $f^\star$) and is equal to the minimum possible value of the objective function in the feasible region[7]

$$p^\star = \min\{f(x) : g_i(x) \leq 0, i = 1, \ldots, m, h_i(x) = 0, i = 1, \ldots, p\}.$$

We allow $p^\star$ to take on the values $+\infty$ and $-\infty$ when the problem is either *infeasible* (the feasible region is empty) or *unbounded below* (there exists feasible points such that $f(x) \to -\infty$), respectively. We say that $x^\star$ is an ***optimal point*** if $f(x^\star) = p^\star$. Note that there can be more than one optimal point, even when the optimal value is finite.

4.1 Global Optimality in Convex Problems

Before stating the result of global optimality in convex problems, let us formally define the concepts of local optima and global optima. Intuitively, a feasible point is called ***locally optimal*** if there are no "nearby" feasible points that have a lower objective value. Similarly, a feasible point is called ***globally optimal*** if there are no feasible points at all that have a lower objective value. To formalize this a little bit more, we give the following two definitions.

Definition 4.1 *A point x is locally optimal if it is feasible (i.e., it satisfies the constraints of the optimization problem) and if there exists some $R > 0$ such that all feasible points z with $\|x - z\|_2 \leq R$, satisfy $f(x) \leq f(z)$.*

Definition 4.2 *A point x is globally optimal if it is feasible and for all feasible points z, $f(x) \leq f(z)$.*

We now come to the crucial element of convex optimization problems, from which they derive most of their utility. The key idea is that ***for a convex optimization problem all locally optimal points are globally optimal***.

Let's give a quick proof of this property by contradiction. Suppose that x is a locally optimal point which is not globally optimal, i.e., there exists a feasible point y such that

[7]Math majors might note that the min appearing below should more correctly be an inf. We won't worry about such technicalities here, and use min for simplicity.

$f(x) > f(y)$. By the definition of local optimality, there exist no feasible points z such that $\|x - z\|_2 \leq R$ and $f(z) < f(x)$. But now suppose we choose the point

$$z = \theta y + (1 - \theta)x \quad \text{with} \quad \theta = \frac{R}{2\|x - y\|_2}.$$

Then

$$\begin{aligned} \|x - z\|_2 &= \left\| x - \left(\frac{R}{2\|x - y\|_2} y + \left(1 - \frac{R}{2\|x - y\|_2} \right) x \right) \right\|_2 \\ &= \left\| \frac{R}{2\|x - y\|_2} (x - y) \right\|_2 \\ &= R/2 \leq R. \end{aligned}$$

In addition, by the convexity of f we have

$$f(z) = f(\theta y + (1 - \theta)x) \leq \theta f(y) + (1 - \theta) f(x) < f(x).$$

Furthermore, since the feasible set is a convex set, and since x and y are both feasible $z = \theta y + (1 - \theta)$ will be feasible as well. Therefore, z is a feasible point, with $\|x - z\|_2 < R$ and $f(z) < f(x)$. This contradicts our assumption, showing that x cannot be locally optimal.

4.2 Special Cases of Convex Problems

For a variety of reasons, it is often times convenient to consider special cases of the general convex programming formulation. For these special cases we can often devise extremely efficient algorithms that can solve very large problems, and because of this you will probably see these special cases referred to any time people use convex optimization techniques.

- **Linear Programming.** We say that a convex optimization problem is a ***linear program*** (LP) if both the objective function f and inequality constraints g_i are affine functions. In other words, these problems have the form

$$\begin{aligned} \text{minimize} \quad & c^T x + d \\ \text{subject to} \quad & Gx \preceq h \\ & Ax = b \end{aligned}$$

  where $x \in \mathbb{R}^n$ is the optimization variable, $c \in \mathbb{R}^n$, $d \in \mathbb{R}$, $G \in \mathbb{R}^{m \times n}$, $h \in \mathbb{R}^m$, $A \in \mathbb{R}^{p \times n}$, $b \in \mathbb{R}^p$ are defined by the problem, and '$\preceq$' denotes elementwise inequality.

- **Quadratic Programming.** We say that a convex optimization problem is a ***quadratic program*** (QP) if the inequality constraints g_i are still all affine, but if the objective function f is a convex quadratic function. In other words, these problems have the form,

$$\begin{aligned} \text{minimize} \quad & \tfrac{1}{2} x^T P x + c^T x + d \\ \text{subject to} \quad & Gx \preceq h \\ & Ax = b \end{aligned}$$

where again $x \in \mathbb{R}^n$ is the optimization variable, $c \in \mathbb{R}^n$, $d \in \mathbb{R}$, $G \in \mathbb{R}^{m \times n}$, $h \in \mathbb{R}^m$, $A \in \mathbb{R}^{p \times n}$, $b \in \mathbb{R}^p$ are defined by the problem, but we also have $P \in \mathbb{S}^n_+$, a symmetric positive semidefinite matrix.

- **Quadratically Constrained Quadratic Programming.** We say that a convex optimization problem is a ***quadratically constrained quadratic program*** (QCQP) if both the objective f and the inequality constraints g_i are convex quadratic functions,

$$\begin{array}{ll} \text{minimize} & \frac{1}{2}x^T P x + c^T x + d \\ \text{subject to} & \frac{1}{2}x^T Q_i x + r_i^T x + s_i \leq 0, \quad i = 1, \ldots, m \\ & Ax = b \end{array}$$

  where, as before, $x \in \mathbb{R}^n$ is the optimization variable, $c \in \mathbb{R}^n$, $d \in \mathbb{R}$, $A \in \mathbb{R}^{p \times n}$, $b \in \mathbb{R}^p$, $P \in \mathbb{S}^n_+$, but we also have $Q_i \in \mathbb{S}^n_+$, $r_i \in \mathbb{R}^n$, $s_i \in \mathbb{R}$, for $i = 1, \ldots, m$.

- **Semidefinite Programming.** This last example is a bit more complex than the previous ones, so don't worry if it doesn't make much sense at first. However, semidefinite programming is become more and more prevalent in many different areas of machine learning research, so you might encounter these at some point, and it is good to have an idea of what they are. We say that a convex optimization problem is a ***semidefinite program*** (SDP) if it is of the form

$$\begin{array}{ll} \text{minimize} & \text{tr}(CX) \\ \text{subject to} & \text{tr}(A_i X) = b_i, \quad i = 1, \ldots, p \\ & X \succeq 0 \end{array}$$

  where the symmetric matrix $X \in \mathbb{S}^n$ is the optimization variable, the symmetric matrices $C, A_1, \ldots, A_p \in \mathbb{S}^n$ are defined by the problem, and the constraint $X \succeq 0$ means that we are constraining X to be positive semidefinite. This looks a bit different than the problems we have seen previously, since the optimization variable is now a matrix instead of a vector. If you are curious as to why such a formulation might be useful, you should look into a more advanced course or book on convex optimization.

It should be fairly obvious from the definitions that quadratic programs are more general than linear programs (since a linear program is just a special case of a quadratic program where $P = 0$), and likewise that quadratically constrained quadratic programs are more general than quadratic programs. However, what is not obvious at all is that semidefinite programs are in fact more general than all the previous types. That is, any quadratically constrained quadratic program (and hence any quadratic program or linear program) can be expressed as a semidefinte program. We won't discuss this relationship further in this document, but this might give you just a small idea as to why semidefinite programming could be useful.

4.3 Examples

Now that we've covered plenty of the boring math and formalisms behind convex optimization, we can finally get to the fun part: using these techniques to solve actual problems. We've already encountered a few such optimization problems in class, and in nearly every field, there is a good chance that someone has tried to apply convex optimization to solve some problem.

- **Support Vector Machines.** One of the most prevalent applications of convex optimization methods in machine learning is the support vector machine classifier. As discussed in class, finding the support vector classifier (in the case with slack variables) can be formulated as the optimization problem

$$\begin{array}{ll} \text{minimize} & \frac{1}{2}\|w\|_2^2 + C\sum_{i=1}^m \xi_i \\ \text{subject to} & y^{(i)}(w^T x^{(i)} + b) \geq 1 - \xi_i, \quad i = 1, \ldots, m \\ & \xi_i \geq 0, \qquad\qquad\qquad\quad i = 1, \ldots, m \end{array}$$

with optimization variables $w \in \mathbb{R}^n$, $\xi \in \mathbb{R}^m$, $b \in \mathbb{R}$, and where $C \in \mathbb{R}$ and $x^{(i)}, y^{(i)}, i = 1, \ldots m$ are defined by the problem. This is an example of a quadratic program, which we try to put the problem into the form described in the previous section. In particular, if define $k = m + n + 1$, let the optimization variable be

$$x \in \mathbb{R}^k \equiv \begin{bmatrix} w \\ \xi \\ b \end{bmatrix}$$

and define the matrices

$$P \in \mathbb{R}^{k \times k} = \begin{bmatrix} I & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \quad c \in \mathbb{R}^k = \begin{bmatrix} 0 \\ C \cdot \mathbf{1} \\ 0 \end{bmatrix},$$

$$G \in \mathbb{R}^{2m \times k} = \begin{bmatrix} -\text{diag}(y)X & -I & -y \\ 0 & -I & 0 \end{bmatrix}, \quad h \in \mathbb{R}^{2m} = \begin{bmatrix} -\mathbf{1} \\ 0 \end{bmatrix}$$

where I is the identity, $\mathbf{1}$ is the vector of all ones, and X and y are defined as in class,

$$X \in \mathbb{R}^{m \times n} = \begin{bmatrix} x^{(1)^T} \\ x^{(2)^T} \\ \vdots \\ x^{(m)^T} \end{bmatrix}, \quad y \in \mathbb{R}^m = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{bmatrix}.$$

You should try to convince yourself that the quadratic program described in the previous section, when using these matrices defined above, is equivalent to the SVM optimization problem. In reality, it is fairly easy to see that there the SVM optimization problem has a quadratic objective and linear constraints, so we typically don't need to put it into standard form to "prove" that it is a QP, and would only do so if we are using an off-the-shelf solver that requires the input to be in standard form.

- **Constrained least squares.** In class we have also considered the least squares problem, where we want to minimize $\|Ax - b\|_2^2$ for some matrix $A \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^m$. As we saw, this particular problem can actually be solved analytically via the normal equations. However, suppose that we also want to constrain the entries in the solution x to lie within some predefined ranges. In other words, suppose we weanted to solve the optimization problem,

$$\begin{array}{ll} \text{minimize} & \frac{1}{2}\|Ax - b\|_2^2 \\ \text{subject to} & l \preceq x \preceq u \end{array}$$

  with optimization variable x and problem data $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$, $l \in \mathbb{R}^n$, and $u \in \mathbb{R}^n$. This might seem like a fairly simple additional constraint, but it turns out that there will no longer be an analytical solution. However, you should be able to convince yourself that this optimization problem is a quadratic program, with matrices defined by

$$P \in \mathbb{R}^{n \times n} = \frac{1}{2}A^T A, \;\; c \in \mathbb{R}^n = -b^T A, \;\; d \in \mathbb{R} = \frac{1}{2}b^T b,$$
$$G \in \mathbb{R}^{2n \times 2n} = \begin{bmatrix} -I & 0 \\ 0 & I \end{bmatrix}, \;\; h \in \mathbb{R}^{2n} = \begin{bmatrix} -l \\ u \end{bmatrix}.$$

- **Maximum Likelihood for Logistic Regression.** For homework one, you were required to show that the log-likelihood of the data in a logistic model was concave. This log likehood under such a model is

$$\ell(\theta) = \sum_{i=1}^{n} \left\{ y^{(i)} \ln g(\theta^T x^{(i)}) + (1 - y^{(i)}) \ln(1 - g(\theta^T x^{(i)})) \right\}$$

  where $g(z)$ denotes the logistic function $g(z) = 1/(1 + e^{-z})$. Finding the maximum likelihood estimate is then a task of maximizing the log-likelihood (or equivalently, minimizing the negative log-likelihood, a convex function), i.e.,

$$\begin{array}{ll} \text{minimize} & -\ell(\theta) \end{array}$$

  with optimization variable $\theta \in \mathbb{R}^n$ and no constraints.

  Unlike the previous two examples, it turns out that it is not so easy to put this problem into a "standard" form optimization problem. Nevertheless, you've seen on the homework that the fact that ℓ is a concave function means that you can very efficiently find the global solution using an algorithm such as Newton's method.

References

[1] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge UP, 2004. Online: `http://www.stanford.edu/~boyd/cvxbook/`

Convex Optimization Overview (cnt'd)

Chuong B. Do

October 26, 2007

1 Recap

During last week's section, we began our study of ***convex optimization***, the study of mathematical optimization problems of the form,

$$\begin{array}{ll} \underset{x \in \mathbb{R}^n}{\text{minimize}} & f(x) \\ \text{subject to} & g_i(x) \leq 0, \quad i = 1, \ldots, m, \\ & h_i(x) = 0, \quad i = 1, \ldots, p, \end{array} \tag{1}$$

where $x \in \mathbb{R}^n$ is the optimization variable, $f : \mathbb{R}^n \to \mathbb{R}$ and $g_i : \mathbb{R}^n \to \mathbb{R}$ are convex functions, and $h_i : \mathbb{R}^n \to \mathbb{R}$ are affine functions. In a convex optimization problem, the convexity of both the objective function f and the feasible region (i.e., the set of x's satisfying all constraints) allows us to conclude that any feasible locally optimal point must also be globally optimal. This fact provides the key intuition for why convex optimization problems can in general be solved efficiently.

In these lecture notes, we continue our foray into the field of convex optimization. In particular, we will introduce the theory of Lagrange duality for convex optimization problems with inequality and equality constraints. We will also discuss generic yet efficient algorithms for solving convex optimization problems, and then briefly mention directions for further exploration.

2 Duality

To explain the fundamental ideas behind duality theory, we start with a motivating example based on CS 229 homework grading. We prove a simple weak duality result in this setting, and then relate it to duality in optimization. We then discuss strong duality and the KKT optimality conditions.

2.1 A motivating example: CS 229 homework grading

In CS 229, students must complete four homeworks throughout the quarter, each consisting of five questions apiece. Suppose that during one year that the course is offered, the TAs

decide to economize on their work load for the quarter by grading only one problem on each submitted problem set. Nevertheless, they also require that every student submit an attempted solution to every problem (a requirement which, if violated, would lead to automatic failure of the course).

Because they are extremely cold-hearted[1], the TAs always try to ensure that the students lose as many points as possible; if the TAs grade a problem that the student did not attempt, the number of points lost is set to $+\infty$ to denote automatic failure in the course. Conversely, each student in the course seeks to minimize the number of points lost on his or her assignments, and thus must decide on a strategy—i.e., an allocation of time to problems—that minimizes the number of points lost on the assignment.

The struggle between student and TAs can be summarized in a matrix $A = (a_{ij}) \in \mathbb{R}^{n \times m}$, whose columns correspond to different problems that the TAs might grade, and whose rows correspond to different strategies for time allocation that the student might use for the problem set. For example, consider the following matrix,

$$A = \begin{bmatrix} 5 & 5 & 5 & 5 & 5 \\ 8 & 8 & 1 & 8 & 8 \\ +\infty & +\infty & +\infty & 0 & +\infty \end{bmatrix},$$

Here, the student must decide between three strategies (corresponding to the three rows of the matrix, A):

- $i = 1$: she invests an equal effort into all five problems and hence loses at most 5 points on each problem,
- $i = 2$: she invests more time into problem 3 than the other four problems, and
- $i = 3$: she skips four problems in order to guarantee no points lost on problem 4.

Similarly, the TAs must decide between five strategies ($j \in \{1, 2, 3, 4, 5\}$) corresponding to the choice of problem graded.

If the student is forced to submit the homework without knowing the TAs choice of problem to be graded, and if the TAs are allowed to decide which problem to grade after having seen the student's problem set, then the number of points she loses will be:

$$p^* = \min_i \max_j a_{ij} \qquad (= 5 \text{ in the example above}) \tag{P}$$

where the order of the minimization and maximization reflect that for each fixed student time allocation strategy i, the TAs will have the opportunity to choose the worst scoring problem $\max_j a_{ij}$ to grade. However, if the TAs announce beforehand which homework problem will be graded, then the the number of points lost will be:

$$d^* = \max_j \min_i a_{ij} \qquad (= 0 \text{ in the example above}) \tag{D}$$

where this time, for each possible announced homework problem j to be graded, the student will have the opportunity to choose the optimal time allocation strategy, $\min_i a_{ij}$, which loses

[1] Clearly, this is a fictional example. The CS 229 TAs want you to succeed. Really, we do.

her the fewest points. Here, (P) is called the ***primal*** optimization problem whereas (D) is called the ***dual*** optimization problem. Rows containing $+\infty$ values correspond to strategies where the student has flagrantly violated the TAs demand that all problems be attempted; for reasons, which will become clear later, we refer to these rows as being ***primal-infeasible***.

In the example, the value of the dual problem is lower than that of the primal problem, i.e., $d^* = 0 < 5 = p^*$. This intuitively makes sense: the second player in this adversarial game has the advantage of knowing his/her opponent's strategy. This principle, however, holds more generally:

Theorem 2.1 (Weak duality)**.** *For any matrix $A = (a_{ij}) \in \mathbb{R}^{m \times n}$, it is always the case that*

$$\max_j \min_i a_{ij} = d^* \leq p^* = \min_i \max_j a_{ij}.$$

Proof. Let (i_d, j_d) be the row and column associated with d^*, and let (i_p, j_p) be the row and column associated with p^*. We have,

$$d^* = a_{i_d j_d} \leq a_{i_p j_d} \leq a_{i_p j_p} = p^*.$$

Here, the first inequality follows from the fact that $a_{i_d j_d}$ is the smallest element in the j_dth column (i.e., i_d was the strategy chosen by the student after the TAs chose problem j_d, and hence, it must correspond to the fewest points lost in that column). Similarly, the second inequality follow from the fact that $a_{i_p j_p}$ is the largest element in the i_pth row (i.e., j_p was the problem chosen by the TAs after the student picked strategy i_p, so it must correspond to the most points lost in that row). □

2.2 Duality in optimization

The task of constrained optimization, it turns out, relates closely with the adversarial game described in the previous section. To see the connection, first recall our original optimization problem,

$$\begin{array}{ll} \underset{x}{\text{minimize}} & f(x) \\ \text{subject to} & g_i(x) \leq 0, \quad i = 1, \ldots, m, \\ & h_i(x) = 0, \quad i = 1, \ldots, p. \end{array}$$

Define the ***generalized Lagrangian*** to be

$$\mathcal{L}(x, \lambda, \nu) := f(x) + \sum_{i=1}^{m} \lambda_i g_i(x) + \sum_{i=1}^{p} \nu_i h_i(x).$$

Here, the variables λ and ν are called the the ***dual variables*** (or ***Lagrange multipliers***). Analogously, the variables x are known as the ***primal variables***.

The correspondence between primal/dual optimization and game playing can be pictured informally using an infinite matrix whose rows are indexed by $x \in \mathbb{R}^n$ and whose columns

are indexed by $(\lambda, \nu) \in \mathbb{R}^m_+ \times \mathbb{R}^p$ (i.e., $\lambda_i \geq 0$, for $i = 1, \ldots, m$). In particular, we have

$$A = \begin{bmatrix} \ddots & \vdots & \iddots \\ \cdots & \mathcal{L}(x, \lambda, \nu) & \cdots \\ \iddots & \vdots & \ddots \end{bmatrix}$$

Here, the "student" manipulates the primal variables x in order to minimize the Lagrangian $\mathcal{L}(x, \lambda, \nu)$ while the "TAs" manipulate the dual variables (λ, ν) in order to maximize the Lagrangian.

To see the relationship between this game and the original optimization problem, we formulate the following ***primal*** problem:

$$\begin{aligned} p^* &= \min_x \max_{\lambda, \nu : \lambda_i \geq 0} \quad \mathcal{L}(x, \lambda, \nu) \\ &= \min_x \quad \theta_P(x) \end{aligned} \tag{P'}$$

where $\theta_P(x) := \max_{\lambda, \nu : \lambda_i \geq 0} \mathcal{L}(x, \lambda, \nu)$. Computing p^* is equivalent to our original convex optimization primal in the following sense: for any candidate solution x,

- if $g_i(x) > 0$ for some $i \in \{1, \ldots, m\}$, then setting $\lambda_i = \infty$ gives $\theta_P(x) = \infty$.
- if $h_i(x) \neq 0$ for some $i \in \{1, \ldots, m\}$, then setting $\lambda_i = \infty \cdot \text{Sign}(h_i(x))$ gives $\theta_P(x) = \infty$.
- if x is feasible (i.e., x obeys all the constraints of our original optimization problem), then $\theta_P(x) = f(x)$, where the maximum is obtained, for example, by setting all of the λ_i's and ν_i's to zero.

Intuitively then, $\theta_P(x)$ behaves conceptually like an "unconstrained" version of the original constrained optimization problem in which the infeasible region of f is "carved away" by forcing $\theta_P(x) = \infty$ for any infeasible x; thus, only points in the feasible region are left as candidate minimizers. This idea of using penalties to ensure that minimizers stay in the feasible region will come up later when talk about barrier algorithms for convex optimization.

By analogy to the CS 229 grading example, we can form the following ***dual*** problem:

$$\begin{aligned} d^* &= \max_{\lambda, \nu : \lambda_i \geq 0} \min_x \quad \mathcal{L}(x, \lambda, \nu) \\ &= \max_{\lambda, \nu : \lambda_i \geq 0} \quad \theta_D(\lambda, \nu) \end{aligned} \tag{D'}$$

where $\theta_D(\lambda, \nu) := \min_x \mathcal{L}(x, \lambda, \nu)$. Dual problems can often be easier to solve than their corresponding primal problems. In the case of SVMs, for instance, SMO is a dual optimization algorithm which considers joint optimization of pairs of dual variables. Its simple form derives largely from the simplicity of the dual objective and the simplicity of the corresponding constraints on the dual variables. Primal-based SVM solutions are indeed possible, but when the number of training examples is large and the kernel matrix K of inner products $K_{ij} = K(x^{(i)}, x^{(j)})$ is large, dual-based optimization can be considerably more efficient.

Using an argument essentially identical to that presented in Theorem (2.1), we can show that in this setting, we again have $d^* \leq p^*$. This is the property of ***weak duality*** for general optimization problems. Weak duality can be particularly useful in the design of optimization algorithms. For example, suppose that during the course of an optimization algorithm we have a candidate primal solution x and dual-feasible vector (λ, ν) such that $\theta_P(x) - \theta_D(\lambda, \nu) \leq \epsilon$. From weak duality, we have that

$$\theta_D(\lambda, \nu) \leq d^* \leq p^* \leq \theta_P(x),$$

implying that x and (λ, ν) must be ϵ-optimal (i.e., their objective functions differ by no more than ϵ from the objective functions of the true optima x^* and (λ^*, ν^*).

In practice, the dual objective $\theta_D(\lambda, \nu)$ can often be found in closed form, thus allowing the dual problem (D') to depend only on the dual variables λ and ν. When the Lagrangian is differentiable with respect to x, then a closed-form for $\theta_D(\lambda, \nu)$ can often be found by setting the gradient of the Lagrangian to zero, so as to ensure that the Lagrangian is minimized with respect to x.[2] An example derivation of the dual problem for the L_1 soft-margin SVM is shown in the Appendix.

2.3 Strong duality

For any primal/dual optimization problems, weak duality will always hold. In some cases, however, the inequality $d^* \leq p^*$ may be replaced with equality, i.e., $d^* = p^*$; this latter condition is known as ***strong duality***. Strong duality does not hold in general. When it does however, the lower-bound property described in the previous section provide a useful termination criterion for optimization algorithms. In particular, we can design algorithms which simultaneously optimize both the primal and dual problems. Once the candidate solutions x of the primal problem and (λ, ν) of the dual problem obey $\theta_P(x) - \theta_D(\lambda, \nu) \leq \epsilon$, then we know that both solutions are ϵ-accurate. This is guaranteed to happen provided our optimization algorithm works properly, since strong duality guarantees that the optimal primal and dual values are equal.

Conditions which guarantee strong duality for convex optimization problems are known as ***constraint qualifications***. The most commonly invoked constraint qualification, for example, is ***Slater's condition***:

Theorem 2.2. *Consider a convex optimization problem of the form* (1)*, whose corresponding primal and dual problems are given by* (P') *and* (D')*. If there exists a primal feasible x for*

[2] Often, differentiating the Lagrangian with respect to x leads to the generation of additional requirements on dual variables that must hold at any fixed point of the Lagrangian with respect to x. When these constraints are not satisfied, one can show that the Lagrangian is unbounded below (i.e., $\theta_D(\lambda, \nu) = -\infty$).

Since such points are clearly not optimal solutions for the dual problem, we can simply exclude them from the domain of the dual problem altogether by adding the derived constraints to the existing constraints of the dual problem. An example of this is the derived constraint, $\sum_{i=1}^{m} \alpha_i y^{(i)} = 0$, in the SVM formulation. This procedure of incorporating derived constraints into the dual problem is known as ***making dual constraints explicit*** (see [1], page 224).

which each inequality constraint is strictly satisfied (i.e., $g_i(x) < 0$), then $d^* = p^*$.[3]

The proof of this theorem is beyond the scope of this course. We will, however, point out its application to the soft-margin SVMs described in class. Recall that soft-margin SVMs were found by solving

$$\begin{array}{ll} \underset{w,b,\xi}{\text{minimize}} & \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{m}\xi_i \\ \text{subject to} & y^{(i)}(w^T x^{(i)} + b) \geq 1 - \xi_i, \quad i = 1, \ldots, m, \\ & \xi_i \geq 0, \qquad\qquad\qquad\quad i = 1, \ldots, m. \end{array}$$

Slater's condition applies provided we can find at least one primal feasible setting of w, b, and ξ where all inequalities are strict. It is easy to verify that $w = \mathbf{0}$, $b = 0$, $\xi = 2 \cdot \mathbf{1}$ satisfies these conditions (where $\mathbf{0}$ and $\mathbf{1}$ denote the vector of all 0's and all 1's, respectively), since

$$y^{(i)}(w^T x^{(i)} + b) = y^{(i)}(\mathbf{0}^T x^{(i)} + 0) = 0 > -1 = 1 - 2 = 1 - \xi_i, \qquad i = 1, \ldots, m,$$

and the remaining m inequalities are trivially strictly satisfied. Hence, strong duality holds, so the optimal values of the primal and dual soft-margin SVM problems will be equal.

2.4 The KKT conditions

In the case of differentiable unconstrained convex optimization problems, setting the gradient to "zero" provides a simple means for identifying candidate local optima. For constrained convex programming, do similar criteria exist for characterizing the optima of primal/dual optimization problems? The answer, it turns out, is provided by a set of requirements known as the ***Karush-Kuhn-Tucker (KKT) necessary and sufficient conditions*** (see [1], pages 242-244).

Suppose that the constraint functions $g_1, \ldots, g_m, h_1, \ldots, h_p$ are not only convex (the h_i's must be affine) but also differentiable.

Theorem 2.3. *If $\tilde{x}$ is primal feasible and $(\tilde{\lambda}, \tilde{\nu})$ are dual feasible, and if*

$$\nabla_x \mathcal{L}(\tilde{x}, \tilde{\lambda}, \tilde{\nu}) = \mathbf{0}, \tag{KKT1}$$

$$\tilde{\lambda}_i g_i(\tilde{x}) = 0, \qquad i = 1, \ldots, m, \tag{KKT2}$$

then $\tilde{x}$ is primal optimal, $(\tilde{\lambda}, \tilde{\nu})$ are dual optimal, and strong duality holds.

Theorem 2.4. *If Slater's condition holds, then conditions of Theorem 2.3 are necessary for any (x^*, λ^*, ν^*) such that x^* is primal optimal and (λ^*, ν^*) are dual feasible.*

[3] One can actually show a more general version of Slater's inequality, which requires only strict satisfaction of non-affine inequality constraints (but allowing affine inequalities to be satisfied with equality). See [1], page 226.

(KKT1) is the standard gradient stationarity condition found for unconstrained differentiable optimization problems. The set of inequalities corresponding to (KKT2) are known as the ***KKT complementarity (or complementary slackness) conditions***. In particular, if x^* is primal optimal and (λ^*, ν^*) is dual optimal, then (KKT2) implies that

$$\begin{aligned} \lambda_i^* > 0 \quad &\Rightarrow \quad g_i(x^*) = 0 \\ g_i(x^*) < 0 \quad &\Rightarrow \quad \lambda_i^* = 0 \end{aligned}$$

That is, whenever λ_i^* is greater than zero, its corresponding inequality constraint must be tight; conversely, any strictly satisfied inequality must have have λ_i^* equal to zero. Thus, we can interpret the dual variables λ_i^* as measuring the "importance" of a particular constraint in characterizing the optimal point.

This interpretation provides an intuitive explanation for the difference between hard-margin and soft-margin SVMs. Recall the dual problems for a hard-margin SVM:

$$\begin{aligned} \underset{\alpha,\beta}{\text{maximize}} \quad & \sum_{i=1}^m \alpha_i - \frac{1}{2} \sum_{i=1}^m \sum_{j=1}^m \alpha_i \alpha_i y^{(i)} y^{(j)} \langle x^{(i)}, x^{(j)} \rangle \\ \text{subject to} \quad & \alpha_i \geq 0, & i = 1, \ldots, m, \\ & \sum_{i=1}^m \alpha_i y^{(i)} = 0, \end{aligned} \tag{2}$$

and the L_1 soft-margin SVM:

$$\begin{aligned} \underset{\alpha,\beta}{\text{maximize}} \quad & \sum_{i=1}^m \alpha_i - \frac{1}{2} \sum_{i=1}^m \sum_{j=1}^m \alpha_i \alpha_i y^{(i)} y^{(j)} \langle x^{(i)}, x^{(j)} \rangle \\ \text{subject to} \quad & 0 \leq \alpha_i \leq C, & i = 1, \ldots, m, \\ & \sum_{i=1}^m \alpha_i y^{(i)} = 0. \end{aligned} \tag{3}$$

Note that the only difference in the soft-margin formulation is the introduction of upper bounds on the dual variables α_i. Effectively, this upper bound constraint limits the influence of any single primal inequality constraint (i.e., any single training example) on the decision boundary, leading to improved robustness for the L_1 soft-margin model.

What consequences do the KKT conditions have for practical optimization algorithms? When Slater's conditions hold, then the KKT conditions are both necessary and sufficient for primal/dual optimality of a candidate primal solution $\tilde{x}$ and a corresponding dual solution $(\tilde{\lambda}, \tilde{\nu})$. Therefore, many optimization algorithms work by trying to guarantee that the KKT conditions are satisfied; the SMO algorithm, for instance, works by iteratively identifying Lagrange multipliers for which the corresponding KKT conditions are unsatisfied and then "fixing" KKT complementarity.[4]

[4]See [1], pages 244-245 for an example of an optimization problem where the KKT conditions can be solved directly, thus skipping the need for primal/dual optimization altogether.

3 Algorithms for convex optimization

Thus far, we have talked about convex optimization problems and their properties. But how does one solve a convex optimization problem in practice? In this section, we describe a generic strategy for solving convex optimization problems known as the ***interior-point*** method. This method combines a safe-guarded variant of Newton's algorithm with a "barrier" technique for enforcing inequality constraints.

3.1 Unconstrained optimization

We consider first the problem of unconstrained optimization, i.e.,

$$\underset{x}{\text{minimize}} \quad f(x).$$

In Newton's algorithm for unconstrained optimization, we consider the Taylor approximation $\tilde{f}$ of the function f, centered at the current iterate x_t. Discarding terms of higher order than two, we have

$$\tilde{f}(x) = f(x_t) + \nabla_x f(x_t)^T (x - x_t) + \frac{1}{2}(x - x_t)\nabla_x^2 f(x_t)(x - x_t).$$

To minimize $\tilde{f}(x)$, we can set its gradient to zero. In particular, if x_{nt} denotes the minimum of $\tilde{f}(x)$, then

$$\begin{aligned}
\nabla_x f(x_t) + \nabla_x^2 f(x_t)(x_{\text{nt}} - x_t) &= 0 \\
\nabla_x^2 f(x_t)(x_{\text{nt}} - x_t) &= -\nabla_x f(x_t) \\
x_{\text{nt}} - x_t &= -\nabla_x^2 f(x_t)^{-1} \nabla_x f(x_t) \\
x_{\text{nt}} &= x_t - \nabla_x^2 f(x_t)^{-1} \nabla_x f(x_t)
\end{aligned}$$

assuming $\nabla_x^2 f(x_t)^T$ is positive definite (and hence, full rank). This, of course, is the standard Newton algorithm for unconstrained minimization.

While Newton's method converges quickly if given an initial point near the minimum, for points far from the minimum, Newton's method can sometimes diverge (as you may have discovered in problem 1 of Problem Set #1 if you picked an unfortunate initial point!). A simple fix for this behavior is to use a ***line-search*** procedure. Define the search direction d to be,

$$d := \nabla_x^2 f(x_t)^{-1} \nabla_x f(x_t).$$

A line-search procedure is an algorithm for finding an appropriate step size $\gamma \geq 0$ such that the iteration

$$x_{t+1} = x_t - \gamma \cdot d$$

will ensure that the function f decreases by a sufficient amount (relative to the size of the step taken) during each iteration.

One simple yet effective method for doing this is called a ***backtracking line search***. In this method, one initially sets γ to 1 and then iteratively reduces γ by a multiplicative factor β until $f(x_t + \gamma \cdot d)$ is sufficiently smaller than $f(x_t)$:

Backtracking line-search

- Choose $\alpha \in (0, 0.5)$, $\beta \in (0, 1)$.
- Set $\gamma \leftarrow 1$.
- While $f(x_t + \gamma \cdot d) > f(x_t) + \gamma \cdot \alpha \nabla_x f(x_t)^T d$, do $\gamma \leftarrow \beta\gamma$.
- Return γ.

Since the function f is known to decrease locally near x_t in the direction of d, such a step will be found, provided γ is small enough. For more details, see [1], pages 464-466.

In order to use Newton's method, one must be able to compute and invert the Hessian matrix $\nabla_x^2 f(x_t)$, or equivalently, compute the search direction d indirectly without forming the Hessian. For some problems, the number of primal variables x is sufficiently large that computing the Hessian can be very difficult. In many cases, this can be dealt with by clever use of linear algebra. In other cases, however, we can resort to other nonlinear minimization schemes, such as ***quasi-Newton*** methods, which initially behave like gradient descent but gradually construct approximations of the inverse Hessian based on the gradients observed throughout the course of the optimization.[5] Alternatively, ***nonlinear conjugate gradient*** schemes (which augment the standard conjugate gradient (CG) algorithm for solving linear least squares systems with a line-search) provide another generic blackbox tool for multivariable function minimization which is simple to implement, yet highly effective in practice.[6]

3.2 Inequality-constrained optimization

Using our tools for unconstrained optimization described in the previous section, we now tackle the (slightly) harder problem of constrained optimization. For simplicity, we consider convex optimization problems without equality constraints[7], i.e., problems of the form,

$$\begin{array}{ll} \underset{x}{\text{minimize}} & f(x) \\ \text{subject to} & g_i(x) \leq 0, \quad i = 1, \ldots, m. \end{array}$$

[5]For more information on Quasi-Newton methods, the standard reference is Jorge Nocedal and Stephen J. Wright's textbook, *Numerical Optimization*.

[6]For an excellent tutorial on the conjugate gradient method, see Jonathan Shewchuk's tutorial, available at: `http://www.cs.cmu.edu/~quake-papers/painless-conjugate-gradient.pdf`

[7]In practice, there are many of ways of dealing with equality constraints. Sometimes, we can eliminate equality constraints by either reparameterizing of the original primal problem, or converting to the dual problem. A more general strategy is to rely on equality-constrained variants of Newton's algorithms which ensure that the equality constraints are satisfied at every iteration of the optimization. For a more complete treatment of this topic, see [1], Chapter 10.

We will also assume knowledge of a feasible starting point x_0 which satisfies all of our constraints with strict inequality (as needed for Slater's condition to hold).[8]

Recall that in our discussion of the Lagrangian-based formulation of the primal problem,

$$\min_x \max_{\lambda:\lambda_i \geq 0} \quad \mathcal{L}(x, \lambda).$$

we stated that the inner maximization, $\max_{\lambda:\lambda_i \geq 0} \mathcal{L}(x, \lambda)$, was constructed in such a way that the infeasible region of f was "carved away", leaving only points in the feasible region as candidate minima. The same idea of using penalties to ensure that minimizers stay in the feasible region is the basis of ***barrier***-based optimization. Specifically, if $B(z)$ is the barrier function

$$B(z) = \begin{cases} 0 & z < 0 \\ \infty & z \geq 0, \end{cases}$$

then the primal problem is equivalent to

$$\min_x \quad f(x) + \sum_{i=1}^{m} B(g_i(x)). \tag{4}$$

When $g_i(x) < 0$, the objective of the problem is simply $f(x)$; infeasible points are "carved away" using the barrier function $B(z)$.

While conceptually correct, optimization using the straight barrier function $B(x)$ is numerically difficult. To ameliorate this, the ***log-barrier*** optimization algorithm approximates the solution to (4) by solving the unconstrained problem,

$$\underset{x}{\text{minimize}} \quad f(x) - \frac{1}{t} \sum_{i=1}^{m} \log(-g_i(x)).$$

for some fixed $t > 0$. Here, the function $-(1/t)\log(-z) \approx B(z)$, and the accuracy of the approximation increases as $t \to \infty$. Rather than using a large value of t in order to obtain a good approximation, however, the log-barrier algorithm works by solving a sequence of unconstrained optimization problems, increasing t each time, and using the solution of the previous unconstrained optimization problem as the initial point for the next unconstrained optimization. Furthermore, at each point in the algorithm, the primal solution points stay strictly in the interior of the feasible region:

[8] For more information on finding feasible starting points for barrier algorithms, see [1], pages 579-585. For inequality-problems where the primal problem is feasible but not strictly feasible, ***primal-dual interior point*** methods are applicable, also described in [1], pages 609-615.

Log-barrier optimization

- Choose $\mu > 1$, $t > 0$.
- $x \leftarrow x_0$.
- Repeat until convergence:
  (a) Compute $x' = \min_x f(x) - \frac{1}{t}\sum_{i=1}^{m} \log(-g_i(x))$ using x as the initial point.
  (b) $t \leftarrow \mu \cdot t$, $x \leftarrow x'$.

One might expect that as t increases, the difficulty of solving each unconstrained minimization problem also increases due to numerical issues or ill-conditioning of the optimization problem. Surprisingly, Nesterov and Nemirovski showed in 1994 that this is not the case for certain types of barrier functions, including the log-barrier; in particular, by using an appropriate barrier function, one obtains a general convex optimization algorithm which takes time polynomial in the dimensionality of the optimization variables and the desired accuracy!

4 Directions for further exploration

In many real-world tasks, 90% of the challenge involves figuring out how to write an optimization problem in a convex form. Once the correct form has been found, a number of pre-existing software packages for convex optimization have been well-tuned to handle different specific types of optimization problems. The following constitute a small sample of the available tools:

- commerical packages: CPLEX, MOSEK
- MATLAB-based: CVX, Optimization Toolbox (linprog, quadprog), SeDuMi
- libraries: CVXOPT (Python), GLPK (C), COIN-OR (C)
- SVMs: LIBSVM, SVM-light
- machine learning: Weka (Java)

In particular, we specifically point out CVX as an easy-to-use generic tool for solving convex optimization problems easily using MATLAB, and CVXOPT as a powerful Python-based library which runs independently of MATLAB.[9] If you're interested in looking at some of the other packages listed above, they are easy to find with a web search. In short, if you need a specific convex optimization algorithm, pre-existing software packages provide a rapid way to prototype your idea without having to deal with the numerical trickiness of implementing your own complete convex optimization routines.

[9] CVX is available at `http://www.stanford.edu/~boyd/cvx` and CVXOPT is available at `http://www.ee.ucla.edu/~vandenbe/cvxopt/`.

Also, if you find this material fascinating, make sure to check out Stephen Boyd's class, EE364: Convex Optimization I, which will be offered during the Winter Quarter. The textbook for the class (listed as [1] in the References) has a wealth of information about convex optimization and is available for browsing online.

References

[1] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge UP, 2004. Online: `http://www.stanford.edu/~boyd/cvxbook/`

Appendix: The soft-margin SVM

To see the primal/dual action in practice, we derive the dual of the soft-margin SVM primal presented in class, and corresponding KKT complementarity conditions. We have,

$$\begin{aligned} \underset{w,b,\xi}{\text{minimize}} \quad & \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{m}\xi_i \\ \text{subject to} \quad & y^{(i)}(w^T x^{(i)} + b) \geq 1 - \xi_i, \quad i = 1,\ldots,m, \\ & \xi_i \geq 0, \qquad\qquad\qquad\quad i = 1,\ldots,m. \end{aligned}$$

First, we put this into our standard form, with "≤ 0" inequality constraints and no equality constraints. That is,

$$\begin{aligned} \underset{w,b,\xi}{\text{minimize}} \quad & \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{m}\xi_i \\ \text{subject to} \quad & 1 - \xi_i - y^{(i)}(w^T x^{(i)} + b) \leq 0, \quad i = 1,\ldots,m, \\ & -\xi_i \leq 0, \qquad\qquad\qquad\quad i = 1,\ldots,m. \end{aligned}$$

Next, we form the generalized Lagrangian,[10]

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

which gives the primal and dual optimization problems:

$$\max_{\alpha,\beta:\alpha_i\geq 0,\beta_i\geq 0} \quad \theta_D(\alpha,\beta) \qquad \text{where } \theta_D(\alpha,\beta) := \min_{w,b,\xi} \quad \mathcal{L}(w,b,\xi,\alpha,\beta), \tag{SVM-D}$$

$$\min_{w,b,\xi} \quad \theta_P(w,b,\xi) \qquad \text{where } \theta_P(w,b,\xi) := \max_{\alpha,\beta:\alpha_i\geq 0,\beta_i\geq 0} \quad \mathcal{L}(w,b,\xi,\alpha,\beta). \tag{SVM-P}$$

To get the dual problem in the form shown in the lecture notes, however, we still have a little more work to do. In particular,

[10] Here, it is important to note that (w,b,ξ) collectively play the role of the x primal variables. Similarly, (α,β) collectively play the role of the λ dual variables used for inequality constraints. There are no "ν" dual variables here since there are no affine constraints in this problem.

1. **Eliminating the primal variables.** To eliminate the primal variables from the dual problem, we compute $\theta_D(\alpha, \beta)$ by noticing that

$$\theta_D(\alpha, \beta) = \min_{w,b,\xi} \quad \mathcal{L}(w, b, \xi, \alpha, \beta)$$

is an unconstrained optimization problem, where the objective function $\mathcal{L}(w, b, \xi, \alpha, \beta)$ is differentiable. Therefore, for any fixed (α, β), if $(\hat{w}, \hat{b}, \hat{\xi})$ minimize the Lagrangian, it must be the case that

$$\nabla_w \mathcal{L}(\hat{w}, \hat{b}, \hat{\xi}, \alpha, \beta) = \hat{w} - \sum_{i=1}^{m} \alpha_i y^{(i)} x^{(i)} = 0 \tag{5}$$

$$\frac{\partial}{\partial b} \mathcal{L}(\hat{w}, \hat{b}, \hat{\xi}, \alpha, \beta) = -\sum_{i=1}^{m} \alpha_i y^{(i)} = 0 \tag{6}$$

$$\frac{\partial}{\partial \xi_i} \mathcal{L}(\hat{w}, \hat{b}, \hat{\xi}, \alpha, \beta) = C - \alpha_i - \beta_i = 0. \tag{7}$$

Adding (6) and (7) to the constraints of our dual optimizaton problem, we obtain,

$$\begin{aligned}
\theta_D(\alpha, \beta) &= \mathcal{L}(\hat{w}, \hat{b}, \hat{\xi}) \\
&= \frac{1}{2}\|\hat{w}\|^2 + C\sum_{i=1}^{m} \hat{\xi}_i + \sum_{i=1}^{m} \alpha_i (1 - \hat{\xi}_i - y^{(i)}(\hat{w}^T x^{(i)} + \hat{b})) - \sum_{i=1}^{m} \beta_i \hat{\xi}_i \\
&= \frac{1}{2}\|\hat{w}\|^2 + C\sum_{i=1}^{m} \hat{\xi}_i + \sum_{i=1}^{m} \alpha_i (1 - \hat{\xi}_i - y^{(i)}(\hat{w}^T x^{(i)})) - \sum_{i=1}^{m} \beta_i \hat{\xi}_i \\
&= \frac{1}{2}\|\hat{w}\|^2 + \sum_{i=1}^{m} \alpha_i (1 - y^{(i)}(\hat{w}^T x^{(i)})).
\end{aligned}$$

where the first equality follows from the optimality of $(\hat{w}, \hat{b}, \hat{\xi})$ for fixed (α, β), the second equality uses the definition of the generalized Lagrangian, and the third and fourth equalities follow from (6) and (7), respectively. Finally, to use (5), observe that

$$\begin{aligned}
\frac{1}{2}\|\hat{w}\|^2 + \sum_{i=1}^{m} \alpha_i (1 - y^{(i)}(\hat{w}^T x^{(i)})) &= \sum_{i=1}^{m} \alpha_i + \frac{1}{2}\|\hat{w}\|^2 - \hat{w}^T \sum_{i=1}^{m} \alpha_i y^{(i)} x^{(i)} \\
&= \sum_{i=1}^{m} \alpha_i + \frac{1}{2}\|\hat{w}\|^2 - \|\hat{w}\|^2 \\
&= \sum_{i=1}^{m} \alpha_i - \frac{1}{2}\|\hat{w}\|^2 \\
&= \sum_{i=1}^{m} \alpha_i - \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m} \alpha_i \alpha_i y^{(i)} y^{(j)} \langle x^{(i)}, x^{(j)} \rangle.
\end{aligned}$$

Therefore, our dual problem (with no more primal variables) is simply

$$\begin{array}{ll} \underset{\alpha,\beta}{\text{maximize}} & \displaystyle\sum_{i=1}^{m} \alpha_i - \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m} \alpha_i \alpha_i y^{(i)} y^{(j)} \langle x^{(i)}, x^{(j)} \rangle \\ \text{subject to} & \alpha_i \geq 0, \qquad\qquad i = 1, \ldots, m, \\ & \beta_i \geq 0, \qquad\qquad i = 1, \ldots, m, \\ & \alpha_i + \beta_i = C, \qquad\qquad i = 1, \ldots, m, \\ & \displaystyle\sum_{i=1}^{m} \alpha_i y^{(i)} = 0. \end{array}$$

2. **KKT complementary.** KKT complementarity requires that for any primal optimal (w^*, b^*, ξ^*) and dual optimal (α^*, β^*),

$$\begin{aligned} \alpha_i^*(1 - \xi_i^* - y^{(i)}(w^{*T} x^{(i)} + b^*)) &= 0 \\ \beta_i^* \xi_i^* &= 0 \end{aligned}$$

for $i = 1, \ldots, m$. From the first condition, we see that if $\alpha_i > 0$, then in order for the product to be zero, then $1 - \xi_i^* - y^{(i)}(w^{*T} x^{(i)} + b^*) = 0$. It follows that

$$y^{(i)}(w^{*T} x^{(i)} + b^*) \leq 1$$

since $\xi^* \geq 0$ by primal feasibility. Similarly, if $\beta_i^* > 0$, then $\xi_i^* = 0$ to ensure complementarity. From the primal constraint, $y^{(i)}(w^T x^{(i)} + b) \geq 1 - \xi_i$, it follows that

$$y^{(i)}(w^{*T} x^{(i)} + b^*) \geq 1.$$

Finally, since $\beta_i^* > 0$ is equivalent to $\alpha_i^* < C$ (since $\alpha^* + \beta_i^* = C$), we can summarize the KKT conditions as follows:

$$\begin{aligned} \alpha_i^* = 0 \quad &\Rightarrow \quad y^{(i)}(w^{*T} x^{(i)} + b^*) \geq 1, \\ 0 < \alpha_i^* < C \quad &\Rightarrow \quad y^{(i)}(w^{*T} x^{(i)} + b^*) = 1, \\ \alpha_i^* = C \quad &\Rightarrow \quad y^{(i)}(w^{*T} x^{(i)} + b^*) \leq 1. \end{aligned}$$

3. **Simplification.** We can tidy up our dual problem slightly by observing that each pair of constraints of the form

$$\beta_i \geq 0 \qquad\qquad \alpha_i + \beta_i = C$$

is equivalent to the single constraint, $\alpha_i \leq C$; that is, if we solve the optimization problem

$$\begin{array}{ll} \underset{\alpha,\beta}{\text{maximize}} & \displaystyle\sum_{i=1}^{m} \alpha_i - \frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m} \alpha_i \alpha_i y^{(i)} y^{(j)} \langle x^{(i)}, x^{(j)} \rangle \\ \text{subject to} & 0 \leq \alpha_i \leq C, \qquad\qquad i = 1, \ldots, m, \\ & \displaystyle\sum_{i=1}^{m} \alpha_i y^{(i)} = 0. \end{array} \tag{8}$$

and subsequently set $\beta_i = C - \alpha_i$, then it follows that (α, β) will be optimal for the previous dual problem above. This last form, indeed, is the form of the soft-margin SVM dual given in the lecture notes.

Hidden Markov Models Fundamentals

Daniel Ramage
CS229 Section Notes

December 1, 2007

Abstract

How can we apply machine learning to data that is represented as a sequence of observations over time? For instance, we might be interested in discovering the sequence of words that someone spoke based on an audio recording of their speech. Or we might be interested in annotating a sequence of words with their part-of-speech tags. These notes provides a thorough mathematical introduction to the concept of Markov Models — a formalism for reasoning about states over time — and Hidden Markov Models — where we wish to recover a series of states from a series of observations. The final section includes some pointers to resources that present this material from other perspectives.

1 Markov Models

Given a set of states $S = \{s_1, s_2, ...s_{|S|}\}$ we can observe a series over time $\vec{z} \in S^T$. For example, we might have the states from a weather system $S = \{sun, cloud, rain\}$ with $|S| = 3$ and observe the weather over a few days $\{z_1 = s_{sun}, z_2 = s_{cloud}, z_3 = s_{cloud}, z_4 = s_{rain}, z_5 = s_{cloud}\}$ with $T = 5$.

The observed states of our weather example represent the output of a random process over time. Without some further assumptions, state s_j at time t could be a function of any number of variables, including all the states from times 1 to $t-1$ and possibly many others that we don't even model. However, we will make two MARKOV ASSUMPTIONS that will allow us to tractably reason about time series.

The LIMITED HORIZON ASSUMPTION is that the probability of being in a state at time t depends only on the state at time $t-1$. The intuition underlying this assumption is that the state at time t represents "enough" summary of the past to reasonably predict the future. Formally:

$$P(z_t|z_{t-1}, z_{t-2}, ..., z_1) = P(z_t|z_{t-1})$$

The STATIONARY PROCESS ASSUMPTION is that the conditional distribution over next state given current state does not change over time. Formally:

$$P(z_t|z_{t-1}) = P(z_2|z_1);\ t \in 2...T$$

As a convention, we will also assume that there is an initial state and initial observation $z_0 \equiv s_0$, where s_0 represents the initial probability distribution over states at time 0. This notational convenience allows us to encode our belief about the prior probability of seeing the first real state z_1 as $P(z_1|z_0)$. Note that $P(z_t|z_{t-1}, ..., z_1) = P(z_t|z_{t-1}, ..., z_1, z_0)$ because we've defined $z_0 = s_0$ for any state sequence. (Other presentations of HMMs sometimes represent these prior believes with a vector $\pi \in \mathbb{R}^{|S|}$.)

We parametrize these transitions by defining a state transition matrix $A \in \mathbb{R}^{(|S|+1)\times(|S|+1)}$. The value A_{ij} is the probability of transitioning from state i to state j at any time t. For our sun and rain example, we might have following transition matrix:

$$A = \begin{array}{c|cccc} & s_0 & s_{sun} & s_{cloud} & s_{rain} \\ \hline s_0 & 0 & .33 & .33 & .33 \\ s_{sun} & 0 & .8 & .1 & .1 \\ s_{cloud} & 0 & .2 & .6 & .2 \\ s_{rain} & 0 & .1 & .2 & .7 \end{array}$$

Note that these numbers (which I made up) represent the intuition that the weather is self-correlated: if it's sunny it will tend to stay sunny, cloudy will stay cloudy, etc. This pattern is common in many Markov models and can be observed as a strong diagonal in the transition matrix. Note that in this example, our initial state s_0 shows uniform probability of transitioning to each of the three states in our weather system.

1.1 Two questions of a Markov Model

Combining the Markov assumptions with our state transition parametrization A, we can answer two basic questions about a sequence of states in a Markov chain. What is the probability of a particular sequence of states $\vec{z}$? And how do we estimate the parameters of our model A such to maximize the likelihood of an observed sequence $\vec{z}$?

1.1.1 Probability of a state sequence

We can compute the probability of a particular series of states $\vec{z}$ by use of the chain rule of probability:

$$\begin{aligned} P(\vec{z}) &= P(z_t, z_{t-1}, ..., z_1; A) \\ &= P(z_t, z_{t-1}, ..., z_1, z_0; A) \\ &= P(z_t|z_{t-1}, z_{t-2}, ..., z_1; A)P(z_{t-1}|z_{t-2}, ..., z_1; A)...P(z_1|z_0; A) \\ &= P(z_t|z_{t-1}; A)P(z_{t-1}|z_{t-2}; A)...P(z_2|z_1; A)P(z_1|z_0; A) \end{aligned}$$

$$
\begin{aligned}
&= \prod_{t=1}^{T} P(z_t | z_{t-1}; A) \\
&= \prod_{t=1}^{T} A_{z_{t-1}\, z_t}
\end{aligned}
$$

In the second line we introduce z_0 into our joint probability, which is allowed by the definition of z_0 above. The third line is true of any joint distribution by the chain rule of probabilities or repeated application of Bayes rule. The fourth line follows from the Markov assumptions and the last line represents these terms as their elements in our transition matrix A.

Let's compute the probability of our example time sequence from earlier. We want $P(z_1 = s_{sun}, z_2 = s_{cloud}, z_3 = s_{rain}, z_4 = s_{rain}, z_5 = s_{cloud})$ which can be factored as $P(s_{sun}|s_0)P(s_{cloud}|s_{sun})P(s_{rain}|s_{cloud})P(s_{rain}|s_{rain})P(s_{cloud}|s_{rain}) = .33 \times .1 \times .2 \times .7 \times .2$.

1.1.2 Maximum likelihood parameter assignment

From a learning perspective, we could seek to find the parameters A that maximize the log-likelihood of sequence of observations $\vec{z}$. This corresponds to finding the likelihoods of transitioning from sunny to cloudy versus sunny to sunny, etc., that make a set of observations most likely. Let's define the log-likelihood a Markov model.

$$
\begin{aligned}
l(A) &= \log P(\vec{z}; A) \\
&= \log \prod_{t=1}^{T} A_{z_{t-1}\, z_t} \\
&= \sum_{t=1}^{T} \log A_{z_{t-1}\, z_t} \\
&= \sum_{i=1}^{|S|} \sum_{j=1}^{|S|} \sum_{t=1}^{T} 1\{z_{t-1} = s_i \wedge z_t = s_j\} \log A_{ij}
\end{aligned}
$$

In the last line, we use an indicator function whose value is one when the condition holds and zero otherwise to select the observed transition at each time step. When solving this optimization problem, it's important to ensure that solved parameters A still make a valid transition matrix. In particular, we need to enforce that the outgoing probability distribution from state i always sums to 1 and all elements of A are non-negative. We can solve this optimization problem using the method of Lagrange multipliers.

$$
\max_{A} \qquad l(A)
$$

$$
\begin{aligned}
s.t. \qquad & \sum_{j=1}^{|S|} A_{ij} = 1, \;\; i = 1..|S| \\
& A_{ij} \geq 0, \;\; i, j = 1..|S|
\end{aligned}
$$

This constrained optimization problem can be solved in closed form using the method of Lagrange multipliers. We'll introduce the equality constraint into the Lagrangian, but the inequality constraint can safely be ignored — the optimal solution will produce positive values for A_{ij} anyway. Therefore we construct the Lagrangian as:

$$
\mathcal{L}(A, \alpha) = \sum_{i=1}^{|S|} \sum_{j=1}^{|S|} \sum_{t=1}^{T} 1\{z_{t-1} = s_i \wedge z_t = s_j\} \log A_{ij} + \sum_{i=1}^{|S|} \alpha_i (1 - \sum_{j=1}^{|S|} A_{ij})
$$

Taking partial derivatives and setting them equal to zero we get:

$$
\begin{aligned}
\frac{\partial \mathcal{L}(A, \alpha)}{\partial A_{ij}} &= \frac{\partial}{\partial A_{ij}} (\sum_{t=1}^{T} 1\{z_{t-1} = s_i \wedge z_t = s_j\} \log A_{ij}) + \frac{\partial}{\partial A_{ij}} \alpha_i (1 - \sum_{j=1}^{|S|} A_{ij}) \\
&= \frac{1}{A_{ij}} \sum_{t=1}^{T} 1\{z_{t-1} = s_i \wedge z_t = s_j\} - \alpha_i \equiv 0 \\
&\Rightarrow \\
A_{ij} &= \frac{1}{\alpha_i} \sum_{t=1}^{T} 1\{z_{t-1} = s_i \wedge z_t = s_j\}
\end{aligned}
$$

Substituting back in and setting the partial with respect to α equal to zero:

$$
\begin{aligned}
\frac{\partial \mathcal{L}(A, \beta)}{\partial \alpha_i} &= 1 - \sum_{j=1}^{|S|} A_{ij} \\
&= 1 - \sum_{j=1}^{|S|} \frac{1}{\alpha_i} \sum_{t=1}^{T} 1\{z_{t-1} = s_i \wedge z_t = s_j\} \equiv 0 \\
&\Rightarrow \\
\alpha_i &= \sum_{j=1}^{|S|} \sum_{t=1}^{T} 1\{z_{t-1} = s_i \wedge z_t = s_j\} \\
&= \sum_{t=1}^{T} 1\{z_{t-1} = s_i\}
\end{aligned}
$$

Substituting in this value for α_i into the expression we derived for A_{ij} we obtain our final maximum likelihood parameter value for $\hat{A}_{ij}$.

$$\hat{A}_{ij} = \frac{\sum_{t=1}^{T} 1\{z_{t-1} = s_i \wedge z_t = s_j\}}{\sum_{t=1}^{T} 1\{z_{t-1} = s_i\}}$$

This formula encodes a simple intuition: the maximum likelihood probability of transitioning from state i to state j is just the number of times we transition from i to j divided by the total number of times we are in i. In other words, the maximum likelihood parameter corresponds to the fraction of the time when we were in state i that we transitioned to j.

2 Hidden Markov Models

Markov Models are a powerful abstraction for time series data, but fail to capture a very common scenario. How can we reason about a series of states if we cannot observe the states themselves, but rather only some probabilistic function of those states? This is the scenario for part-of-speech tagging where the words are observed but the parts-of-speech tags aren't, and for speech recognition where the sound sequence is observed but not the words that generated it. For a simple example, let's borrow the setup proposed by Jason Eisner in 2002 [1], "Ice Cream Climatology."

> The situation: You are a climatologist in the year 2799, studying the history of global warming. You can't find any records of Baltimore weather, but you do find my (Jason Eisner's) diary, in which I assiduously recorded how much ice cream I ate each day. *What can you figure out from this about the weather that summer?*

A Hidden Markov Model (HMM) can be used to explore this scenario. We don't get to observe the actual sequence of states (the weather on each day). Rather, we can only observe some outcome generated by each state (how many ice creams were eaten that day).

Formally, an HMM is a Markov model for which we have a series of *observed* outputs $x = \{x_1, x_2, ..., x_T\}$ drawn from an output alphabet $V = \{v_1, v_2, ..., v_{|V|}\}$, i.e. $x_t \in V$, $t = 1..T$. As in the previous section, we also posit the existence of series of states $z = \{z_1, z_2, ..., z_T\}$ drawn from a state alphabet $S = \{s_1, s_2, ...s_{|S|}\}$, $z_t \in S$, $t = 1..T$ but in this scenario the values of the states are *unobserved*. The transition between states i and j will again be represented by the corresponding value in our state transition matrix A_{ij}.

We also model the probability of generating an output observation as a function of our hidden state. To do so, we make the OUTPUT INDEPENDENCE ASSUMPTION and define $P(x_t = v_k | z_t = s_j) = P(x_t = v_k | x_1, ..., x_T, z_1, ..., z_T) = B_{jk}$. The matrix B encodes the probability of our hidden state generating output v_k given that the state at the corresponding time was s_j.

Returning to the weather example, imagine that you have logs of ice cream consumption over a four day period: $\vec{x} = \{x_1 = v_3, x_2 = v_2, x_3 = v_1, x_4 = v_2\}$

where our alphabet just encodes the number of ice creams consumed, i.e. $V = \{v_1 = 1\,ice\,cream, v_2 = 2\,ice\,creams, v_3 = 3\,ice\,creams\}$. What questions can an HMM let us answer?

2.1 Three questions of a Hidden Markov Model

There are three fundamental questions we might ask of an HMM. What is the probability of an observed sequence (how likely were we to see $3, 2, 1, 2$ ice creams consumed)? What is the most likely series of states to generate the observations (what was the weather for those four days)? And how can we learn values for the HMM's parameters A and B given some data?

2.2 Probability of an observed sequence: Forward procedure

In an HMM, we assume that our data was generated by the following process: posit the existence of a series of states $\vec{z}$ over the length of our time series. This state sequence is generated by a Markov model parametrized by a state transition matrix A. At each time step t, we select an output x_t as a function of the state z_t. Therefore, to get the probability of a sequence of observations, we need to add up the likelihood of the data $\vec{x}$ given every possible series of states.

$$\begin{aligned} P(\vec{x}; A, B) &= \sum_{\vec{z}} P(\vec{x}, \vec{z}; A, B) \\ &= \sum_{\vec{z}} P(\vec{x}|\vec{z}; A, B) P(\vec{z}; A, B) \end{aligned}$$

The formulas above are true for any probability distribution. However, the HMM assumptions allow us to simplify the expression further:

$$\begin{aligned} P(\vec{x}; A, B) &= \sum_{\vec{z}} P(\vec{x}|\vec{z}; A, B) P(\vec{z}; A, B) \\ &= \sum_{\vec{z}} (\prod_{t=1}^{T} P(x_t|z_t; B)) (\prod_{t=1}^{T} P(z_t|z_{t-1}; A)) \\ &= \sum_{\vec{z}} (\prod_{t=1}^{T} B_{z_t\, x_t}) (\prod_{t=1}^{T} A_{z_{t-1}\, z_t}) \end{aligned}$$

The good news is that this is a simple expression in terms of our parameters. The derivation follows the HMM assumptions: the output independence assumption, Markov assumption, and stationary process assumption are all used to derive the second line. The bad news is that the sum is over every possible assignment to $\vec{z}$. Because z_t can take one of $|S|$ possible values at each time step, evaluating this sum directly will require $O(|S|^T)$ operations.

Algorithm 1 Forward Procedure for computing $\alpha_i(t)$

1. Base case: $\alpha_i(0) = A_{0\,i}$, $i = 1..|S|$
2. Recursion: $\alpha_j(t) = \sum_{i=1}^{|S|} \alpha_i(t-1) A_{ij} B_{j\,x_t}$, $j = 1..|S|$, $t = 1..T$

Fortunately, a faster means of computing $P(\vec{x}; A, B)$ is possible via a dynamic programming algorithm called the FORWARD PROCEDURE. First, let's define a quantity $\alpha_i(t) = P(x_1, x_2, ..., x_t, z_t = s_i; A, B)$. $\alpha_i(t)$ represents the total probability of all the observations up through time t (by any state assignment) and that we are in state s_i at time t. If we had such a quantity, the probability of our full set of observations $P(\vec{x})$ could be represented as:

$$\begin{aligned} P(\vec{x}; A, B) &= P(x_1, x_2, ..., x_T; A, B) \\ &= \sum_{i=1}^{|S|} P(x_1, x_2, ..., x_T, z_T = s_i; A, B) \\ &= \sum_{i=1}^{|S|} \alpha_i(T) \end{aligned}$$

Algorithm 2.2 presents an efficient way to compute $\alpha_i(t)$. At each time step we must do only $O(|S|)$ operations, resulting in a final algorithm complexity of $O(|S| \cdot T)$ to compute the total probability of an observed state sequence $P(\vec{x}; A, B)$.

A similar algorithm known as the BACKWARD PROCEDURE can be used to compute an analogous probability $\beta_i(t) = P(x_T, x_{T-1}, .., x_{t+1}, z_t = s_i; A, B)$.

2.3 Maximum Likelihood State Assignment: The Viterbi Algorithm

One of the most common queries of a Hidden Markov Model is to ask what was the most likely series of states $\vec{z} \in S^T$ given an observed series of outputs $\vec{x} \in V^T$. Formally, we seek:

$$\arg\max_{\vec{z}} P(\vec{z}|\vec{x}; A, B) = \arg\max_{\vec{z}} \frac{P(\vec{x}, \vec{z}; A, B)}{\sum_{\vec{z}} P(\vec{x}, \vec{z}; A, B)} = \arg\max_{\vec{z}} P(\vec{x}, \vec{z}; A, B)$$

The first simplification follows from Bayes rule and the second from the observation that the denominator does not directly depend on $\vec{z}$. Naively, we might try every possible assignment to $\vec{z}$ and take the one with the highest joint probability assigned by our model. However, this would require $O(|S|^T)$ operations just to enumerate the set of possible assignments. At this point, you might think a dynamic programming solution like the Forward Algorithm might save the day, and you'd be right. Notice that if you replaced the $\arg\max_{\vec{z}}$ with $\sum_{\vec{z}}$, our current task is exactly analogous to the expression which motivated the forward procedure.

Algorithm 2 Naive application of EM to HMMs

Repeat until convergence {
(E-Step) For every possible labeling $\vec{z} \in S^T$, set

$$Q(\vec{z}) \quad := \quad p(\vec{z}|\vec{x}; A, B)$$

(M-Step) Set

$$\begin{aligned} A, B \quad &:= \quad \arg\max_{A,B} \sum_{\vec{z}} Q(\vec{z}) \log \frac{P(\vec{x}, \vec{z}; A, B)}{Q(\vec{z})} \\ &s.t. \quad \sum_{j=1}^{|S|} A_{ij} = 1,\ i = 1..|S|;\ A_{ij} \geq 0,\ \ i, j = 1..|S| \\ &\qquad \sum_{k=1}^{|V|} B_{ik} = 1,\ i = 1..|S|;\ B_{ik} \geq 0,\ \ i = 1..|S|, k = 1..|V| \end{aligned}$$

}

The VITERBI ALGORITHM is just like the forward procedure except that instead of tracking the total probability of generating the observations seen so far, we need only track the *maximum* probability and record its corresponding state sequence.

2.4 Parameter Learning: EM for HMMs

The final question to ask of an HMM is: given a set of observations, what are the values of the state transition probabilities A and the output emission probabilities B that make the data most likely? For example, solving for the maximum likelihood parameters based on a speech recognition dataset will allow us to effectively train the HMM before asking for the maximum likelihood state assignment of a candidate speech signal.

In this section, we present a derivation of the Expectation Maximization algorithm for Hidden Markov Models. This proof follows from the general formulation of EM presented in the CS229 lecture notes. Algorithm 2.4 shows the basic EM algorithm. Notice that the optimization problem in the M-Step is now constrained such that A and B contain valid probabilities. Like the maximum likelihood solution we found for (non-Hidden) Markov models, we'll be able to solve this optimization problem with Lagrange multipliers. Notice also that the E-Step and M-Step both require enumerating all $|S|^T$ possible labellings of $\vec{z}$. We'll make use of the Forward and Backward algorithms mentioned earlier to compute a set of sufficient statistics for our E-Step and M-Step tractably.

First, let's rewrite the objective function using our Markov assumptions.

$$
\begin{aligned}
A, B &= \arg\max_{A,B} \sum_{\vec{z}} Q(\vec{z}) \log \frac{P(\vec{x}, \vec{z}; A, B)}{Q(\vec{z})} \\
&= \arg\max_{A,B} \sum_{\vec{z}} Q(\vec{z}) \log P(\vec{x}, \vec{z}; A, B) \\
&= \arg\max_{A,B} \sum_{\vec{z}} Q(\vec{z}) \log(\prod_{t=1}^{T} P(x_t|z_t; B)) \, (\prod_{t=1}^{T} P(z_t|z_{t-1}; A)) \\
&= \arg\max_{A,B} \sum_{\vec{z}} Q(\vec{z}) \sum_{t=1}^{T} \log B_{z_t \, x_t} + \log A_{z_{t-1} \, z_t} \\
&= \arg\max_{A,B} \sum_{\vec{z}} Q(\vec{z}) \sum_{i=1}^{|S|} \sum_{j=1}^{|S|} \sum_{k=1}^{|V|} \sum_{t=1}^{T} 1\{z_t = s_j \wedge x_t = v_k\} \log B_{jk} + 1\{z_{t-1} = s_i \wedge z_t = s_j\} \log A_{ij}
\end{aligned}
$$

In the first line we split the log division into a subtraction and note that the denominator's term does not depend on the parameters A, B. The Markov assumptions are applied in line 3. Line 5 uses indicator functions to index A and B by state.

Just as for the maximum likelihood parameters for a visible Markov model, it is safe to ignore the inequality constraints because the solution form naturally results in only positive solutions. Constructing the Lagrangian:

$$
\begin{aligned}
\mathcal{L}(A, B, \delta, \epsilon) &= \sum_{\vec{z}} Q(\vec{z}) \sum_{i=1}^{|S|} \sum_{j=1}^{|S|} \sum_{k=1}^{|V|} \sum_{t=1}^{T} 1\{z_t = s_j \wedge x_t = v_k\} \log B_{jk} + 1\{z_{t-1} = s_i \wedge z_t = s_j\} \log A_{ij} \\
&\quad + \sum_{j=1}^{|S|} \epsilon_j (1 - \sum_{k=1}^{|V|} B_{jk}) + \sum_{i=1}^{|S|} \delta_i (1 - \sum_{j=1}^{|S|} A_{ij})
\end{aligned}
$$

Taking partial derivatives and setting them equal to zero:

$$
\begin{aligned}
\frac{\partial \mathcal{L}(A, B, \delta, \epsilon)}{\partial A_{ij}} &= \sum_{\vec{z}} Q(\vec{z}) \frac{1}{A_{ij}} \sum_{t=1}^{T} 1\{z_{t-1} = s_i \wedge z_t = s_j\} - \delta_i \equiv 0 \\
A_{ij} &= \frac{1}{\delta_i} \sum_{\vec{z}} Q(\vec{z}) \sum_{t=1}^{T} 1\{z_{t-1} = s_i \wedge z_t = s_j\}
\end{aligned}
$$

$$
\begin{aligned}
\frac{\partial \mathcal{L}(A, B, \delta, \epsilon)}{\partial B_{jk}} &= \sum_{\vec{z}} Q(\vec{z}) \frac{1}{B_{jk}} \sum_{t=1}^{T} 1\{z_t = s_j \wedge x_t = v_k\} - \epsilon_j \equiv 0 \\
B_{jk} &= \frac{1}{\epsilon_j} \sum_{\vec{z}} Q(\vec{z}) \sum_{t=1}^{T} 1\{z_t = s_j \wedge x_t = v_k\}
\end{aligned}
$$

Taking partial derivatives with respect to the Lagrange multipliers and substituting our values of A_{ij} and B_{jk} above:

$$\begin{aligned}
\frac{\partial \mathcal{L}(A,B,\delta,\epsilon)}{\partial \delta_i} &= 1 - \sum_{j=1}^{|S|} A_{ij} \\
&= 1 - \sum_{j=1}^{|S|} \frac{1}{\delta_i} \sum_{\vec{z}} Q(\vec{z}) \sum_{t=1}^{T} 1\{z_{t-1} = s_i \wedge z_t = s_j\} \equiv 0 \\
\delta_i &= \sum_{j=1}^{|S|} \sum_{\vec{z}} Q(\vec{z}) \sum_{t=1}^{T} 1\{z_{t-1} = s_i \wedge z_t = s_j\} \\
&= \sum_{\vec{z}} Q(\vec{z}) \sum_{t=1}^{T} 1\{z_{t-1} = s_i\}
\end{aligned}$$

$$\begin{aligned}
\frac{\partial \mathcal{L}(A,B,\delta,\epsilon)}{\partial \epsilon_j} &= 1 - \sum_{k=1}^{|V|} B_{jk} \\
&= 1 - \sum_{k=1}^{|V|} \frac{1}{\epsilon_j} \sum_{\vec{z}} Q(\vec{z}) \sum_{t=1}^{T} 1\{z_t = s_j \wedge x_t = v_k\} \equiv 0 \\
\epsilon_j &= \sum_{k=1}^{|V|} \sum_{\vec{z}} Q(\vec{z}) \sum_{t=1}^{T} 1\{z_t = s_j \wedge x_t = v_k\} \\
&= \sum_{\vec{z}} Q(\vec{z}) \sum_{t=1}^{T} 1\{z_t = s_j\}
\end{aligned}$$

Substituting back into our expressions above, we find that parameters $\hat{A}$ and $\hat{B}$ that maximize our predicted counts with respect to the dataset are:

$$\begin{aligned}
\hat{A}_{ij} &= \frac{\sum_{\vec{z}} Q(\vec{z}) \sum_{t=1}^{T} 1\{z_{t-1} = s_i \wedge z_t = s_j\}}{\sum_{\vec{z}} Q(\vec{z}) \sum_{t=1}^{T} 1\{z_{t-1} = s_i\}} \\
\hat{B}_{jk} &= \frac{\sum_{\vec{z}} Q(\vec{z}) \sum_{t=1}^{T} 1\{z_t = s_j \wedge x_t = v_k\}}{\sum_{\vec{z}} Q(\vec{z}) \sum_{t=1}^{T} 1\{z_t = s_j\}}
\end{aligned}$$

Unfortunately, each of these sums is over all possible labellings $\vec{z} \in S^T$. But recall that $Q(\vec{z})$ was defined in the E-step as $P(\vec{z}|\vec{x}; A, B)$ for parameters A and B at the last time step. Let's consider how to represent first the numerator of $\hat{A}_{ij}$ in terms of our forward and backward probabilities, $\alpha_i(t)$ and $\beta_j(t)$.

$$\sum_{\vec{z}} Q(\vec{z}) \sum_{t=1}^{T} 1\{z_{t-1} = s_i \wedge z_t = s_j\}$$

$$
\begin{aligned}
&= \sum_{t=1}^{T} \sum_{\vec{z}} 1\{z_{t-1} = s_i \wedge z_t = s_j\} Q(\vec{z}) \\
&= \sum_{t=1}^{T} \sum_{\vec{z}} 1\{z_{t-1} = s_i \wedge z_t = s_j\} P(\vec{z}|\vec{x}; A, B) \\
&= \frac{1}{P(\vec{x}; A, B)} \sum_{t=1}^{T} \sum_{\vec{z}} 1\{z_{t-1} = s_i \wedge z_t = s_j\} P(\vec{z}, \vec{x}; A, B) \\
&= \frac{1}{P(\vec{x}; A, B)} \sum_{t=1}^{T} \alpha_i(t) A_{ij} B_{j\, x_t} \beta_j(t+1)
\end{aligned}
$$

In the first two steps we rearrange terms and substitute in for our definition of Q. Then we use Bayes rule in deriving line four, followed by the definitions of α, β, A, and B, in line five. Similarly, the denominator can be represented by summing out over j the value of the numerator.

$$
\begin{aligned}
& \sum_{\vec{z}} Q(\vec{z}) \sum_{t=1}^{T} 1\{z_{t-1} = s_i\} \\
&= \sum_{j=1}^{|S|} \sum_{\vec{z}} Q(\vec{z}) \sum_{t=1}^{T} 1\{z_{t-1} = s_i \wedge z_t = s_j\} \\
&= \frac{1}{P(\vec{x}; A, B)} \sum_{j=1}^{|S|} \sum_{t=1}^{T} \alpha_i(t) A_{ij} B_{j\, x_t} \beta_j(t+1)
\end{aligned}
$$

Combining these expressions, we can fully characterize our maximum likelihood state transitions $\hat{A}_{ij}$ without needing to enumerate all possible labellings as:

$$
\hat{A}_{ij} = \frac{\sum_{t=1}^{T} \alpha_i(t) A_{ij} B_{j\, x_t} \beta_j(t+1)}{\sum_{j=1}^{|S|} \sum_{t=1}^{T} \alpha_i(t) A_{ij} B_{j\, x_t} \beta_j(t+1)}
$$

Similarly, we can represent the numerator for $\hat{B}_{jk}$ as:

$$
\begin{aligned}
& \sum_{\vec{z}} Q(\vec{z}) \sum_{t=1}^{T} 1\{z_t = s_j \wedge x_t = v_k\} \\
&= \frac{1}{P(\vec{x}; A, B)} \sum_{t=1}^{T} \sum_{\vec{z}} 1\{z_t = s_j \wedge x_t = v_k\} P(\vec{z}, \vec{x}; A, B) \\
&= \frac{1}{P(\vec{x}; A, B)} \sum_{i=1}^{|S|} \sum_{t=1}^{T} \sum_{\vec{z}} 1\{z_{t-1} = s_i \wedge z_t = s_j \wedge x_t = v_k\} P(\vec{z}, \vec{x}; A, B)
\end{aligned}
$$

Algorithm 3 Forward-Backward algorithm for HMM parameter learning

Initialization: Set A and B as random valid probability matrices
where $A_{i0} = 0$ and $B_{0k} = 0$ for $i = 1..|S|$ and $k = 1..|V|$.
Repeat until convergence {
(E-Step) Run the Forward and Backward algorithms to compute α_i and β_i for $i = 1..|S|$. Then set:

$$\gamma_t(i,j) \quad := \quad \alpha_i(t) A_{ij} B_{j\,x_t} \beta_j(t+1)$$

(M-Step) Re-estimate the maximum likelihood parameters as:

$$\begin{aligned} A_{ij} &:= \frac{\sum_{t=1}^{T} \gamma_t(i,j)}{\sum_{j=1}^{|S|} \sum_{t=1}^{T} \gamma_t(i,j)} \\ B_{jk} &:= \frac{\sum_{i=1}^{|S|} \sum_{t=1}^{T} 1\{x_t = v_k\}\, \gamma_t(i,j)}{\sum_{i=1}^{|S|} \sum_{t=1}^{T} \gamma_t(i,j)} \end{aligned}$$

}

$$= \frac{1}{P(\vec{x}; A, B)} \sum_{i=1}^{|S|} \sum_{t=1}^{T} 1\{x_t = v_k\} \alpha_i(t) A_{ij} B_{j\,x_t} \beta_j(t+1)$$

And the denominator of $\hat{B}_{jk}$ as:

$$\begin{aligned} & \sum_{\vec{z}} Q(\vec{z}) \sum_{t=1}^{T} 1\{z_t = s_j\} \\ = & \frac{1}{P(\vec{x}; A, B)} \sum_{i=1}^{|S|} \sum_{t=1}^{T} \sum_{\vec{z}} 1\{z_{t-1} = s_i \wedge z_t = s_j\} P(\vec{z}, \vec{x}; A, B) \\ = & \frac{1}{P(\vec{x}; A, B)} \sum_{i=1}^{|S|} \sum_{t=1}^{T} \alpha_i(t) A_{ij} B_{j\,x_t} \beta_j(t+1) \end{aligned}$$

Combining these expressions, we have the following form for our maximum likelihood emission probabilities as:

$$\hat{B}_{jk} \quad = \quad \frac{\sum_{i=1}^{|S|} \sum_{t=1}^{T} 1\{x_t = v_k\} \alpha_i(t) A_{ij} B_{j\,x_t} \beta_j(t+1)}{\sum_{i=1}^{|S|} \sum_{t=1}^{T} \alpha_i(t) A_{ij} B_{j\,x_t} \beta_j(t+1)}$$

Algorithm 2.4 shows a variant of the Forward-Backward Algorithm, or the Baum-Welch Algorithm for parameter learning in HMMs. In the

E-Step, rather than explicitly evaluating $Q(\vec{z})$ for all $\vec{z} \in S^T$, we compute a sufficient statistics $\gamma_t(i,j) = \alpha_i(t)A_{ij}B_{j\,x_t}\beta_j(t+1)$ that is proportional to the probability of transitioning between sate s_i and s_j at time t given all of our observations $\vec{x}$. The derived expressions for A_{ij} and B_{jk} are intuitively appealing. A_{ij} is computed as the expected number of transitions from s_i to s_j divided by the expected number of appearances of s_i. Similarly, B_{jk} is computed as the expected number of emissions of v_k from s_j divided by the expected number of appearances of s_j.

Like many applications of EM, parameter learning for HMMs is a non-convex problem with many local maxima. EM will converge to a maximum based on its initial parameters, so multiple runs might be in order. Also, it is often important to smooth the probability distributions represented by A and B so that no transition or emission is assigned 0 probability.

2.5 Further reading

There are many good sources for learning about Hidden Markov Models. For applications in NLP, I recommend consulting Jurafsky & Martin's draft second edition of *Speech and Language Processing*[1] or Manning & Schütze's *Foundations of Statistical Natural Language Processing*. Also, Eisner's HMM-in-a-spreadsheet [1] is a light-weight interactive way to play with an HMM that requires only a spreadsheet application.

References

[1] Jason Eisner. An interactive spreadsheet for teaching the forward-backward algorithm. In Dragomir Radev and Chris Brew, editors, *Proceedings of the ACL Workshop on Effective Tools and Methodologies for Teaching NLP and CL*, pages 10–18, 2002.

[1] http://www.cs.colorado.edu/~martin/slp2.html

Gaussian processes

Chuong B. Do

December 1, 2007

Many of the classical machine learning algorithms that we talked about during the first half of this course fit the following pattern: given a training set of i.i.d. examples sampled from some unknown distribution,

1. solve a convex optimization problem in order to identify the single "best fit" model for the data, and

2. use this estimated model to make "best guess" predictions for future test input points.

In these notes, we will talk about a different flavor of learning algorithms, known as **Bayesian methods**. Unlike classical learning algorithm, Bayesian algorithms do not attempt to identify "best-fit" models of the data (or similarly, make "best guess" predictions for new test inputs). Instead, they compute a posterior distribution over models (or similarly, compute posterior predictive distributions for new test inputs). These distributions provide a useful way to quantify our uncertainty in model estimates, and to exploit our knowledge of this uncertainty in order to make more robust predictions on new test points.

We focus on **regression** problems, where the goal is to learn a mapping from some input space $\mathcal{X} = \mathbf{R}^n$ of n-dimensional vectors to an output space $\mathcal{Y} = \mathbf{R}$ of real-valued targets. In particular, we will talk about a kernel-based fully Bayesian regression algorithm, known as Gaussian process regression. The material covered in these notes draws heavily on many different topics that we discussed previously in class (namely, the probabilistic interpretation of linear regression[1], Bayesian methods[2], kernels[3], and properties of multivariate Gaussians[4]).

The organization of these notes is as follows. In Section 1, we provide a brief review of multivariate Gaussian distributions and their properties. In Section 2, we briefly review Bayesian methods in the context of probabilistic linear regression. The central ideas underlying Gaussian processes are presented in Section 3, and we derive the full Gaussian process regression model in Section 4.

[1] See course lecture notes on "Supervised Learning, Discriminative Algorithms."
[2] See course lecture notes on "Regularization and Model Selection."
[3] See course lecture notes on "Support Vector Machines."
[4] See course lecture notes on "Factor Analysis."

1 Multivariate Gaussians

A vector-valued random variable $x \in \mathbf{R}^n$ is said to have a **multivariate normal (or Gaussian) distribution** with mean $\mu \in \mathbf{R}^n$ and covariance matrix $\Sigma \in \mathbf{S}^n_{++}$ if

$$p(x; \mu, \Sigma) = \frac{1}{(2\pi)^{n/2}|\Sigma|} \exp\left(-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)\right). \tag{1}$$

We write this as $x \sim \mathcal{N}(\mu, \Sigma)$. Here, recall from the section notes on linear algebra that $\mathbf{S}^n_{++}$ refers to the space of symmetric positive definite $n \times n$ matrices.[5]

Generally speaking, Gaussian random variables are extremely useful in machine learning and statistics for two main reasons. First, they are extremely common when modeling "noise" in statistical algorithms. Quite often, noise can be considered to be the accumulation of a large number of small independent random perturbations affecting the measurement process; by the Central Limit Theorem, summations of independent random variables will tend to "look Gaussian." Second, Gaussian random variables are convenient for many analytical manipulations, because many of the integrals involving Gaussian distributions that arise in practice have simple closed form solutions. In the remainder of this section, we will review a number of useful properties of multivariate Gaussians.

Consider a random vector $x \in \mathbf{R}^n$ with $x \sim \mathcal{N}(\mu, \Sigma)$. Suppose also that the variables in x have been partitioned into two sets $x_A = [x_1 \ \cdots \ x_r]^T \in \mathbf{R}^r$ and $x_B = [x_{r+1} \ \cdots \ x_n]^T \in \mathbf{R}^{n-r}$ (and similarly for μ and Σ), such that

$$x = \begin{bmatrix} x_A \\ x_B \end{bmatrix} \qquad \mu = \begin{bmatrix} \mu_A \\ \mu_B \end{bmatrix} \qquad \Sigma = \begin{bmatrix} \Sigma_{AA} & \Sigma_{AB} \\ \Sigma_{BA} & \Sigma_{BB} \end{bmatrix}.$$

Here, $\Sigma_{AB} = \Sigma_{BA}^T$ since $\Sigma = E[(x-\mu)(x-\mu)^T] = \Sigma^T$. The following properties hold:

1. **Normalization.** The density function normalizes, i.e.,

$$\int_x p(x; \mu, \Sigma)dx = 1.$$

   This property, though seemingly trivial at first glance, turns out to be immensely useful for evaluating all sorts of integrals, even ones which appear to have no relation to probability distributions at all (see Appendix A.1)!

2. **Marginalization.** The marginal densities,

$$p(x_A) = \int_{x_B} p(x_A, x_B; \mu, \Sigma)dx_B$$
$$p(x_B) = \int_{x_A} p(x_A, x_B; \mu, \Sigma)dx_A$$

[5] There are actually cases in which we would want to deal with multivariate Gaussian distributions where Σ is positive semidefinite but not positive definite (i.e., Σ is not full rank). In such cases, Σ^{-1} does not exist, so the definition of the Gaussian density given in (1) does not apply. For instance, see the course lecture notes on "Factor Analysis."

are Gaussian:

$$x_A \sim \mathcal{N}(\mu_A, \Sigma_{AA})$$
$$x_B \sim \mathcal{N}(\mu_B, \Sigma_{BB}).$$

3. **Conditioning.** The conditional densities

$$p(x_A \mid x_B) = \frac{p(x_A, x_B; \mu, \Sigma)}{\int_{x_A} p(x_A, x_B; \mu, \Sigma) dx_A}$$
$$p(x_B \mid x_A) = \frac{p(x_A, x_B; \mu, \Sigma)}{\int_{x_B} p(x_A, x_B; \mu, \Sigma) dx_B}$$

are also Gaussian:

$$x_A \mid x_B \sim \mathcal{N}\big(\mu_A + \Sigma_{AB}\Sigma_{BB}^{-1}(x_B - \mu_B), \Sigma_{AA} - \Sigma_{AB}\Sigma_{BB}^{-1}\Sigma_{BA}\big)$$
$$x_B \mid x_A \sim \mathcal{N}\big(\mu_B + \Sigma_{BA}\Sigma_{AA}^{-1}(x_A - \mu_A), \Sigma_{BB} - \Sigma_{BA}\Sigma_{AA}^{-1}\Sigma_{AB}\big).$$

A proof of this property is given in Appendix A.2.

4. **Summation.** The sum of independent Gaussian random variables (with the same dimensionality), $y \sim \mathcal{N}(\mu, \Sigma)$ and $z \sim \mathcal{N}(\mu', \Sigma')$, is also Gaussian:

$$y + z \sim \mathcal{N}(\mu + \mu', \Sigma + \Sigma').$$

2 Bayesian linear regression

Let $S = \{(x^{(i)}, y^{(i)})\}_{i=1}^m$ be a training set of i.i.d. examples from some unknown distribution. The standard probabilistic interpretation of linear regression states that

$$y^{(i)} = \theta^T x^{(i)} + \varepsilon^{(i)}, \qquad i = 1, \ldots, m$$

where the $\varepsilon^{(i)}$ are i.i.d. "noise" variables with independent $\mathcal{N}(0, \sigma^2)$ distributions. It follows that $y^{(i)} - \theta^T x^{(i)} \sim \mathcal{N}(0, \sigma^2)$, or equivalently,

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

For notational convenience, we define

$$X = \begin{bmatrix} — & (x^{(1)})^T & — \\ — & (x^{(2)})^T & — \\ & \vdots & \\ — & (x^{(m)})^T & — \end{bmatrix} \in \mathbf{R}^{m \times n} \qquad \vec{y} = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{bmatrix} \in \mathbf{R}^m \qquad \vec{\varepsilon} = \begin{bmatrix} \varepsilon^{(1)} \\ \varepsilon^{(2)} \\ \vdots \\ \varepsilon^{(m)} \end{bmatrix} \in \mathbf{R}^m.$$

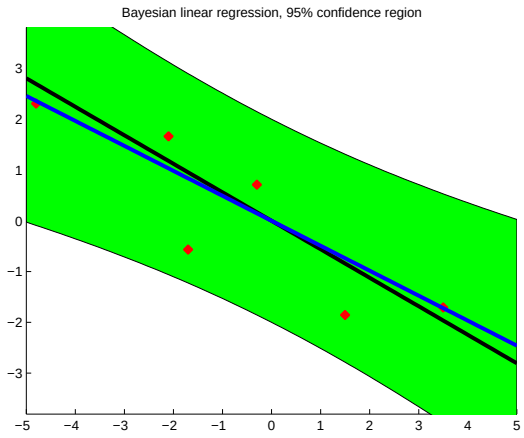


Figure 1: Bayesian linear regression for a one-dimensional linear regression problem, $y^{(i)} = \theta x^{(i)} + \epsilon^{(i)}$, with $\epsilon^{(i)} \sim \mathcal{N}(0, 1)$ i.i.d. noise. The green region denotes the 95% confidence region for predictions of the model. Note that the (vertical) width of the green region is largest at the ends but narrowest in the middle. This region reflects the uncertain in the estimates for the parameter θ. In contrast, a classical linear regression model would display a confidence region of constant width, reflecting only the $\mathcal{N}(0, \sigma^2)$ noise in the outputs.

In Bayesian linear regression, we assume that a **prior distribution** over parameters is also given; a typical choice, for instance, is $\theta \sim \mathcal{N}(0, \tau^2 I)$. Using Bayes's rule, we obtain the **parameter posterior**,

$$p(\theta \mid S) = \frac{p(\theta)p(S \mid \theta)}{\int_{\theta'} p(\theta')p(S \mid \theta')d\theta'} = \frac{p(\theta)\prod_{i=1}^{m} p(y^{(i)} \mid x^{(i)}, \theta)}{\int_{\theta'} p(\theta')\prod_{i=1}^{m} p(y^{(i)} \mid x^{(i)}, \theta')d\theta'}. \tag{2}$$

Assuming the same noise model on testing points as on our training points, the "output" of Bayesian linear regression on a new test point x_* is not just a single guess "y_*", but rather an entire probability distribution over possible outputs, known as the **posterior predictive distribution**:

$$p(y_* \mid x_*, S) = \int_{\theta} p(y_* \mid x_*, \theta)p(\theta \mid S)d\theta. \tag{3}$$

For many types of models, the integrals in (2) and (3) are difficult to compute, and hence, we often resort to approximations, such as MAP estimation (see course lecture notes on "Regularization and Model Selection").

In the case of Bayesian linear regression, however, the integrals actually are tractable! In particular, for Bayesian linear regression, one can show (after much work!) that

$$\theta \mid S \sim \mathcal{N}\left(\frac{1}{\sigma^2}A^{-1}X^T\vec{y}, A^{-1}\right)$$

$$y_* \mid x_*, S \sim \mathcal{N}\left(\frac{1}{\sigma^2}x_*^T A^{-1}X^T\vec{y}, x_*^T A^{-1}x_* + \sigma^2\right)$$

where $A = \frac{1}{\sigma^2} X^T X + \frac{1}{\tau^2} I$. The derivation of these formulas is somewhat involved.[6] Nonetheless, from these equations, we get at least a flavor of what Bayesian methods are all about: the posterior distribution over the test output y_* for a test input x_* is a Gaussian distribution—this distribution reflects the uncertainty in our predictions $y_* = \theta^T x_* + \varepsilon_*$ arising from both the randomness in ε_* and the uncertainty in our choice of parameters θ. In contrast, classical probabilistic linear regression models estimate parameters θ directly from the training data but provide no estimate of how reliable these learned parameters may be (see Figure 1).

3 Gaussian processes

As described in Section 1, multivariate Gaussian distributions are useful for modeling finite collections of real-valued variables because of their nice analytical properties. **Gaussian processes** are the extension of multivariate Gaussians to infinite-sized collections of real-valued variables. In particular, this extension will allow us to think of Gaussian processes as distributions not just over random vectors but in fact distributions over **random functions**.[7]

3.1 Probability distributions over functions with finite domains

To understand how one might paramterize probability distributions over functions, consider the following simple example. Let $\mathcal{X} = \{x_1, \ldots, x_m\}$ be any finite set of elements. Now, consider the set $\mathcal{H}$ of all possible functions mapping from $\mathcal{X}$ to $\mathbf{R}$. For instance, one example of a function $h_0(\cdot) \in \mathcal{H}$ is given by

$$h_0(x_1) = 5, \quad h_0(x_2) = 2.3, \quad h_0(x_2) = -7, \quad \ldots, \quad h_0(x_{m-1}) = -\pi, \quad h_0(x_m) = 8.$$

Since the domain of any $h(\cdot) \in \mathcal{H}$ has only m elements, we can always represent $h(\cdot)$ compactly as an m-dimensional vector, $\vec{h} = \begin{bmatrix} h(x_1) & h(x_2) & \cdots & h(x_m) \end{bmatrix}^T$. In order to specify a probability distribution over functions $h(\cdot) \in \mathcal{H}$, we must associate some "probability density" with each function in $\mathcal{H}$. One natural way to do this is to exploit the one-to-one correspondence between functions $h(\cdot) \in \mathcal{H}$ and their vector representations, $\vec{h}$. In particular, if we specify that $\vec{h} \sim \mathcal{N}(\vec{\mu}, \sigma^2 I)$, then this in turn implies a probability distribution over functions $h(\cdot)$, whose probability density function is given by

$$p(h) = \prod_{i=1}^{m} \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{1}{2\sigma^2} (h(x_i) - \mu_i)^2 \right).$$

[6] For the complete derivation, see, for instance, [1]. Alternatively, read the Appendices, which gives a number of arguments based on the "completion-of-squares" trick, and derive this formula yourself!

[7] Let $\mathcal{H}$ be a class of functions mapping from $\mathcal{X} \to \mathcal{Y}$. A random function $h(\cdot)$ from $\mathcal{H}$ is a function which is randomly drawn from $\mathcal{H}$, according to some probability distribution over $\mathcal{H}$. One potential source of confusion is that you may be tempted to think of random functions as functions whose outputs are in some way stochastic; this is not the case. Instead, a random function $h(\cdot)$, once selected from $\mathcal{H}$ probabilistically, implies a deterministic mapping from inputs in $\mathcal{X}$ to outputs in $\mathcal{Y}$.

In the example above, we showed that probability distributions over functions with finite domains can be represented using a finite-dimensional multivariate Gaussian distribution over function outputs $h(x_1), \ldots, h(x_m)$ at a finite number of input points $x_1, \ldots, x_m$. How can we specify probability distributions over functions when the domain size may be infinite? For this, we turn to a fancier type of probability distribution known as a Gaussian process.

3.2 Probability distributions over functions with infinite domains

A stochastic process is a collection of random variables, $\{h(x) : x \in \mathcal{X}\}$, indexed by elements from some set $\mathcal{X}$, known as the index set.[8] A **Gaussian process** is a stochastic process such that any finite subcollection of random variables has a multivariate Gaussian distribution.

In particular, a collection of random variables $\{h(x) : x \in \mathcal{X}\}$ is said to be drawn from a Gaussian process with **mean function** $m(\cdot)$ and **covariance function** $k(\cdot, \cdot)$ if for any finite set of elements $x_1, \ldots, x_m \in \mathcal{X}$, the associated finite set of random variables $h(x_1), \ldots, h(x_m)$ have distribution,

$$\begin{bmatrix} h(x_1) \\ \vdots \\ h(x_m) \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} m(x_1) \\ \vdots \\ m(x_m) \end{bmatrix}, \begin{bmatrix} k(x_1, x_1) & \cdots & k(x_1, x_m) \\ \vdots & \ddots & \vdots \\ k(x_m, x_1) & \cdots & k(x_m, x_m) \end{bmatrix} \right).$$

We denote this using the notation,

$$h(\cdot) \sim \mathcal{GP}(m(\cdot), k(\cdot, \cdot)).$$

Observe that the mean function and covariance function are aptly named since the above properties imply that

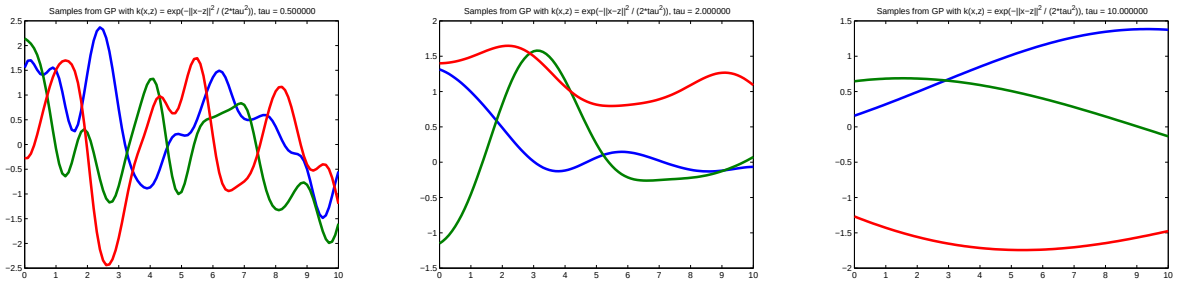
$$\begin{aligned} m(x) &= E[x] \\ k(x, x') &= E[(x - m(x))(x' - m(x')). \end{aligned}$$

for any $x, x' \in \mathcal{X}$.

Intuitively, one can think of a function $h(\cdot)$ drawn from a Gaussian process prior as an extremely high-dimensional vector drawn from an extremely high-dimensional multivariate Gaussian. Here, each dimension of the Gaussian corresponds to an element x from the index set $\mathcal{X}$, and the corresponding component of the random vector represents the value of $h(x)$. Using the marginalization property for multivariate Gaussians, we can obtain the marginal multivariate Gaussian density corresponding to any finite subcollection of variables.

What sort of functions $m(\cdot)$ and $k(\cdot, \cdot)$ give rise to valid Gaussian processes? In general, any real-valued function $m(\cdot)$ is acceptable, but for $k(\cdot, \cdot)$, it must be the case that for any

[8] Often, when $\mathcal{X} = \mathbf{R}$, one can interpret the indices $x \in \mathcal{X}$ as representing times, and hence the variables $h(x)$ represent the temporal evolution of some random quantity over time. In the models that are used for Gaussian process regression, however, the index set is taken to be the input space of our regression problem.



(a) (b) (c)

Figure 2: Samples from a zero-mean Gaussian process prior with $k_{SE}(\cdot, \cdot)$ covariance function, using (a) $\tau = 0.5$, (b) $\tau = 2$, and (c) $\tau = 10$. Note that as the bandwidth parameter τ increases, then points which are farther away will have higher correlations than before, and hence the sampled functions tend to be smoother overall.

set of elements $x_1, \ldots, x_m \in \mathcal{X}$, the resulting matrix

$$K = \begin{bmatrix} k(x_1, x_1) & \cdots & k(x_1, x_m) \\ \vdots & \ddots & \vdots \\ k(x_m, x_1) & \cdots & k(x_m, x_m) \end{bmatrix}$$

is a valid covariance matrix corresponding to some multivariate Gaussian distribution. A standard result in probability theory states that this is true provided that K is positive semidefinite. Sound familiar?

The positive semidefiniteness requirement for covariance matrices computed based on arbitrary input points is, in fact, identical to Mercer's condition for kernels! A function $k(\cdot, \cdot)$ is a valid kernel provided the resulting kernel matrix K defined as above is always positive semidefinite for any set of input points $x_1, \ldots, x_m \in \mathcal{X}$. Gaussian processes, therefore, are kernel-based probability distributions in the sense that any valid kernel function can be used as a covariance function!

3.3 The squared exponential kernel

In order to get an intuition for how Gaussian processes work, consider a simple zero-mean Gaussian process,

$$h(\cdot) \sim \mathcal{GP}(0, k(\cdot, \cdot)).$$

defined for functions $h : \mathcal{X} \to \mathbf{R}$ where we take $\mathcal{X} = \mathbf{R}$. Here, we choose the kernel function $k(\cdot, \cdot)$ to be the **squared exponential**[9] kernel function, defined as

$$k_{SE}(x, x') = \exp\left(-\frac{1}{2\tau^2}||x - x'||^2\right)$$

[9]In the context of SVMs, we called this the Gaussian kernel; to avoid confusion with "Gaussian" processes, we refer to this kernel here as the squared exponential kernel, even though the two are formally identical.

for some $\tau > 0$. What do random functions sampled from this Gaussian process look like?

In our example, since we use a zero-mean Gaussian process, we would expect that for the function values from our Gaussian process will tend to be distributed around zero. Furthermore, for any pair of elements $x, x' \in \mathcal{X}$.

- $h(x)$ and $h(x')$ will tend to have high covariance x and x' are "nearby" in the input space (i.e., $||x - x'|| = |x - x'| \approx 0$, so $\exp(-\frac{1}{2\tau^2}||x - x'||^2) \approx 1$).
- $h(x)$ and $h(x')$ will tend to have low covariance when x and x' are "far apart" (i.e., $||x - x'|| \gg 0$, so $\exp(-\frac{1}{2\tau^2}||x - x'||^2) \approx 0$).

More simply stated, functions drawn from a zero-mean Gaussian process prior with the squared exponential kernel will tend to be "locally smooth" with high probability; i.e., nearby function values are highly correlated, and the correlation drops off as a function of distance in the input space (see Figure 2).

4 Gaussian process regression

As discussed in the last section, Gaussian processes provide a method for modelling probability distributions over functions. Here, we discuss how probability distributions over functions can be used in the framework of Bayesian regression.

4.1 The Gaussian process regression model

Let $S = \{(x^{(i)}, y^{(i)})\}_{i=1}^m$ be a training set of i.i.d. examples from some unknown distribution. In the Gaussian process regression model,

$$y^{(i)} = h(x^{(i)}) + \varepsilon^{(i)}, \qquad i = 1, \ldots, m$$

where the $\varepsilon^{(i)}$ are i.i.d. "noise" variables with independent $\mathcal{N}(0, \sigma^2)$ distributions. Like in Bayesian linear regression, we also assume a **prior distribution** over functions $h(\cdot)$; in particular, we assume a zero-mean Gaussian process prior,

$$h(\cdot) \sim \mathcal{GP}(0, k(\cdot, \cdot))$$

for some valid covariance function $k(\cdot, \cdot)$.

Now, let $T = \{(x_*^{(i)}, y_*^{(i)})\}_{i=1}^{m_*}$ be a set of i.i.d. testing points drawn from the same unknown

distribution as S.[10] For notational convenience, we define

$$X = \begin{bmatrix} — & (x^{(1)})^T & — \\ — & (x^{(2)})^T & — \\ & \vdots & \\ — & (x^{(m)})^T & — \end{bmatrix} \in \mathbf{R}^{m \times n} \qquad \vec{h} = \begin{bmatrix} h(x^{(1)}) \\ h(x^{(2)}) \\ \vdots \\ h(x^{(m)}) \end{bmatrix}, \qquad \vec{\varepsilon} = \begin{bmatrix} \varepsilon^{(1)} \\ \varepsilon^{(2)} \\ \vdots \\ \varepsilon^{(m)} \end{bmatrix}, \quad \vec{y} = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{bmatrix} \in \mathbf{R}^m,$$

$$X_* = \begin{bmatrix} — & (x_*^{(1)})^T & — \\ — & (x_*^{(2)})^T & — \\ & \vdots & \\ — & (x_*^{(m_*)})^T & — \end{bmatrix} \in \mathbf{R}^{m_* \times n} \quad \vec{h}_* = \begin{bmatrix} h(x_*^{(1)}) \\ h(x_*^{(2)}) \\ \vdots \\ h(x_*^{(m_*)}) \end{bmatrix}, \quad \vec{\varepsilon}_* = \begin{bmatrix} \varepsilon_*^{(1)} \\ \varepsilon_*^{(2)} \\ \vdots \\ \varepsilon_*^{(m)} \end{bmatrix}, \quad \vec{y}_* = \begin{bmatrix} y_*^{(1)} \\ y_*^{(2)} \\ \vdots \\ y_*^{(m_*)} \end{bmatrix} \in \mathbf{R}^{m_*}.$$

Given the training data S, the prior $p(h)$, and the testing inputs X_*, how can we compute the posterior predictive distribution over the testing outputs $\vec{y}_*$? For Bayesian linear regression in Section 2, we used Bayes's rule in order to compute the paramter posterior, which we then used to compute posterior predictive distribution $p(y_* \mid x_*, S)$ for a new test point x_*. For Gaussian process regression, however, it turns out that an even simpler solution exists!

4.2 Prediction

Recall that for any function $h(\cdot)$ drawn from our zero-mean Gaussian process prior with covariance function $k(\cdot, \cdot)$, the marginal distribution over any set of input points belonging to $\mathcal{X}$ must have a joint multivariate Gaussian distribution. In particular, this must hold for the training and test points, so we have

$$\begin{bmatrix} \vec{h} \\ \vec{h_*} \end{bmatrix} \Bigg| X, X_* \sim \mathcal{N}\left(\vec{0}, \begin{bmatrix} K(X,X) & K(X,X_*) \\ K(X_*,X) & K(X_*,X_*) \end{bmatrix}\right),$$

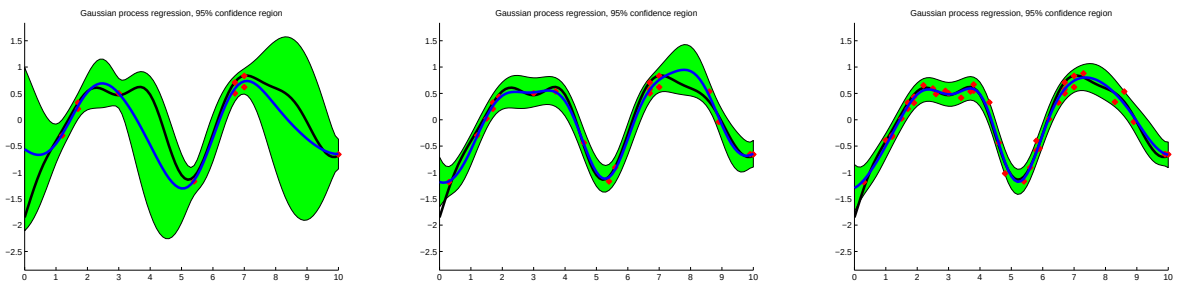
where

$$\begin{aligned} \vec{h} \in \mathbf{R}^m \text{ such that } \vec{h} &= \begin{bmatrix} h(x^{(1)}) & \cdots & h(x^{(m)}) \end{bmatrix}^T \\ \vec{h}_* \in \mathbf{R}^{m_*} \text{ such that } \vec{h_*} &= \begin{bmatrix} h(x_*^{(1)}) & \cdots & h(x_*^{(m)}) \end{bmatrix}^T \\ K(X,X) \in \mathbf{R}^{m \times m} \text{ such that } (K(X,X))_{ij} &= k(x^{(i)}, x^{(j)}) \\ K(X,X_*) \in \mathbf{R}^{m \times m_*} \text{ such that } (K(X,X_*))_{ij} &= k(x^{(i)}, x_*^{(j)}) \\ K(X_*,X) \in \mathbf{R}^{m_* \times m} \text{ such that } (K(X_*,X))_{ij} &= k(x_*^{(i)}, x^{(j)}) \\ K(X_*,X_*) \in \mathbf{R}^{m_* \times m_*} \text{ such that } (K(X_*,X_*))_{ij} &= k(x_*^{(i)}, x_*^{(j)}). \end{aligned}$$

From our i.i.d. noise assumption, we have that

$$\begin{bmatrix} \vec{\varepsilon} \\ \vec{\varepsilon}_* \end{bmatrix} \sim \mathcal{N}\left(\vec{0}, \begin{bmatrix} \sigma^2 I & \vec{0} \\ \vec{0}^T & \sigma^2 I \end{bmatrix}\right).$$

[10] We assume also that T are S are mutually independent.



(a) (b) (c)

Figure 3: Gaussian process regression using a zero-mean Gaussian process prior with $k_{SE}(\cdot, \cdot)$ covariance function (where $\tau = 0.1$), with noise level $\sigma = 1$, and (a) $m = 10$, (b) $m = 20$, and (c) $m = 40$ training examples. The blue line denotes the mean of the posterior predictive distribution, and the green shaded region denotes the 95% confidence region based on the model's variance estimates. As the number of training examples increases, the size of the confidence region shrinks to reflect the diminishing uncertainty in the model estimates. Note also that in panel (a), the 95% confidence region shrinks near training points but is much larger far away from training points, as one would expect.

The sums of independent Gaussian random variables is also Gaussian, so

$$\begin{bmatrix} \vec{y} \\ \vec{y}_* \end{bmatrix} \Bigg| X, X_* = \begin{bmatrix} \vec{h} \\ \vec{h}_* \end{bmatrix} + \begin{bmatrix} \vec{\varepsilon} \\ \vec{\varepsilon}_* \end{bmatrix} \sim \mathcal{N}\left(\vec{0}, \begin{bmatrix} K(X,X) + \sigma^2 I & K(X, X_*) \\ K(X_*, X) & K(X_*, X_*) + \sigma^2 I \end{bmatrix} \right).$$

Now, using the rules for conditioning Gaussians, it follows that

$$\vec{y_*} \mid \vec{y}, X, X_* \sim \mathcal{N}(\mu^*, \Sigma^*)$$

where

$$\begin{aligned} \mu^* &= K(X_*, X)(K(X,X) + \sigma^2 I)^{-1} \vec{y} \\ \Sigma^* &= K(X_*, X_*) + \sigma^2 I - K(X_*, X)(K(X,X) + \sigma^2 I)^{-1} K(X, X_*). \end{aligned}$$

And that's it! Remarkably, performing prediction in a Gaussian process regression model is very simple, despite the fact that Gaussian processes in themselves are fairly complicated![11]

5 Summary

We close our discussion of our Gaussian processes by pointing out some reasons why Gaussian processes are an attractive model for use in regression problems and in some cases may be preferable to alternative models (such as linear and locally-weighted linear regression):

[11] Interestingly, it turns out that Bayesian linear regression, when "kernelized" in the proper way, turns out to be exactly equivalent to Gaussian process regression! But the derivation of the posterior predictive distribution is far more complicated for Bayesian linear regression, and the effort needed to kernelize the algorithm is even greater. The Gaussian process perspective is certainly much easier!

1. As Bayesian methods, Gaussian process models allow one to quantify uncertainty in predictions resulting not just from intrinsic noise in the problem but also the errors in the parameter estimation procedure. Furthermore, many methods for model selection and hyperparameter selection in Bayesian methods are immediately applicable to Gaussian processes (though we did not address any of these advanced topics here).

2. Like locally-weighted linear regression, Gaussian process regression is non-parametric and hence can model essentially arbitrary functions of the input points.

3. Gaussian process regression models provide a natural way to introduce kernels into a regression modeling framework. By careful choice of kernels, Gaussian process regression models can sometimes take advantage of structure in the data (though, we also did not examine this issue here).

4. Gaussian process regression models, though perhaps somewhat tricky to understand conceptually, nonetheless lead to simple and straightforward linear algebra implementations.

References

[1] Carl E. Rasmussen and Christopher K. I. Williams. *Gaussian Processes for Machine Learning.* MIT Press, 2006. Online: `http://www.gaussianprocess.org/gpml/`

Appendix A.1

In this example, we show how the normalization property for multivariate Gaussians can be used to compute rather intimidating multidimensional integrals without performing any real calculus! Suppose you wanted to compute the following multidimensional integral,

$$I(A,b,c) = \int_x \exp\left(-\frac{1}{2}x^T Ax - x^T b - c\right)dx,$$

for some $A \in \mathbf{S}^m_{++}$, $b \in \mathbf{R}^m$, and $c \in \mathbf{R}$. Although one could conceivably perform the multidimensional integration directly (good luck!), a much simpler line of reasoning is based on a mathematical trick known as "completion-of-squares." In particular,

$$\begin{aligned} I(A,b,c) &= \exp(-c) \cdot \int_x \exp\left(-\frac{1}{2}x^T Ax - x^T AA^{-1}b\right)dx \\ &= \exp(-c) \cdot \int_x \exp\left(-\frac{1}{2}(x-A^{-1}b)^T A(x-A^{-1}b) - b^T A^{-1}b\right)dx \\ &= \exp\left(-c - b^T A^{-1}b\right) \cdot \int_x \exp\left(-\frac{1}{2}(x-A^{-1}b)^T A(x-A^{-1}b)\right)dx. \end{aligned}$$

Defining $\mu = A^{-1}b$ and $\Sigma = A^{-1}$, it follows that $I(A,b,c)$ is equal to

$$\frac{(2\pi)^{m/2}|\Sigma|}{\exp(c + b^T A^{-1}b)} \cdot \left[\frac{1}{(2\pi)^{m/2}|\Sigma|}\int_x \exp\left(-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)\right)dx\right].$$

However, the term in brackets is identical in form to the integral of a multivariate Gaussian! Since we know that a Gaussian density normalizes, it follows that the term in brackets is equal to 1. Therefore,

$$I(A,b,c) = \frac{(2\pi)^{m/2}|A^{-1}|}{\exp(c + b^T A^{-1}b)}.$$

Appendix A.2

We derive the form of the distribution of x_A given x_B; the other result follows immediately by symmetry. Note that

$$\begin{aligned} p(x_A \mid x_B) &= \frac{1}{\int_{x_A} p(x_A, x_B; \mu, \Sigma)dx_A} \cdot \left[\frac{1}{(2\pi)^{m/2}|\Sigma|}\exp\left(-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)\right)\right] \\ &= \frac{1}{Z_1}\exp\left\{-\frac{1}{2}\left(\begin{bmatrix}x_A\\x_B\end{bmatrix} - \begin{bmatrix}\mu_A\\\mu_B\end{bmatrix}\right)^T \begin{bmatrix}V_{AA} & V_{AB}\\V_{BA} & V_{BB}\end{bmatrix}\left(\begin{bmatrix}x_A\\x_B\end{bmatrix} - \begin{bmatrix}\mu_A\\\mu_B\end{bmatrix}\right)\right\} \end{aligned}$$

where Z_1 is a proportionality constant which does not depend on x_A, and

$$\Sigma^{-1} = V = \begin{bmatrix}V_{AA} & V_{AB}\\V_{BA} & V_{BB}\end{bmatrix}.$$

To simplify this expression, observe that

$$\begin{aligned}
&\left(\begin{bmatrix} x_A \\ x_B \end{bmatrix} - \begin{bmatrix} \mu_A \\ \mu_B \end{bmatrix}\right)^T \begin{bmatrix} V_{AA} & V_{AB} \\ V_{BA} & V_{BB} \end{bmatrix} \left(\begin{bmatrix} x_A \\ x_B \end{bmatrix} - \begin{bmatrix} \mu_A \\ \mu_B \end{bmatrix}\right) \\
&\quad = (x_A - \mu_A)^T V_{AA}(x_A - \mu_A) + (x_A - \mu_A)^T V_{AB}(x_B - \mu_B) \\
&\qquad + (x_B - \mu_B)^T V_{BA}(x_A - \mu_A) + (x_B - \mu_B)^T V_{BB}(x_B - \mu_B).
\end{aligned}$$

Retaining only terms dependent on x_A (and using the fact that $V_{AB} = V_{BA}^T$), we have

$$p(x_A \mid x_B) = \frac{1}{Z_2} \exp\left(-\frac{1}{2}\left[x_A^T V_{AA} x_A - 2x_A^T V_{AA}\mu_A + 2x_A^T V_{AB}(x_B - \mu_B)\right]\right)$$

where Z_2 is a new proportionality constant which again does not depend on x_A. Finally, using the "completion-of-squares" argument (see Appendix A.1), we have

$$p(x_A \mid x_B) = \frac{1}{Z_3} \exp\left(-\frac{1}{2}(x_A - \mu')^T V_{AA}(x_A - \mu')\right)$$

where Z_3 is again a new proportionality constant not depending on x_A, and where $mu' = \mu_A - V_{AA}^{-1} V_{AB}(x_B - \mu_B)$. This last statement shows that the distribution of x_A, conditioned on x_B, again has the form of a multivariate Gaussian. In fact, from the normalization property, it follows immediately that

$$x_A \mid x_B \sim \mathcal{N}(\mu_A - V_{AA}^{-1} V_{AB}(x_B - \mu_B), V_{AA}^{-1}).$$

To complete the proof, we simply note that

$$\begin{bmatrix} V_{AA} & V_{AB} \\ V_{BA} & V_{BB} \end{bmatrix} = \begin{bmatrix} (\Sigma_{AA} - \Sigma_{AB}\Sigma_{BB}^{-1}\Sigma_{BA})^{-1} & -(\Sigma_{AA} - \Sigma_{AB}\Sigma_{BB}^{-1}\Sigma_{BA})^{-1}\Sigma_{AB}\Sigma_{BB}^{-1} \\ -\Sigma_{BB}^{-1}\Sigma_{BA}(\Sigma_{AA} - \Sigma_{AB}\Sigma_{BB}^{-1}\Sigma_{BA})^{-1} & (\Sigma_{BB} - \Sigma_{BA}\Sigma_{AA}^{-1}\Sigma_{AB})^{-1} \end{bmatrix}$$

follows from standard formulas for the inverse of a partitioned matrix. Substituting the relevant blocks into the previous expression gives the desired result. □