CS229 Lecture notes

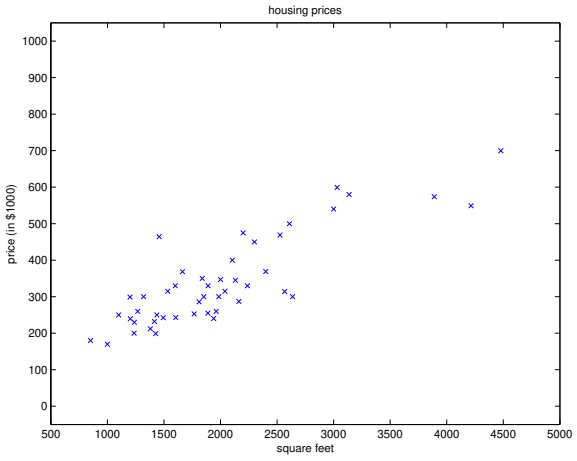
Andrew Ng

Supervised learning

Let's 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
⋮	⋮

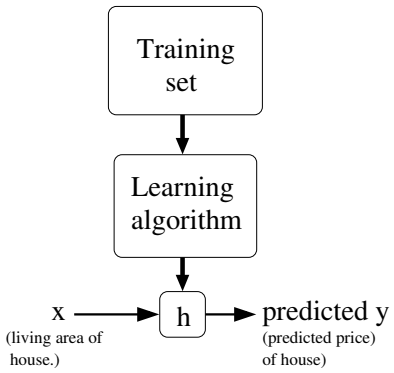
We can plot this data:



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, let's 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 let's 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, let's 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, let's 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, let's 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, let's 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. Let's 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:

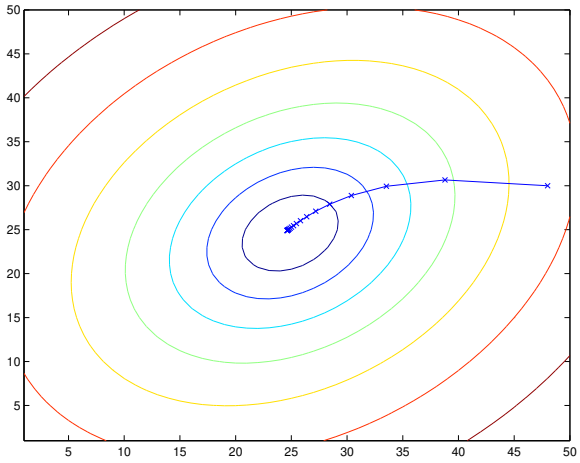
Repeat until convergence {

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

}

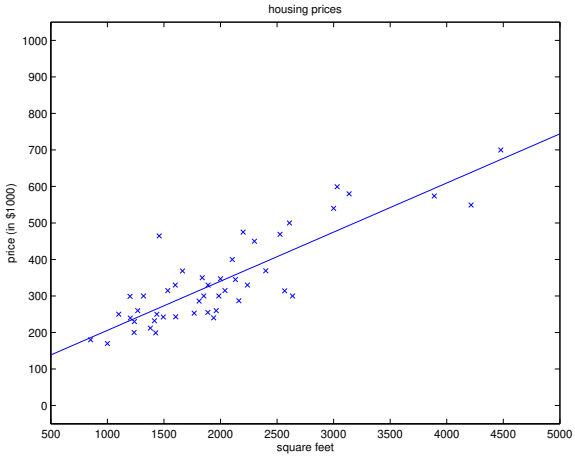
The reader can easily verify that the quantity in the summation in the update rule above is just $\partial J(\theta)/\partial \theta_j$ (for the original definition of J). So, this is simply gradient descent on the original cost function J. This method looks at every example in the entire training set 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, which 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, {

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

}

}

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. Let's 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, let's 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.

Given 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, let's 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

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

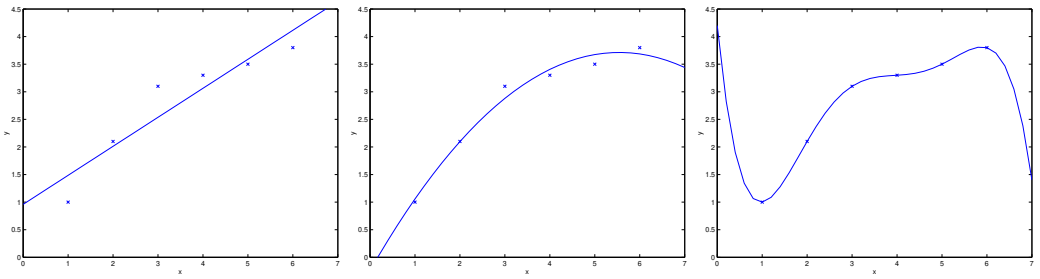
which we recognize to be $J(\theta)$, our original least-squares cost function.

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

Let's 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 $y \in \{0, 1\}$.

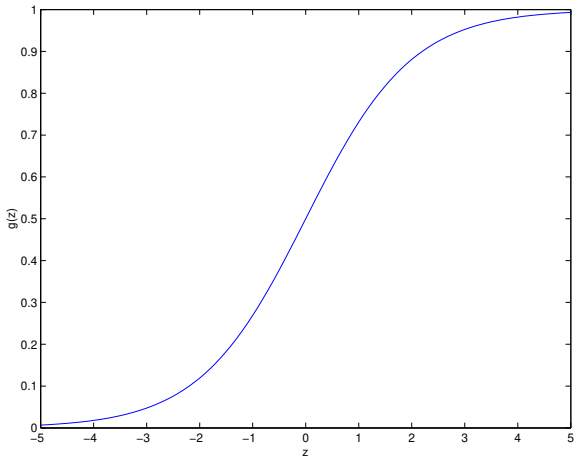
To fix this, let's change the form for our hypotheses $h_\theta(x)$. We will choose

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

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, let's 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 as 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, let's 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.) Let's 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 $\ell(\theta)$

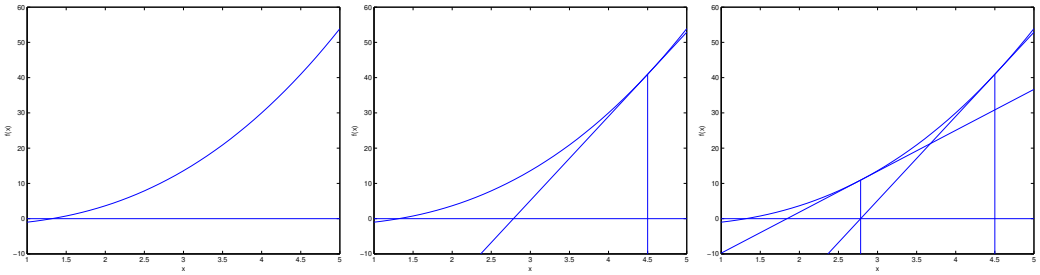
Returning to logistic regression with $g(z)$ being the sigmoid function, let's now talk about a different algorithm for maximizing $\ell(\theta)$.

To get us started, let's 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, $y|x;\theta \sim \text{Bernoulli}(\phi)$, 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 vary 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.

Let's 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, let's 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

Let's 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.

Let's 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, let's 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 an 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. Let's 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:

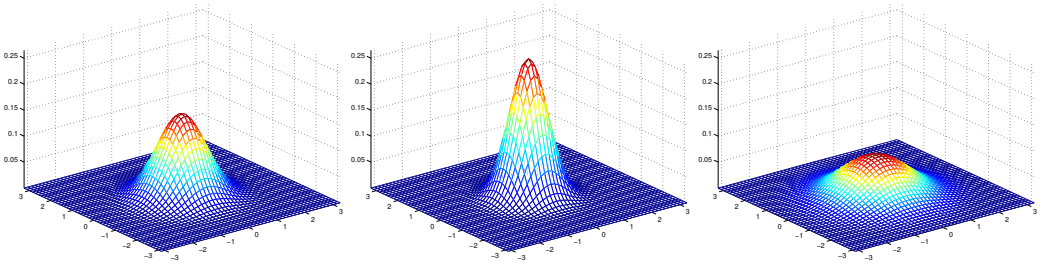
$$\mathrm{E}[X] = \int_x x\, p(x; \mu, \Sigma)dx = \mu$$

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

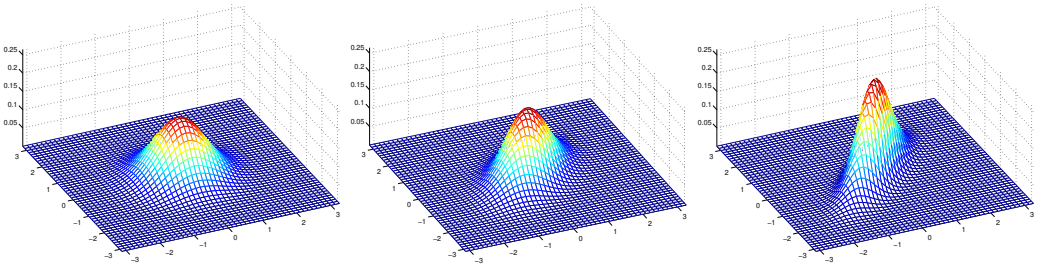
$$\mathrm{Cov}(X) = \Sigma.$$

Here're some examples of what the density of a Gaussian distribution looks 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."

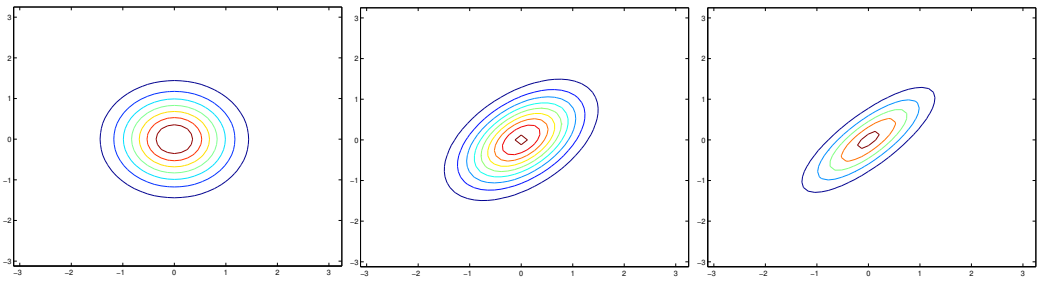
Let's look at some more examples.



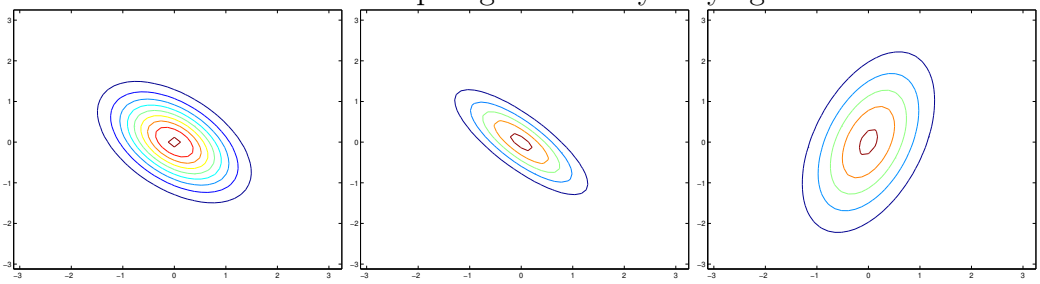
The figures above show Gaussians with mean 0, and with covariance matrices respectively

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

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:

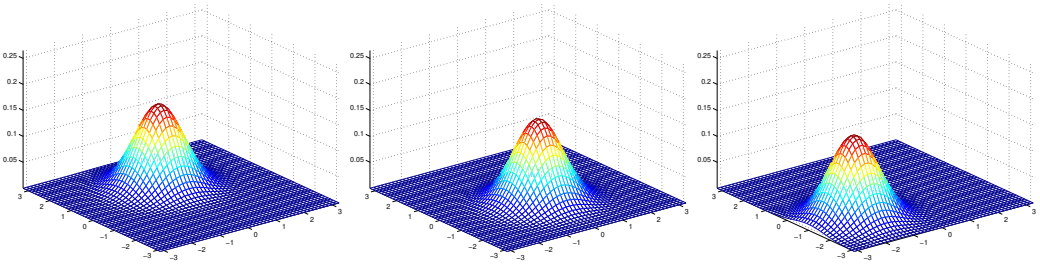


The plots above used, respectively,

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

From the leftmost and middle figures, we see that by decreasing the off-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:

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

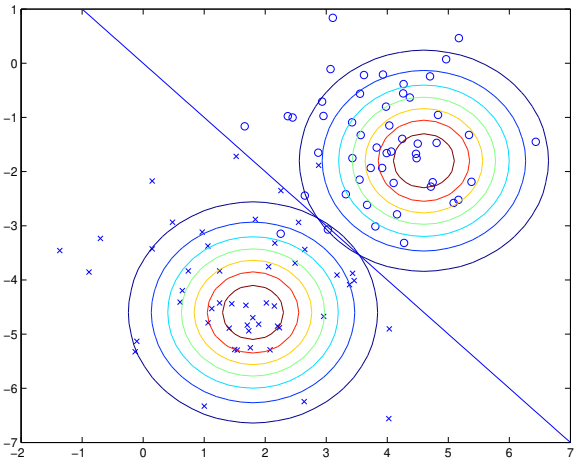
Here, the parameters of our model are ϕ, Σ, μ_0 and μ_1. (Note that while there're two different mean vectors μ_0 and μ_1, this model is usually applied using only one covariance matrix Σ.) The log-likelihood of the data is given by

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

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. Let's 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**.

Let's 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

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 generative 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.

---

number of words modeled and hence reducing our computational and space requirements, 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.

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 $\{(x^{(i)}, y^{(i)}); i = 1, \ldots, m\}$, we can write down the joint likelihood of the data:

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

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

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

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. Let's 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 let's 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

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

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, let's 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, $\phi_{k|y=1} = p(x_j = k|y = 1)$ (for any j) and $\phi_{i|y=0} = p(x_j = k|y = 0)$. 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

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

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 are 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.

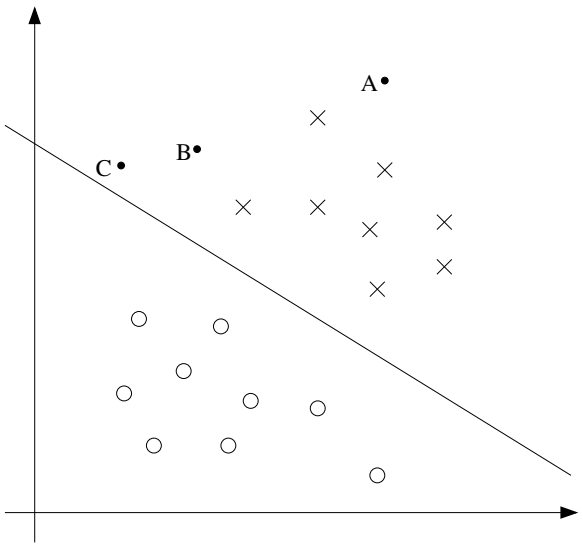
1 Margins: Intuition

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 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 our 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

Let's 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), 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, 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.

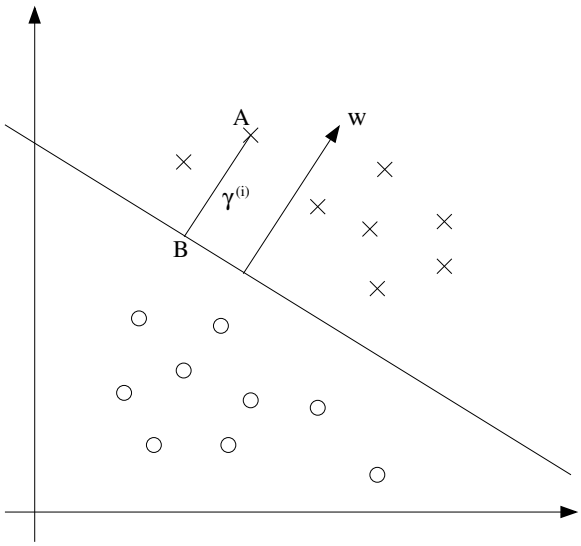
For a linear classifier with the choice of g given above (taking values in $\{-1, 1\}$), there's one property of the functional margin that makes it not a very good measure of confidence, however. Given our choice of g, we note that if we replace w with $2w$ and b with $2b$, then since $g(w^T x + b)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

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 to be 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, let's 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, let's try transforming the problem into a nicer one. Consider:

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

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.

Let's 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

Let's 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, let's 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. Let's 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:

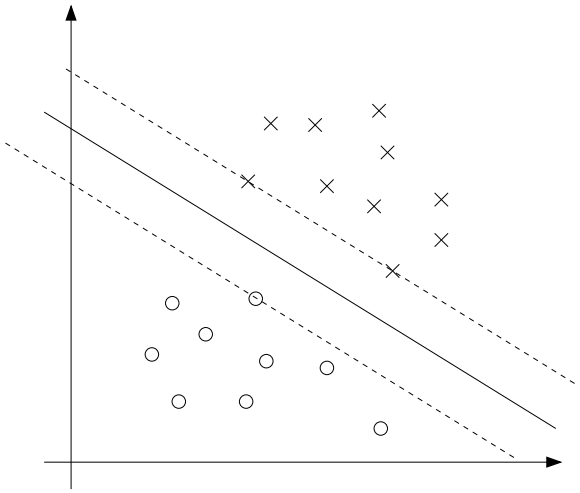
$$\begin{aligned}
\min_{\gamma, w, b} \quad & \frac{1}{2}||w||^2 \\
\text{s.t.} \quad & y^{(i)}(w^T x^{(i)} + b) \geq 1, \quad i = 1, \ldots, m
\end{aligned}$$

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.

Let's 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.

Let's 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, let's 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 $\phi(x)$.

Let's 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 $K(x, z) = (x^Tz + c)^d$ 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, let's 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.

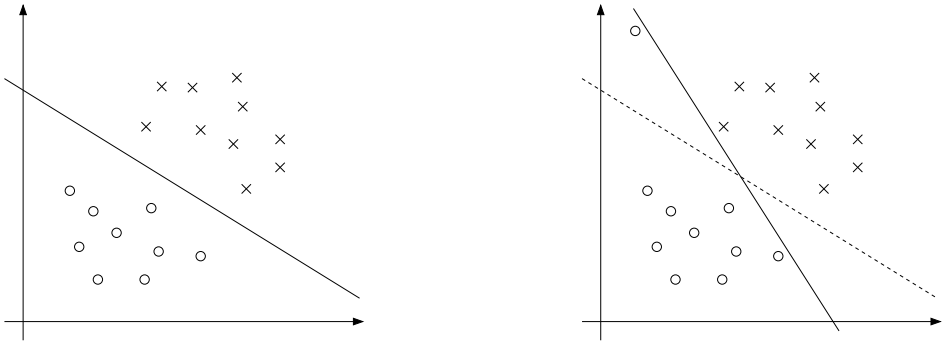
In class, we also briefly talked about a couple of other examples of kernels. For instance, consider the digit recognition problem, in which given an image (16x16 pixels) of a handwritten digit (0-9), we have to figure out which digit it was. Using either a simple polynomial kernel $K(x, z) = (x^T z)^d$ 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 letters, then there are 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 has functional margin $1 - \xi_i$ (with $\xi > 0$), 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 $||w||^2$ small (which we saw earlier makes the margin large) 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, let's 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 $i = 1, \ldots, m$, {

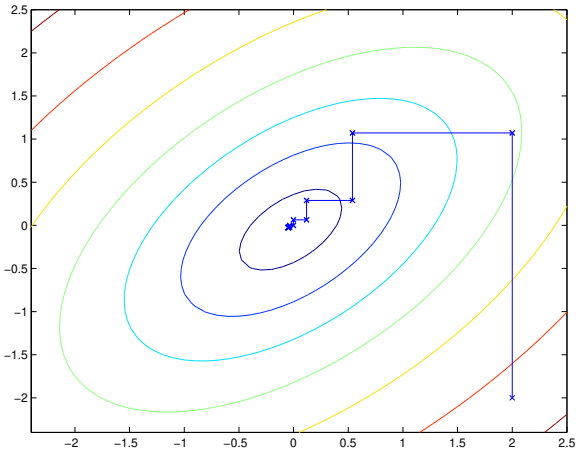
$\alpha_i := \arg\max_{\hat{\alpha}_i} W(\alpha_1, \ldots, \alpha_{i-1}, \hat{\alpha}_i, \alpha_{i+1}, \ldots, \alpha_m)$.

}

}

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:

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

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

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

Let's 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. Let's now briefly sketch the main ideas for deriving the efficient update.

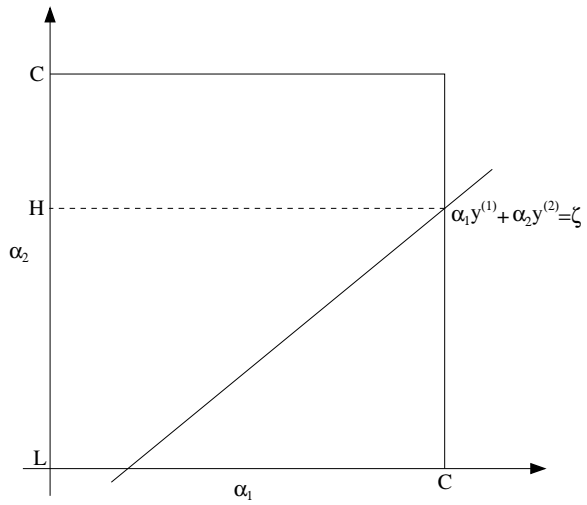
Let's say we currently have some setting of the α_i's that satisfy the constraints (18-19), and suppose we've decided to hold $\alpha_3, \ldots, \alpha_m$ fixed, and want to reoptimize $W(\alpha_1, \alpha_2, \ldots, \alpha_m)$ with respect to α_1 and α_2 (subject to the constraints). From (19), we require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

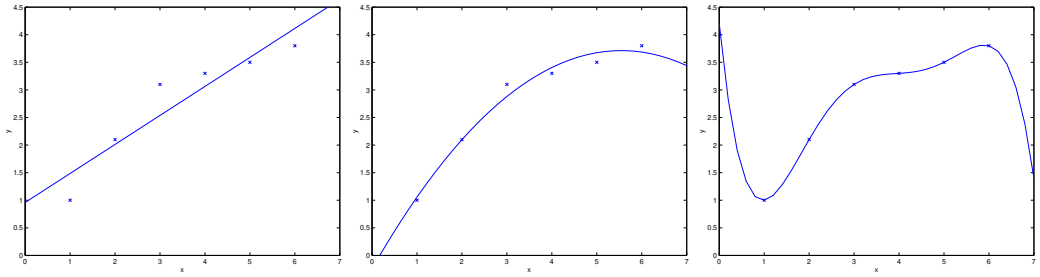
CS229 Lecture notes

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, let's 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 $\mathcal{H}$

Let's 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, give 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 shows 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, let's 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 $\mathcal{H}$!

Let's 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?

Let's 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.)

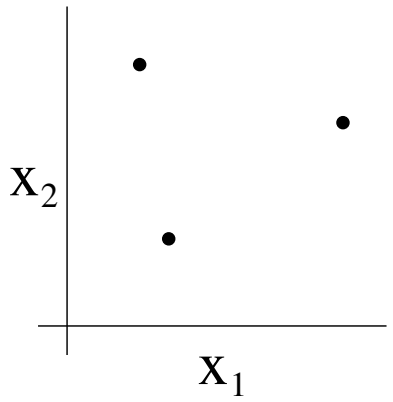
The other part of our previous argument that's slightly unsatisfying is that it relies on the parameterization of $\mathcal{H}$. Intuitively, this doesn't seem like it should matter: We had written the class of linear classifiers as $h_\theta(x) = 1\{\theta_0 + \theta_1 x_1 + \cdots \theta_n x_n \geq 0\}$, 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, let's 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 $i = 1, \ldots d$.

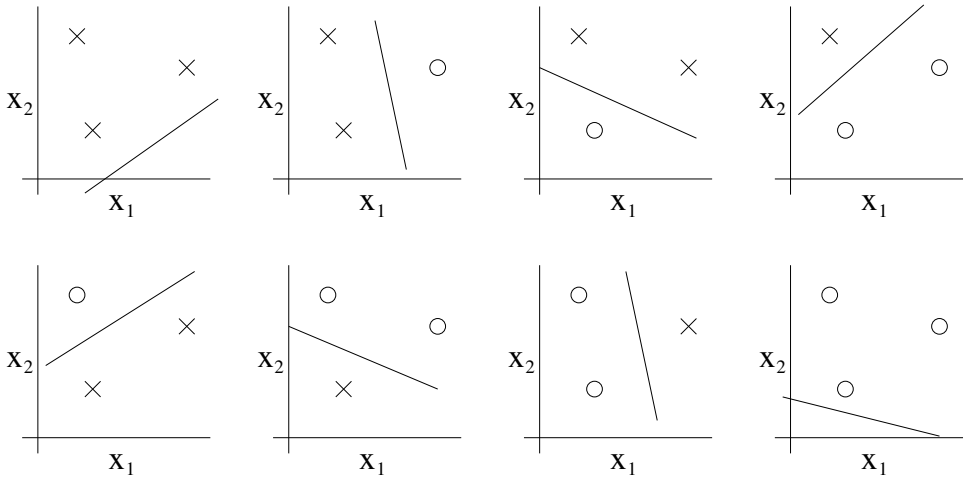
Given a hypothesis class $\mathcal{H}$, we then define its **Vapnik-Chervonenkis dimension**, written $\text{VC}(\mathcal{H})$, to be the size of the largest set that is shattered by $\mathcal{H}$. (If $\mathcal{H}$ can shatter arbitrarily large sets, then $\text{VC}(\mathcal{H}) = \infty$.)

For instance, consider the following set of three points:



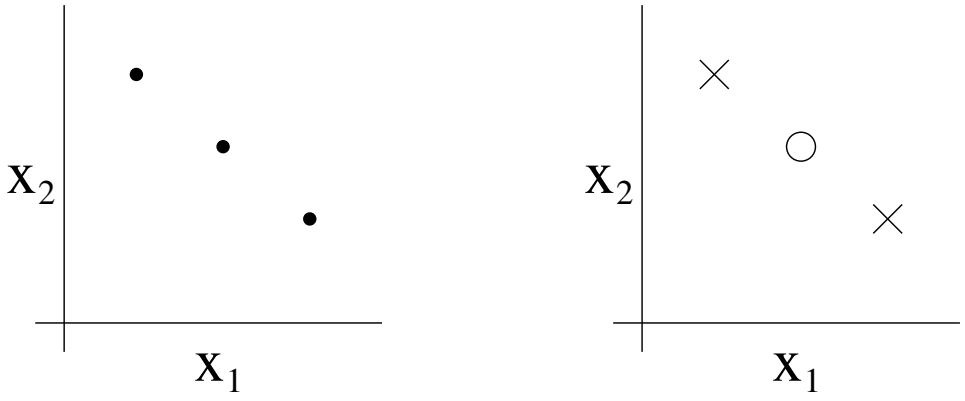
Can the set $\mathcal{H}$ of linear classifiers in two dimensions ($h(x) = 1\{\theta_0 + \theta_1 x_1 + \theta_2 x_2 \geq 0\}$) can shatter the set above? The answer is yes. Specifically, we

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 VII
Regularization and model selection

Suppose we are trying to 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

Let's 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. Let's 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 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.

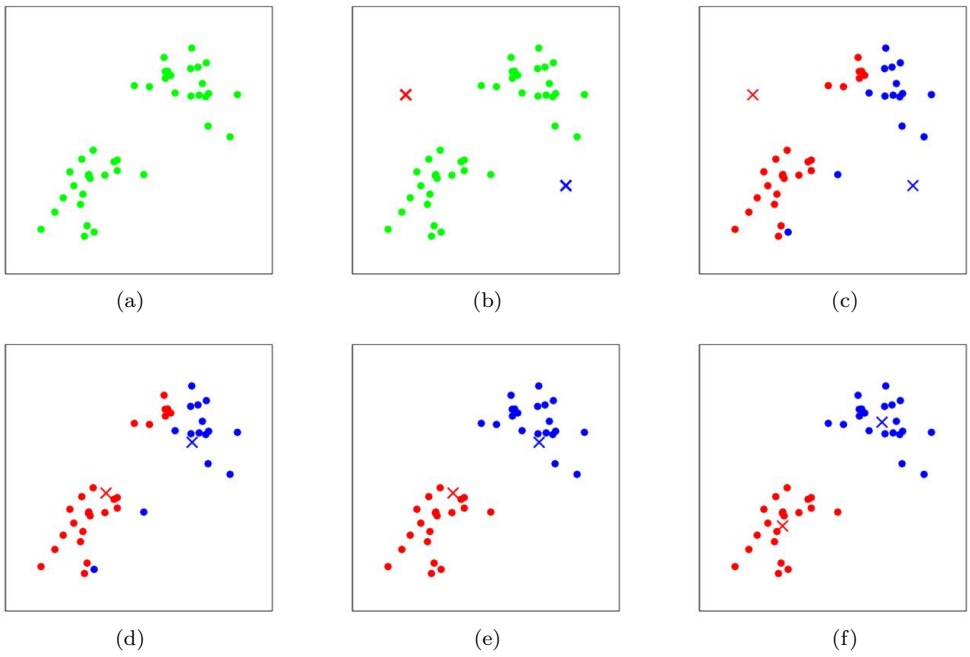


Figure 1: K-means algorithm. Training examples are shown as dots, and cluster centroids are shown as crosses. (a) Original dataset. (b) Random initial cluster centroids (in this instance, not chosen to be equal to two training examples). (c-f) Illustration of running two iterations of k-means. In each 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 depending 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, μ 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 positive definite, written $H > 0$). Jensen's inequality can then be stated as follows:

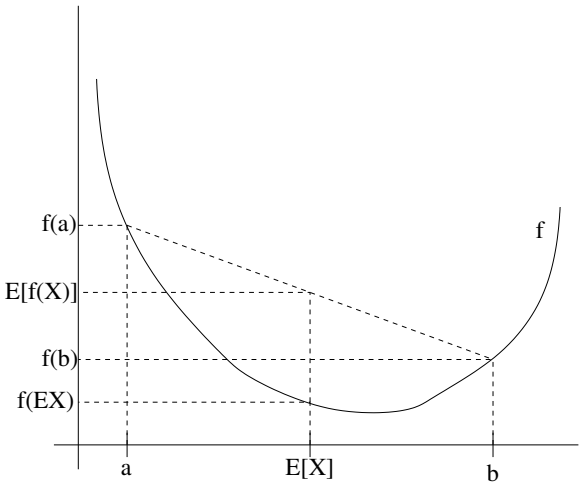
Theorem. Let f be a convex function, and let X be a random variable. Then:

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

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

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

then 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, let's 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

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

Let's 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.

Let's 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 where 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 dimensions; 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.

Let's 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:

$$\ell(\mu, \Lambda, \Psi) = \log \prod_{i=1}^{m} \frac{1}{(2\pi)^{n/2} |\Lambda\Lambda^T + \Psi|^{1/2}} \exp\left(-\frac{1}{2} (x^{(i)} - \mu)^T (\Lambda\Lambda^T + \Psi)^{-1} (x^{(i)} - \mu) \right).$$

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

$$Q_i(z^{(i)}) = \frac{1}{(2\pi)^{k/2} |\Sigma_{z^{(i)}|x^{(i)}}|^{1/2}} \exp\left(-\frac{1}{2} (z^{(i)} - \mu_{z^{(i)}|x^{(i)}})^T \Sigma_{z^{(i)}|x^{(i)}}^{-1} (z^{(i)} - \mu_{z^{(i)}|x^{(i)}}) \right).$$

Let's 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:

$$\begin{aligned}
&\sum_{i=1}^{m} \mathrm{E}\left[\log p(x^{(i)}|z^{(i)}; \mu, \Lambda, \Psi)\right] \\
&= \sum_{i=1}^{m} \mathrm{E}\left[\log \frac{1}{(2\pi)^{n/2}|\Psi|^{1/2}} \exp\left(-\frac{1}{2}(x^{(i)} - \mu - \Lambda z^{(i)})^T \Psi^{-1} (x^{(i)} - \mu - \Lambda z^{(i)})\right)\right] \\
&= \sum_{i=1}^{m} \mathrm{E}\left[-\frac{1}{2}\log|\Psi| - \frac{n}{2}\log(2\pi) - \frac{1}{2}(x^{(i)} - \mu - \Lambda z^{(i)})^T \Psi^{-1} (x^{(i)} - \mu - \Lambda z^{(i)})\right]
\end{aligned}$$

Let's 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:

$$\begin{aligned}
&\nabla_\Lambda \sum_{i=1}^{m} -\mathrm{E}\left[\frac{1}{2}(x^{(i)} - \mu - \Lambda z^{(i)})^T \Psi^{-1} (x^{(i)} - \mu - \Lambda z^{(i)})\right] \\
&= \sum_{i=1}^{m} \nabla_\Lambda \mathrm{E}\left[-\mathrm{tr}\frac{1}{2} z^{(i)T} \Lambda^T \Psi^{-1} \Lambda z^{(i)} + \mathrm{tr} z^{(i)T} \Lambda^T \Psi^{-1} (x^{(i)} - \mu)\right] \\
&= \sum_{i=1}^{m} \nabla_\Lambda \mathrm{E}\left[-\mathrm{tr}\frac{1}{2} \Lambda^T \Psi^{-1} \Lambda z^{(i)} z^{(i)T} + \mathrm{tr}\Lambda^T \Psi^{-1} (x^{(i)} - \mu) z^{(i)T}\right] \\
&= \sum_{i=1}^{m} \mathrm{E}\left[-\Psi^{-1} \Lambda z^{(i)} z^{(i)T} + \Psi^{-1} (x^{(i)} - \mu) z^{(i)T}\right]
\end{aligned}$$

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, let's 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

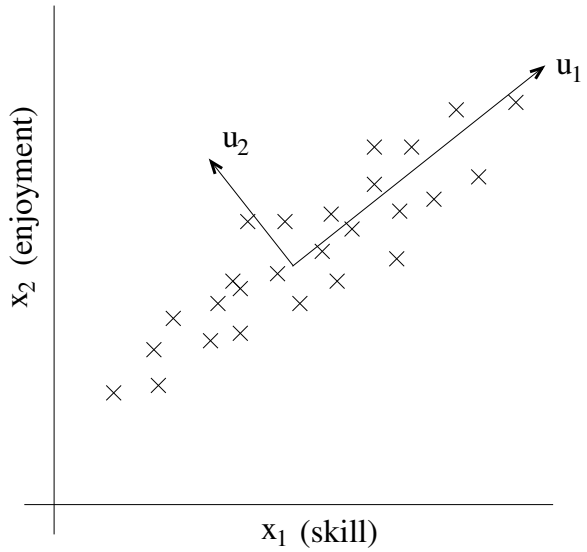
In our discussion of factor analysis, we gave a way to model data $x \in \mathbb{R}^n$ as "approximately" lying in some k-dimension subspace, where $k \ll n$. Specifically, we imagined that each point $x^{(i)}$ was created by first generating some $z^{(i)}$ lying in the k-dimension affine space $\{\Lambda z + \mu; z \in \mathbb{R}^k\}$, 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 a 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. Let $x^{(i)} \in \mathbb{R}^n$ 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:

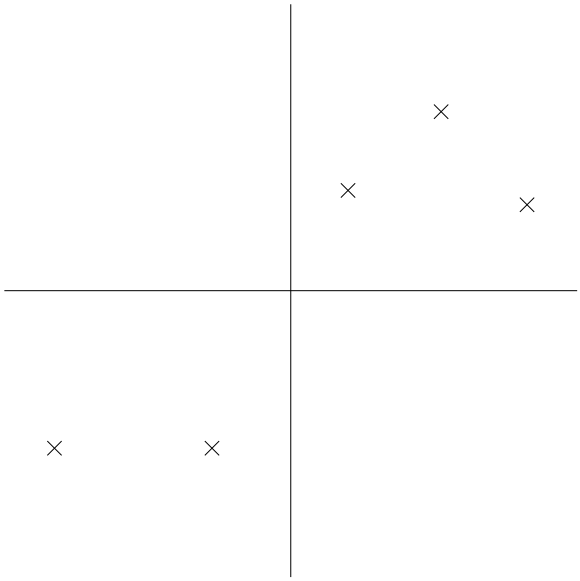
1. Let $\mu = \frac{1}{m} \sum_{i=1}^{m} x^{(i)}$.
2. Replace each $x^{(i)}$ with $x^{(i)} - \mu$.
3. Let $\sigma_j^2 = \frac{1}{m} \sum_i (x_j^{(i)})^2$
4. Replace each $x_j^{(i)}$ with $x_j^{(i)} / \sigma_j$.

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

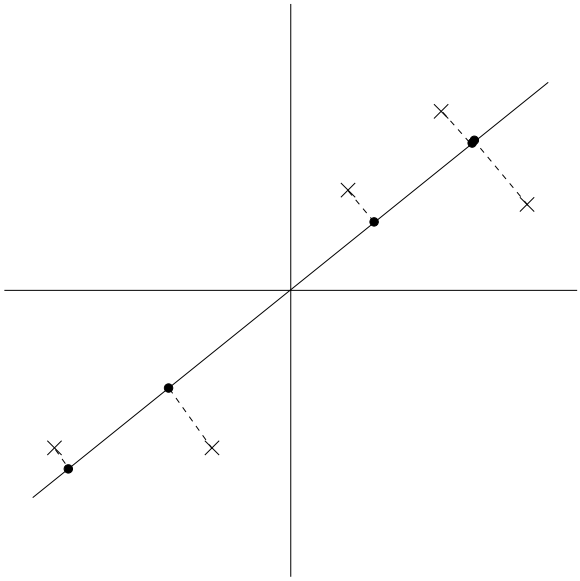
example of this is if each data point represented a grayscale image, and each $x_j^{(i)}$ took a value in $\{0, 1, \ldots, 255\}$ corresponding to the intensity value of pixel j in image i.

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.

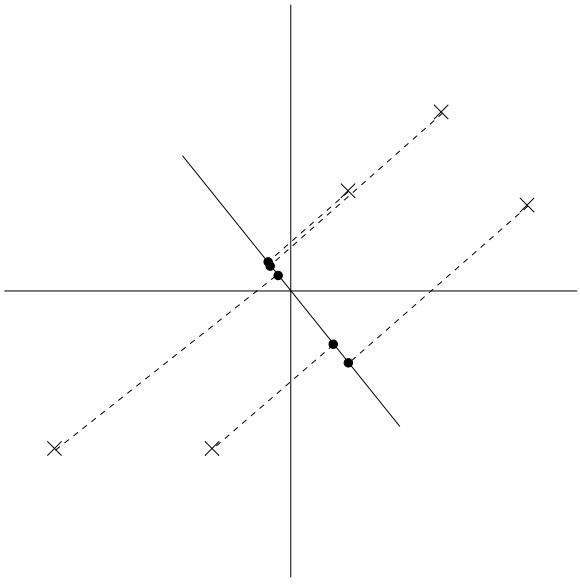
Consider the following dataset, on which we have already carried out the normalization steps:



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; we will close our discussion with a few 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 let's 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 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've 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.

4 Continuous state MDPs

So far, we've focused our attention on MDPs with a finite number of states. We now discuss algorithms for MDPs that may have an infinite number of states. For example, for a car, we might represent the state as $(x, y, \theta, \dot{x}, \dot{y}, \dot{\theta})$, comprising its position (x, y); orientation θ; velocity in the x and y directions $\dot{x}$ and $\dot{y}$; and angular velocity $\dot{\theta}$. Hence, $S = \mathbb{R}^6$ is an infinite set of states,

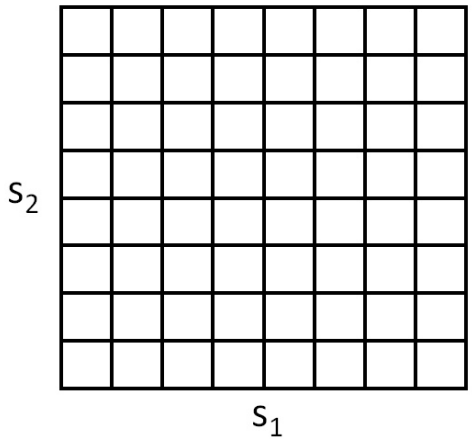
because there is an infinite number of possible positions and orientations for the car.[2] Similarly, the inverted pendulum you saw in PS4 has states $(x, \theta, \dot{x}, \dot{\theta})$, where θ is the angle of the pole. And, a helicopter flying in 3d space has states of the form $(x, y, z, \phi, \theta, \psi, \dot{x}, \dot{y}, \dot{z}, \dot{\phi}, \dot{\theta}, \dot{\psi})$, where here the roll ϕ, pitch θ, and yaw ψ angles specify the 3d orientation of the helicopter.

In this section, we will consider settings where the state space is $S = \mathbb{R}^n$, and describe ways for solving such MDPs.

4.1 Discretization

Perhaps the simplest way to solve a continuous-state MDP is to discretize the state space, and then to use an algorithm like value iteration or policy iteration, as described previously.

For example, if we have 2d states (s_1, s_2), we can use a grid to discretize the state space:



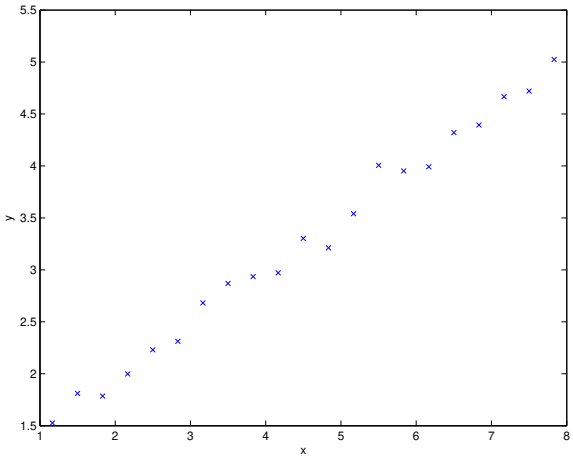
Here, each grid cell represents a separate discrete state $\bar{s}$. We can then approximate the continuous-state MDP via a discrete-state one $(\bar{S}, A, \{P_{\bar{s}a}\}, \gamma, R)$, where $\bar{S}$ is the set of discrete states, $\{P_{\bar{s}a}\}$ are our state transition probabilities over the discrete states, and so on. We can then use value iteration or policy iteration to solve for the $V^*(\bar{s})$ and $\pi^*(\bar{s})$ in the discrete state MDP $(\bar{S}, A, \{P_{\bar{s}a}\}, \gamma, R)$. When our actual system is in some continuous-valued state $s \in S$ and we need to pick an action to execute, we compute the corresponding discretized state $\bar{s}$, and execute action $\pi^*(\bar{s})$.

This discretization approach can work well for many problems. However, there are two downsides. First, it uses a fairly naive representation for V^*

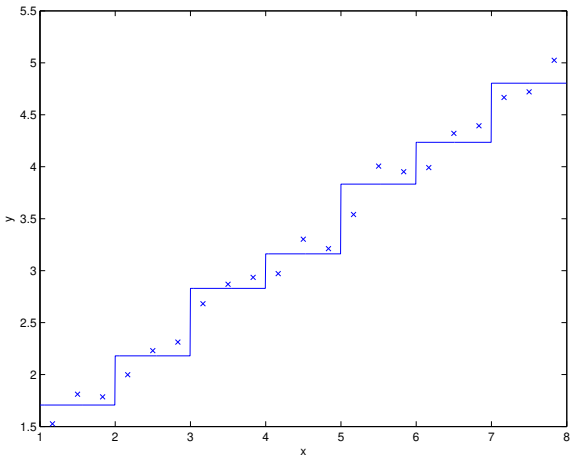
[2] Technically, θ is an orientation and so the range of θ is better written $\theta \in [-\pi, \pi)$ than $\theta \in \mathbb{R}$; but for our purposes, this distinction is not important.

(and π^*). Specifically, it assumes that the value function is takes a constant value over each of the discretization intervals (i.e., that the value function is piecewise constant in each of the gridcells).

To better understand the limitations of such a representation, consider a *supervised learning* problem of fitting a function to this dataset:



Clearly, linear regression would do fine on this problem. However, if we instead discretize the x-axis, and then use a representation that is piecewise constant in each of the discretization intervals, then our fit to the data would look like this:



This piecewise constant representation just isn't a good representation for many smooth functions. It results in little smoothing over the inputs, and no generalization over the different grid cells. Using this sort of representation, we would also need a very fine discretization (very small grid cells) to get a good approximation.

A second downside of this representation is called the **curse of dimensionality**. Suppose $S = \mathbb{R}^n$, and we discretize each of the n dimensions of the state into k values. Then the total number of discrete states we have is k^n. This grows exponentially quickly in the dimension of the state space n, and thus does not scale well to large problems. For example, with a 10d state, if we discretize each state variable into 100 values, we would have $100^{10} = 10^{20}$ discrete states, which is far too many to represent even on a modern desktop computer.

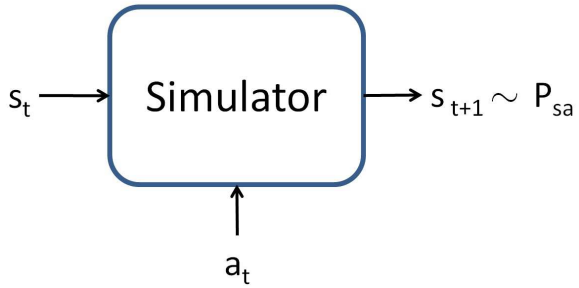
As a rule of thumb, discretization usually works extremely well for 1d and 2d problems (and has the advantage of being simple and quick to implement). Perhaps with a little bit of cleverness and some care in choosing the discretization method, it often works well for problems with up to 4d states. If you're extremely clever, and somewhat lucky, you may even get it to work for some 6d problems. But it very rarely works for problems any higher dimensional than that.

4.2 Value function approximation

We now describe an alternative method for finding policies in continuous-state MDPs, in which we approximate V^* directly, without resorting to discretization. This approach, caled value function approximation, has been successfully applied to many RL problems.

4.2.1 Using a model or simulator

To develop a value function approximation algorithm, we will assume that we have a **model**, or **simulator**, for the MDP. Informally, a simulator is a black-box that takes as input any (continuous-valued) state s_t and action a_t, and outputs a next-state s_{t+1} sampled according to the state transition probabilities $P_{s_t a_t}$:



There're several ways that one can get such a model. One is to use physics simulation. For example, the simulator for the inverted pendulum

in PS4 was obtained by using the laws of physics to calculate what position and orientation the cart/pole will be in at time $t+1$, given the current state at time t and the action a taken, assuming that we know all the parameters of the system such as the length of the pole, the mass of the pole, and so on. Alternatively, one can also use an off-the-shelf physics simulation software package which takes as input a complete physical description of a mechanical system, the current state s_t and action a_t, and computes the state s_{t+1} of the system a small fraction of a second into the future.[3]

An alternative way to get a model is to learn one from data collected in the MDP. For example, suppose we execute m **trials** in which we repeatedly take actions in an MDP, each trial for T timesteps. This can be done picking actions at random, executing some specific policy, or via some other way of choosing actions. We would then observe m state sequences like the following:

$$\begin{aligned}
&s_0^{(1)} \xrightarrow{a_0^{(1)}} s_1^{(1)} \xrightarrow{a_1^{(1)}} s_2^{(1)} \xrightarrow{a_2^{(1)}} \cdots \xrightarrow{a_{T-1}^{(1)}} s_T^{(1)} \\
&s_0^{(2)} \xrightarrow{a_0^{(2)}} s_1^{(2)} \xrightarrow{a_1^{(2)}} s_2^{(2)} \xrightarrow{a_2^{(2)}} \cdots \xrightarrow{a_{T-1}^{(2)}} s_T^{(2)} \\
&\cdots \\
&s_0^{(m)} \xrightarrow{a_0^{(m)}} s_1^{(m)} \xrightarrow{a_1^{(m)}} s_2^{(m)} \xrightarrow{a_2^{(m)}} \cdots \xrightarrow{a_{T-1}^{(m)}} s_T^{(m)}
\end{aligned}$$

We can then apply a learning algorithm to predict s_{t+1} as a function of s_t and a_t.

For example, one may choose to learn a linear model of the form

$$s_{t+1} = As_t + Ba_t, \tag{5}$$

using an algorithm similar to linear regression. Here, the parameters of the model are the matrices A and B, and we can estimate them using the data collected from our m trials, by picking

$$\arg\min_{A,B} \sum_{i=1}^{m} \sum_{t=0}^{T-1} \left\| s_{t+1}^{(i)} - \left(As_t^{(i)} + Ba_t^{(i)} \right) \right\|^2 .$$

(This corresponds to the maximum likelihood estimate of the parameters.)

Having learned A and B, one option is to build a **deterministic** model, in which given an input s_t and a_t, the output s_{t+1} is exactly determined.

[3]Open Dynamics Engine (http://www.ode.com) is one example of a free/open-source physics simulator that can be used to simulate systems like the inverted pendulum, and that has been a reasonably popular choice among RL researchers.

Specifically, we always compute s_{t+1} according to Equation (5). Alternatively, we may also build a **stochastic** model, in which s_{t+1} is a random function of the inputs, by modelling it as

$$s_{t+1} = As_t + Ba_t + \epsilon_t,$$

where here ϵ_t is a noise term, usually modeled as $\epsilon_t \sim \mathcal{N}(0, \Sigma)$. (The covariance matrix Σ can also be estimated from data in a straightforward way.)

Here, we've written the next-state s_{t+1} as a linear function of the current state and action; but of course, non-linear functions are also possible. Specifically, one can learn a model $s_{t+1} = A\phi_s(s_t) + B\phi_a(a_t)$, where ϕ_s and ϕ_a are some non-linear feature mappings of the states and actions. Alternatively, one can also use non-linear learning algorithms, such as locally weighted linear regression, to learn to estimate s_{t+1} as a function of s_t and a_t. These approaches can also be used to build either deterministic or stochastic simulators of an MDP.

4.2.2 Fitted value iteration

We now describe the **fitted value iteration** algorithm for approximating the value function of a continuous state MDP. In the sequel, we will assume that the problem has a continuous state space $S = \mathbb{R}^n$, but that the action space A is small and discrete.[4]

Recall that in value iteration, we would like to perform the update

$$\begin{aligned} V(s) &:= R(s) + \gamma \max_a \int_{s'} P_{sa}(s')V(s')ds' && (6) \\ &= R(s) + \gamma \max_a \mathrm{E}_{s' \sim P_{sa}}[V(s')] && (7) \end{aligned}$$

(In Section 2, we had written the value iteration update with a summation $V(s) := R(s) + \gamma \max_a \sum_{s'} P_{sa}(s')V(s')$ rather than an integral over states; the new notation reflects that we are now working in continuous states rather than discrete states.)

The main idea of fitted value iteration is that we are going to approximately carry out this step, over a finite sample of states $s^{(1)}, \ldots, s^{(m)}$. Specifically, we will use a supervised learning algorithm—linear regression in our

[4]In practice, most MDPs have much smaller action spaces than state spaces. E.g., a car has a 6d state space, and a 2d action space (steering and velocity controls); the inverted pendulum has a 4d state space, and a 1d action space; a helicopter has a 12d state space, and a 4d action space. So, discretizing ths set of actions is usually less of a problem than discretizing the state space would have been.

description below—to approximate the value function as a linear or non-linear function of the states:

$$V(s) = \theta^T \phi(s).$$

Here, ϕ is some appropriate feature mapping of the states.

For each state s in our finite sample of m states, fitted value iteration will first compute a quantity $y^{(i)}$, which will be our approximation to $R(s) + \gamma \max_a \mathrm{E}_{s' \sim P_{sa}}[V(s')]$ (the right hand side of Equation 7). Then, it will apply a supervised learning algorithm to try to get $V(s)$ close to $R(s) + \gamma \max_a \mathrm{E}_{s' \sim P_{sa}}[V(s')]$ (or, in other words, to try to get $V(s)$ close to $y^{(i)}$).

In detail, the algorithm is as follows:

1. Randomly sample m states $s^{(1)}, s^{(2)}, \ldots s^{(m)} \in S$.

2. Initialize $\theta := 0$.

3. Repeat {

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

        For each action $a \in A$ {

            Sample $s'_1, \ldots, s'_k \sim P_{s^{(i)}a}$ (using a model of the MDP).

            Set $q(a) = \frac{1}{k} \sum_{j=1}^{k} R(s^{(i)}) + \gamma V(s'_j)$

                // Hence, $q(a)$ is an estimate of $R(s^{(i)}) + \gamma \mathrm{E}_{s' \sim P_{s^{(i)}a}}[V(s')]$.

        }

        Set $y^{(i)} = \max_a q(a)$.

            // Hence, $y^{(i)}$ is an estimate of $R(s^{(i)}) + \gamma \max_a \mathrm{E}_{s' \sim P_{s^{(i)}a}}[V(s')]$.

    }

    // In the original value iteration algorithm (over discrete states)

    // we updated the value function according to $V(s^{(i)}) := y^{(i)}$.

    // In this algorithm, we want $V(s^{(i)}) \approx y^{(i)}$, which we'll achieve

    // using supervised learning (linear regression).

    Set $\theta := \arg\min_\theta \frac{1}{2} \sum_{i=1}^{m} \left(\theta^T \phi(s^{(i)}) - y^{(i)}\right)^2$

}

Above, we had written out fitted value iteration using linear regression as the algorithm to try to make $V(s^{(i)})$ close to $y^{(i)}$. That step of the algorithm is completely analogous to a standard supervised learning (regression) problem in which we have a training set $(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(m)}, y^{(m)})$, and want to learn a function mapping from x to y; the only difference is that here s plays the role of x. Even though our description above used linear regression, clearly other regression algorithms (such as locally weighted linear regression) can also be used.

Unlike value iteration over a discrete set of states, fitted value iteration cannot be proved to always to converge. However, in practice, it often does converge (or approximately converge), and works well for many problems. Note also that if we are using a deterministic simulator/model of the MDP, then fitted value iteration can be simplified by setting $k = 1$ in the algorithm. This is because the expectation in Equation (7) becomes an expectation over a deterministic distribution, and so a single example is sufficient to exactly compute that expectation. Otherwise, in the algorithm above, we had to draw k samples, and average to try to approximate that expectation (see the definition of $q(a)$, in the algorithm pseudo-code).

Finally, fitted value iteration outputs V, which is an approximation to V^*. This implicitly defines our policy. Specifically, when our system is in some state s, and we need to choose an action, we would like to choose the action

$$\arg\max_a \mathrm{E}_{s' \sim P_{sa}}[V(s')] \tag{8}$$

The process for computing/approximating this is similar to the inner-loop of fitted value iteration, where for each action, we sample $s'_1, \ldots, s'_k \sim P_{sa}$ to approximate the expectation. (And again, if the simulator is deterministic, we can set $k = 1$.)

In practice, there're often other ways to approximate this step as well. For example, one very common case is if the simulator is of the form $s_{t+1} = f(s_t, a_t) + \epsilon_t$, where f is some determinstic function of the states (such as $f(s_t, a_t) = As_t + Ba_t$), and ϵ is zero-mean Gaussian noise. In this case, we can pick the action given by

$$\arg\max_a V(f(s, a)).$$

In other words, here we are just setting $\epsilon_t = 0$ (i.e., ignoring the noise in the simulator), and setting $k = 1$. Equivalently, this can be derived from Equation (8) using the approximation

$$\begin{aligned} \mathrm{E}_{s'}[V(s')] &\approx V(\mathrm{E}_{s'}[s']) &(9)\\ &= V(f(s, a)), &(10) \end{aligned}$$

where here the expection is over the random $s' \sim P_{sa}$. So long as the noise terms ϵ_t are small, this will usually be a reasonable approximation.

However, for problems that don't lend themselves to such approximations, having to sample $k|A|$ states using the model, in order to approximate the expectation above, can be computationally expensive.

CS229 Lecture Notes

Andrew Ng and Kian Katanforoosh

Deep Learning

We now begin our study of deep learning. In this set of notes, we give an overview of neural networks, discuss vectorization and discuss training neural networks with backpropagation.

1 Neural Networks

We will start small and slowly build up a neural network, step by step. Recall the housing price prediction problem from before: given the size of the house, we want to predict the price.

Previously, we fitted a straight line to the graph. Now, instead of fitting a straight line, we wish prevent negative housing prices by setting the absolute minimum price as zero. This produces a "kink" in the graph as shown in Figure 1.

Our goal is to input some input x into a function $f(x)$ that outputs the price of the house y. Formally, $f : x \to y$. One of the simplest possible neural networks is to define $f(x)$ as a single "neuron" in the network where $f(x) = \max(ax + b, 0)$, for some coefficients a, b. What $f(x)$ does is return a single value: $(ax + b)$ or zero, whichever is greater. In the context of neural networks, this function is called a ReLU (pronounced "ray-lu"), or rectified linear unit. A more complex neural network may take the single neuron described above and "stack" them together such that one neuron passes its output as input into the next neuron, resulting in a more complex function.

Let us now deepen the housing prediction example. In addition to the size of the house, suppose that you know the number of bedrooms, the zip code

Scribe: Albert Haque

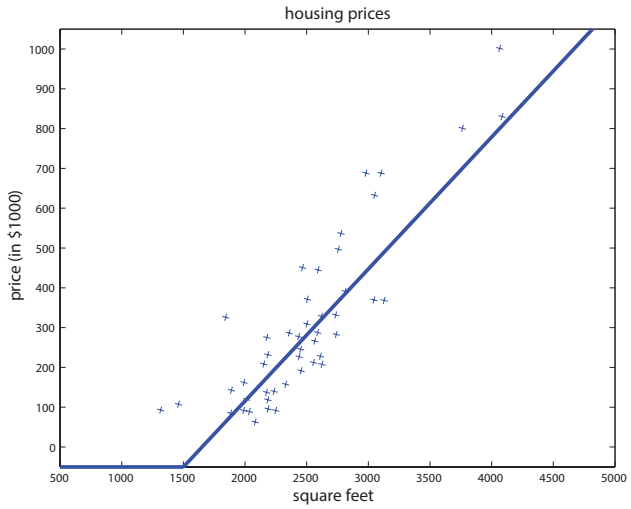


Figure 1: Housing prices with a "kink" in the graph.

and the wealth of the neighborhood. Building neural networks is analogous to Lego bricks: you take individual bricks and stack them together to build complex structures. The same applies to neural networks: we take individual neurons and stack them together to create complex neural networks.

Given these features (size, number of bedrooms, zip code, and wealth), we might then decide that the price of the house depends on the maximum family size it can accommodate. Suppose the family size is a function of the size of the house and number of bedrooms (see Figure 2). The zip code may provide additional information such as how walkable the neighborhood is (i.e., can you walk to the grocery store or do you need to drive everywhere). Combining the zip code with the wealth of the neighborhood may predict the quality of the local elementary school. Given these three derived features (family size, walkable, school quality), we may conclude that the price of the home ultimately depends on these three features.

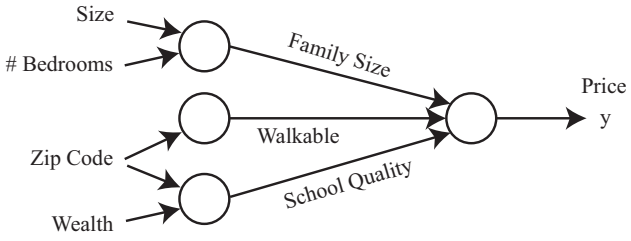


Figure 2: Diagram of a small neural network for predicting housing prices.

We have described this neural network as if you (the reader) already have the insight to determine these three factors ultimately affect the housing price. Part of the magic of a neural network is that all you need are the input features x and the output y while the neural network will figure out everything in the middle by itself. The process of a neural network learning the intermediate features is called *end-to-end learning*.

Following the housing example, formally, the input to a neural network is a set of input features x_1, x_2, x_3, x_4. We connect these four features to three neurons. These three "internal" neurons are called *hidden units*. The goal for the neural network is to automatically determine three relevant features such that the three features predict the price of a house. The only thing we must provide to the neural network is a sufficient number of training examples $(x^{(i)}, y^{(i)})$. Often times, the neural network will discover complex features which are very useful for predicting the output but may be difficult for a human to understand since it does not have a "common" meaning. This is why some people refer to neural networks as a *black box*, as it can be difficult to understand the features it has invented.

Let us formalize this neural network representation. Suppose we have three input features x_1, x_2, x_3 which are collectively called the *input layer*, four hidden units which are collectively called the *hidden layer* and one output neuron called the *output layer*. The term hidden layer is called "hidden" because we do not have the ground truth/training value for the hidden units. This is in contrast to the input and output layers, both of which we know the ground truth values from $(x^{(i)}, y^{(i)})$.

The first hidden unit requires the input x_1, x_2, x_3 and outputs a value denoted by a_1. We use the letter a since it refers to the neuron's "activation" value. In this particular example, we have a single hidden layer but it is possible to have multiple hidden layers. Let $a_1^{[1]}$ denote the output value of the first hidden unit in the first hidden layer. We use zero-indexing to refer to the layer numbers. That is, the input layer is layer 0, the first hidden layer is layer 1 and the output layer is layer 2. Again, more complex neural networks may have more hidden layers. Given this mathematical notation, the output of layer 2 is $a_1^{[2]}$. We can unify our notation:

$$x_1 = a_1^{[0]} \tag{1.1}$$

$$x_2 = a_2^{[0]} \tag{1.2}$$

$$x_3 = a_3^{[0]} \tag{1.3}$$

To clarify, $\text{foo}^{[1]}$ with brackets denotes anything associated with layer 1, $x^{(i)}$ with parenthesis refers to the i^{th} training example, and $a_j^{[\ell]}$ refers to the

activation of the j^{th} unit in layer ℓ. If we look at logistic regression $g(x)$ as a single neuron (see Figure 3):

$$g(x) = \frac{1}{1 + \exp(-w^T x)}$$

The input to the logistic regression $g(x)$ is three features x_1, x_2 and x_3 and it outputs an estimated value of y. We can represent $g(x)$ with a single neuron in the neural network. We can break $g(x)$ into two distinct computations: (1) $z = w^T x + b$ and (2) $a = \sigma(z)$ where $\sigma(z) = 1/(1 + e^{-z})$. Note the notational difference: previously we used $z = \theta^T x$ but now we are using $z = w^T x + b$, where w is a vector. Later in these notes you will see capital W to denote a matrix. The reasoning for this notational difference is conform with standard neural network notation. More generally, $a = g(z)$ where $g(z)$ is some activation function. Example activation functions include:

$$g(z) = \frac{1}{1 + e^{-z}} \qquad \text{(sigmoid)} \tag{1.4}$$

$$g(z) = \max(z, 0) \qquad \text{(ReLU)} \tag{1.5}$$

$$g(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \qquad \text{(tanh)} \tag{1.6}$$

In general, $g(z)$ is a non-linear function.

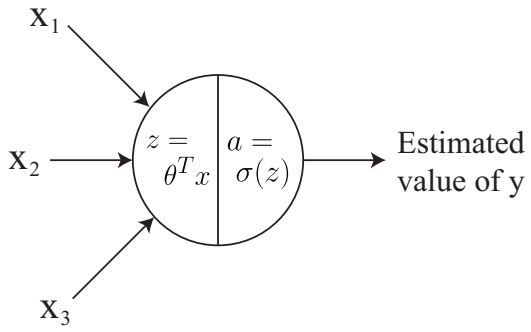


Figure 3: Logistic regression as a single neuron.

Returning to our neural network from before, the first hidden unit in the first hidden layer will perform the following computation:

$$z_1^{[1]} = {W_1^{[1]}}^T x + b_1^{[1]} \quad \text{and} \quad a_1^{[1]} = g(z_1^{[1]}) \tag{1.7}$$

where W is a matrix of parameters and W_1 refers to the first row of this matrix. The parameters associated with the first hidden unit is the vector

$W_1^{[1]} \in \mathbb{R}^3$ and the scalar $b_1^{[1]} \in \mathbb{R}$. For the second and third hidden units in the first hidden layer, the computation is defined as:

$$z_2^{[1]} = {W_2^{[1]}}^T x + b_2^{[1]} \quad \text{and} \quad a_2^{[1]} = g(z_2^{[1]})$$
$$z_3^{[1]} = {W_3^{[1]}}^T x + b_3^{[1]} \quad \text{and} \quad a_3^{[1]} = g(z_3^{[1]})$$

where each hidden unit has its corresponding parameters W and b. Moving on, the output layer performs the computation:

$$z_1^{[2]} = {W_1^{[2]}}^T a^{[1]} + b_1^{[2]} \quad \text{and} \quad a_1^{[2]} = g(z_1^{[2]}) \tag{1.8}$$

where $a^{[1]}$ is defined as the concatenation of all first layer activations:

$$a^{[1]} = \begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \\ a_3^{[1]} \\ a_4^{[1]} \end{bmatrix} \tag{1.9}$$

The activation $a_1^{[2]}$ from the second layer, which is a single scalar as defined by $a_1^{[2]} = g(z_1^{[2]})$, represents the neural network's final output prediction. Note that for regression tasks, one typically does not apply a non-linear function which is strictly positive (i.e., ReLU or sigmoid) because for some tasks, the ground truth y value may in fact be negative.

2 Vectorization

In order to implement a neural network at a reasonable speed, one must be careful when using for loops. In order to compute the hidden unit activations in the first layer, we must compute $z_1, ..., z_4$ and $a_1, ..., a_4$.

$$z_1^{[1]} = {W_1^{[1]}}^T x + b_1^{[1]} \quad \text{and} \quad a_1^{[1]} = g(z_1^{[1]}) \tag{2.1}$$
$$\vdots \qquad\qquad \vdots \qquad\qquad \vdots \tag{2.2}$$
$$z_4^{[1]} = {W_4^{[1]}}^T x + b_4^{[1]} \quad \text{and} \quad a_4^{[1]} = g(z_4^{[1]}) \tag{2.3}$$

The most natural way to implement this in code is to use a for loop. One of the treasures that deep learning has given to the field of machine learning is that deep learning algorithms have high computational requirements. As a result, code will run very slowly if you use for loops.

This gave rise to *vectorization*. Instead of using for loops, vectorization takes advantage of matrix algebra and highly optimized numerical linear algebra packages (e.g., BLAS) to make neural network computations run quickly. Before the deep learning era, a for loop may have been sufficient on smaller datasets, but modern deep networks and state-of-the-art datasets will be infeasible to run with for loops.

2.1 Vectorizing the Output Computation

We now present a method for computing $z_1, ..., z_4$ without a for loop. Using our matrix algebra, we can compute the activations:

$$\underbrace{\begin{bmatrix} z_1^{[1]} \\ \vdots \\ \vdots \\ z_4^{[1]} \end{bmatrix}}_{z^{[1]} \in \mathbb{R}^{4\times 1}} = \underbrace{\begin{bmatrix} — \; {W_1^{[1]}}^T \; — \\ — \; {W_2^{[1]}}^T \; — \\ \vdots \\ — \; {W_4^{[1]}}^T \; — \end{bmatrix}}_{W^{[1]} \in \mathbb{R}^{4\times 3}} \underbrace{\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}}_{x \in \mathbb{R}^{3\times 1}} + \underbrace{\begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ \vdots \\ b_4^{[1]} \end{bmatrix}}_{b^{[1]} \in \mathbb{R}^{4\times 1}} \tag{2.4}$$

Where the $\mathbb{R}^{n\times m}$ beneath each matrix indicates the dimensions. Expressing this in matrix notation: $z^{[1]} = W^{[1]}x + b^{[1]}$. To compute $a^{[1]}$ without a for loop, we can leverage vectorized libraries in Matlab, Octave, or Python which compute $a^{[1]} = g(z^{[1]})$ very fast by performing parallel element-wise operations. Mathematically, we defined the sigmoid function $g(z)$ as:

$$g(z) = \frac{1}{1+e^{-z}} \qquad \text{where } z \in \mathbb{R} \tag{2.5}$$

However, the sigmoid function can be defined not only for scalars but also vectors. In a Matlab/Octave-like pseudocode, we can define the sigmoid as:

$$g(z) = \texttt{1 ./ (1+exp(-}z\texttt{))} \qquad \text{where } z \in \mathbb{R}^n \tag{2.6}$$

where `./` denotes element-wise division. With this vectorized implementation, $a^{[1]} = g(z^{[1]})$ can be computed quickly.

To summarize the neural network so far, given an input $x \in \mathbb{R}^3$, we compute the hidden layer's activations with $z^{[1]} = W^{[1]}x + b^{[1]}$ and $a^{[1]} = g(z^{[1]})$. To compute the output layer's activations (i.e., neural network output):

$$\underbrace{z^{[2]}}_{1\times 1} = \underbrace{W^{[2]}}_{1\times 4}\underbrace{a^{[1]}}_{4\times 1} + \underbrace{b^{[2]}}_{1\times 1} \quad \text{and} \quad \underbrace{a^{[2]}}_{1\times 1} = g(\underbrace{z^{[2]}}_{1\times 1}) \tag{2.7}$$

Why do we not use the identity function for $g(z)$? That is, why not use $g(z) = z$? Assume for sake of argument that $b^{[1]}$ and $b^{[2]}$ are zeros. Using Equation (2.7), we have:

$$\begin{aligned} z^{[2]} &= W^{[2]}a^{[1]} && && (2.8)\\ &= W^{[2]}g(z^{[1]}) && \text{by definition} && (2.9)\\ &= W^{[2]}z^{[1]} && \text{since } g(z) = z && (2.10)\\ &= W^{[2]}W^{[1]}x && \text{from Equation (2.4)} && (2.11)\\ &= \tilde{W}x && \text{where } \tilde{W} = W^{[2]}W^{[1]} && (2.12) \end{aligned}$$

Notice how $W^{[2]}W^{[1]}$ collapsed into $\tilde{W}$. This is because applying a linear function to another linear function will result in a linear function over the original input (i.e., you can construct a $\tilde{W}$ such that $\tilde{W}x = W^{[2]}W^{[1]}x$). This loses much of the representational power of the neural network as often times the output we are trying to predict has a non-linear relationship with the inputs. Without non-linear activation functions, the neural network will simply perform linear regression.

2.2 Vectorization Over Training Examples

Suppose you have a training set with three examples. The activations for each example are as follows:

$$\begin{aligned} z^{[1](1)} &= W^{[1]}x^{(1)} + b^{[1]}\\ z^{[1](2)} &= W^{[1]}x^{(2)} + b^{[1]}\\ z^{[1](3)} &= W^{[1]}x^{(3)} + b^{[1]} \end{aligned}$$

Note the difference between square brackets $[\cdot]$, which refer to the layer number, and parenthesis $(\cdot)$, which refer to the training example number. Intuitively, one would implement this using a for loop. It turns out, we can vectorize these operations as well. First, define:

$$X = \begin{bmatrix} | & | & | \\ x^{(1)} & x^{(2)} & x^{(3)} \\ | & | & | \end{bmatrix} \tag{2.13}$$

Note that we are stacking training examples in columns and *not* rows. We can then combine this into a single unified formulation:

$$Z^{[1]} = \begin{bmatrix} | & | & | \\ z^{[1](1)} & z^{[1](2)} & z^{[1](3)} \\ | & | & | \end{bmatrix} = W^{[1]}X + b^{[1]} \tag{2.14}$$

You may notice that we are attempting to add $b^{[1]} \in \mathbb{R}^{4\times 1}$ to $W^{[1]}X \in \mathbb{R}^{4\times 3}$. Strictly following the rules of linear algebra, this is not allowed. In practice however, this addition is performed using *broadcasting*. We create an intermediate $\tilde{b}^{[1]} \in \mathbb{R}^{4\times 3}$:

$$\tilde{b}^{[1]} = \begin{bmatrix} | & | & | \\ b^{[1]} & b^{[1]} & b^{[1]} \\ | & | & | \end{bmatrix} \tag{2.15}$$

We can then perform the computation: $Z^{[1]} = W^{[1]}X + \tilde{b}^{[1]}$. Often times, it is not necessary to explicitly construct $\tilde{b}^{[1]}$. By inspecting the dimensions in (2.14), you can assume $b^{[1]} \in \mathbb{R}^{4\times 1}$ is correctly broadcast to $W^{[1]}X \in \mathbb{R}^{4\times 3}$.

Putting it together: Suppose we have a training set $(x^{(1)}, y^{(1)}), ..., (x^{(m)}, y^{(m)})$ where $x^{(i)}$ is a picture and $y^{(i)}$ is a binary label for whether the picture contains a cat or not (i.e., 1=contains a cat). First, we initialize the parameters $W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}$ to small random numbers. For each example, we compute the output "probability" from the sigmoid function $a^{[2](i)}$. Second, using the logistic regression log likelihood:

$$\sum_{i=1}^{m} \left(y^{(i)} \log a^{[2](i)} + (1 - y^{(i)}) \log(1 - a^{[2](i)}) \right) \tag{2.16}$$

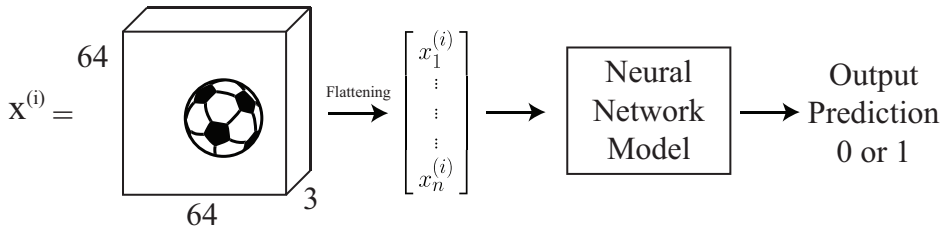
Finally, we maximize this function using gradient ascent. This maximization procedure corresponds to training the neural network.

3 Backpropagation

Instead of the housing example, we now have a new problem. Suppose we wish to detect whether there is a soccer ball in an image or not. Given an input image $x^{(i)}$, we wish to output a binary prediction 1 if there is a ball in the image and 0 otherwise.

Aside: Images can be represented as a matrix with number of elements equal to the number of pixels. However, color images are digitally represented as a volume (i.e., three-channels; or three matrices stacked on each other). The number three is used because colors are represented as red-green-blue (RGB) values. In the diagram below, we have a $64 \times 64 \times 3$ image containing a soccer ball. It is *flattened* into a single vector containing 12,288 elements.

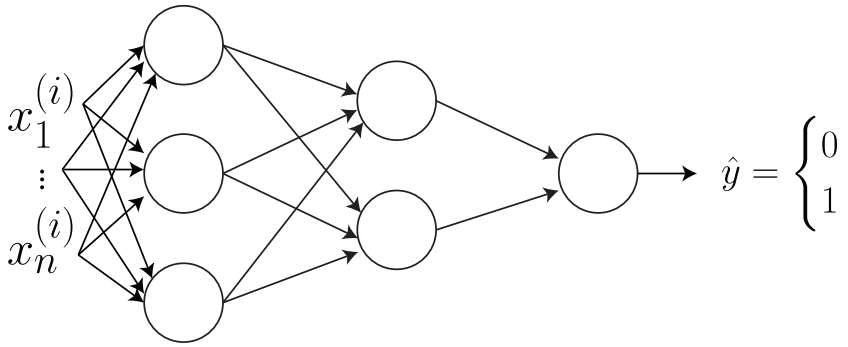
A neural network *model* consists of two components: (i) the network architecture, which defines how many layers, how many neurons, and how the neurons are connected and (ii) the parameters (values; also known as



weights). In this section, we will talk about how to learn the parameters. First we will talk about parameter initialization, optimization and analyzing these parameters.

3.1 Parameter Initialization

Consider a two layer neural network. On the left, the input is a flattened image vector $x^{(1)}, ..., x_n^{(i)}$. In the first hidden layer, notice how all inputs are connected to all neurons in the next layer. This is called a *fully connected* layer.



The next step is to compute how many parameters are in this network. One way of doing this is to compute the forward propagation by hand.

$$z^{[1]} = W^{[1]}x^{(i)} + b^{[1]} \tag{3.1}$$
$$a^{[1]} = g(z^{[1]}) \tag{3.2}$$
$$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]} \tag{3.3}$$
$$a^{[2]} = g(z^{[2]}) \tag{3.4}$$
$$z^{[3]} = W^{[3]}a^{[2]} + b^{[3]} \tag{3.5}$$
$$\hat{y}^{(i)} = a^{[3]} = g(z^{[3]}) \tag{3.6}$$

We know that $z^{[1]}, a^{[1]} \in \mathbb{R}^{3\times 1}$ and $z^{[2]}, a^{[2]} \in \mathbb{R}^{2\times 1}$ and $z^{[3]}, a^{[3]} \in \mathbb{R}^{1\times 1}$. As of now, we do not know the size of $W^{[1]}$. However, we can compute its size.

We know that $x \in \mathbb{R}^{n \times 1}$. This leads us to the following

$$z^{[1]} = W^{[1]}x^{(i)} = \mathbb{R}^{3\times 1} \quad \text{Written as sizes:} \quad \mathbb{R}^{3\times 1} = \mathbb{R}^{?\times ?} \times \mathbb{R}^{n\times 1} \tag{3.7}$$

Using matrix multiplication, we conclude that ?×? must be $3 \times n$. We also conclude that the bias is of size 3×1 because its size must match $W^{[1]}x^{(i)}$. We repeat this process for each hidden layer. This gives us:

$$W^{[2]} \in \mathbb{R}^{2\times 3}, b^{[2]} \in \mathbb{R}^{2\times 1} \quad \text{and} \quad W^{[3]} \in \mathbb{R}^{1\times 2}, b^{[3]} \in \mathbb{R}^{1\times 1} \tag{3.8}$$

In total, we have $3n + 3$ in the first layer, $2 \times 3 + 2$ in the second layer and $2 + 1$ in the third layer. This gives us a total of $3n + 14$ parameters.

Before we start training the neural network, we must select an initial value for these parameters. We do not use the value zero as the initial value. This is because the output of the first layer will always be the same since $W^{[1]}x^{(i)} + b^{[1]} = 0^{3\times 1}x^{(i)} + 0^{3\times 1}$ where $0^{n\times m}$ denotes a matrix of size $n \times m$ filled with zeros. This will cause problems later on when we try to update these parameters (i.e., the gradients will all be the same). The solution is to randomly initialize the parameters to small values (e.g., normally distributed around zero; $\mathcal{N}(0, 0.1)$). Once the parameters have been initialized, we can begin training the neural network with gradient descent.

The next step of the training process is to update the parameters. After a single forward pass through the neural network, the output will be a predicted value $\hat{y}$. We can then compute the loss $\mathcal{L}$, in our case the log loss:

$$\mathcal{L}(\hat{y}, y) = -\Big[(1 - y)\log(1 - \hat{y}) + y \log \hat{y}\Big] \tag{3.9}$$

The loss function $\mathcal{L}(\hat{y}, y)$ produces a single scalar value. For short, we will refer to the loss value as $\mathcal{L}$. Given this value, we now must update all parameters in layers of the neural network. For any given layer index ℓ, we update them:

$$W^{[\ell]} = W^{[\ell]} - \alpha \frac{\partial \mathcal{L}}{\partial W^{[\ell]}} \tag{3.10}$$

$$b^{[\ell]} = b^{[\ell]} - \alpha \frac{\partial \mathcal{L}}{\partial b^{[\ell]}} \tag{3.11}$$

where α is the learning rate. To proceed, we must compute the gradient with respect to the parameters: $\partial\mathcal{L}/\partial W^{[\ell]}$ and $\partial\mathcal{L}/\partial b^{[\ell]}$.

Remember, we made a decision to not set all parameters to zero. What if we had initialized all parameters to be zero? We know that $z^{[3]} = W^{[3]}a^{[2]} + b^{[3]}$

will evaluate to zero, because $W^{[3]}$ and $b^{[3]}$ are all zero. However, the output of the neural network is defined as $a^{[3]} = g(z^{[3]})$. Recall that $g(\cdot)$ is defined as the sigmoid function. This means $a^{[3]} = g(0) = 0.5$. Thus, no matter what value of $x^{(i)}$ we provide, the network will output $\hat{y} = 0.5$.

What if we had initialized all parameters to be the same non-zero value? In this case, consider the activations of the first layer:

$$a^{[1]} = g(z^{[1]}) = g(W^{[1]}x^{(i)} + b^{[1]}) \tag{3.12}$$

Each element of the activation vector $a^{[1]}$ will be the same (because $W^{[1]}$ contains all the same values). This behavior will occur at all layers of the neural network. As a result, when we compute the gradient, all neurons in a layer will be equally responsible for anything contributed to the final loss. We call this property *symmetry*. This means each neuron (within a layer) will receive the exact same gradient update value (i.e., all neurons will learn the same thing).

In practice, it turns out there is something better than random initialization. It is called Xavier/He initialization and initializes the weights:

$$w^{[\ell]} \sim \mathcal{N}\left(0, \sqrt{\frac{2}{n^{[\ell]} + n^{[\ell-1]}}}\right) \tag{3.13}$$

where $n^{[\ell]}$ is the number of neurons in layer ℓ. This acts as a mini-normalization technique. For a single layer, consider the variance of the input to the layer as $\sigma^{(in)}$ and the variance of the output (i.e., activations) of a layer to be $\sigma^{(out)}$. Xavier/He initialization encourages $\sigma^{(in)}$ to be similar to $\sigma^{(out)}$.

3.2 Optimization

Recall our neural network parameters: $W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}, W^{[3]}, b^{[3]}$. To update them, we use stochastic gradient descent (SGD) using the update rules in Equations (3.10) and (3.11). We will first compute the gradient with respect to $W^{[3]}$. The reason for this is that the influence of $W^{[1]}$ on the loss is more complex than that of $W^{[3]}$. This is because $W^{[3]}$ is "closer" to the

output $\hat{y}$, in terms of number of computations.

$$\frac{\partial \mathcal{L}}{\partial W^{[3]}} = -\frac{\partial}{\partial W^{[3]}}\Big((1-y)\log(1-\hat{y}) + y\log\hat{y}\Big) \tag{3.14}$$

$$= -(1-y)\frac{\partial}{\partial W^{[3]}}\log\Big(1 - g(W^{[3]}a^{[2]} + b^{[3]})\Big) \tag{3.15}$$

$$- y\frac{\partial}{\partial W^{[3]}}\log\Big(g(W^{[3]}a^{[2]} + b^{[3]})\Big) \tag{3.16}$$

$$= -(1-y)\frac{1}{1 - g(W^{[3]}a^{[2]} + b^{[3]})}(-1)g'(W^{[3]}a^{[2]} + b^{[3]}){a^{[2]}}^T \tag{3.17}$$

$$- y\frac{1}{g(W^{[3]}a^{[2]} + b^{[3]})}g'(W^{[3]}a^{[2]} + b^{[3]}){a^{[2]}}^T \tag{3.18}$$

$$= (1-y)\sigma(W^{[3]}a^{[2]} + b^{[3]}){a^{[2]}}^T - y(1 - \sigma(W^{[3]}a^{[2]} + b^{[3]})){a^{[2]}}^T \tag{3.19}$$

$$= (1-y)a^{[3]}{a^{[2]}}^T - y(1 - a^{[3]}){a^{[2]}}^T \tag{3.20}$$

$$= (a^{[3]} - y){a^{[2]}}^T \tag{3.21}$$

Note that we are using sigmoid for $g(\cdot)$. The derivative of the sigmoid function: $g' = \sigma' = \sigma(1-\sigma)$. Additionally $a^{[3]} = \sigma(W^{[3]}a^{[2]} + b^{[3]})$. At this point, we have finished computing the gradient for one parameter, $W^{[3]}$.

We will now compute the gradient for $W^{[2]}$. Instead of deriving $\partial\mathcal{L}/\partial W^{[2]}$, we can use the chain rule of calculus. We know that $\mathcal{L}$ depends on $\hat{y} = a^{[3]}$.

$$\frac{\partial \mathcal{L}}{\partial W^{[2]}} = \frac{\partial \mathcal{L}}{?}\frac{?}{\partial W^{[2]}} \tag{3.22}$$

If we look at the forward propagation, we know that loss $\mathcal{L}$ depends on $\hat{y} = a^{[3]}$. Using the chain rule, we can insert $\partial a^{[3]}/\partial a^{[3]}$:

$$\frac{\partial \mathcal{L}}{\partial W^{[2]}} = \frac{\partial \mathcal{L}}{\partial a^{[3]}}\frac{\partial a^{[3]}}{?}\frac{?}{\partial W^{[2]}} \tag{3.23}$$

We know that $a^{[3]}$ is directly related to $z^{[3]}$.

$$\frac{\partial \mathcal{L}}{\partial W^{[2]}} = \frac{\partial \mathcal{L}}{\partial a^{[3]}}\frac{\partial a^{[3]}}{\partial z^{[3]}}\frac{\partial z^{[3]}}{?}\frac{?}{\partial W^{[2]}} \tag{3.24}$$

Furthermore we know that $z^{[3]}$ is directly related to $a^{[2]}$. Note that we cannot use $W^{[2]}$ or $b^{[2]}$ because $a^{[2]}$ is the only common element between Equations (3.5) and (3.6). A common element is required for backpropagation.

$$\frac{\partial \mathcal{L}}{\partial W^{[2]}} = \frac{\partial \mathcal{L}}{\partial a^{[3]}}\frac{\partial a^{[3]}}{\partial z^{[3]}}\frac{\partial z^{[3]}}{\partial a^{[2]}}\frac{\partial a^{[2]}}{?}\frac{?}{\partial W^{[2]}} \tag{3.25}$$

Again, $a^{[2]}$ depends on $z^{[2]}$, which $z^{[2]}$ directly depends on $W^{[2]}$, which allows us to complete the chain:

$$\frac{\partial \mathcal{L}}{\partial W^{[2]}} = \frac{\partial \mathcal{L}}{\partial a^{[3]}} \frac{\partial a^{[3]}}{\partial z^{[3]}} \frac{\partial z^{[3]}}{\partial a^{[2]}} \frac{\partial a^{[2]}}{\partial z^{[2]}} \frac{\partial z^{[2]}}{\partial W^{[2]}} \tag{3.26}$$

Recall $\partial \mathcal{L}/\partial W^{[3]}$:

$$\frac{\partial \mathcal{L}}{\partial W^{[3]}} = (a^{[3]} - y) a^{[2]^T} \tag{3.27}$$

Since we computed $\partial \mathcal{L}/\partial W^{[3]}$ first, we know that $a^{[2]} = \partial z^{[3]}/\partial W^{[3]}$. Similarly we have $(a^{[3]} - y) = \partial \mathcal{L}/\partial z^{[3]}$. These can help us compute $\partial \mathcal{L}/\partial W^{[2]}$. We substitute these values into Equation (3.26). This gives us:

$$\frac{\partial \mathcal{L}}{\partial W^{[2]}} = \underbrace{\frac{\partial \mathcal{L}}{\partial a^{[3]}} \frac{\partial a^{[3]}}{\partial z^{[3]}}}_{(a^{[3]}-y)} \underbrace{\frac{\partial z^{[3]}}{\partial a^{[2]}}}_{W^{[3]}} \underbrace{\frac{\partial a^{[2]}}{\partial z^{[2]}}}_{g'(z^{[2]})} \underbrace{\frac{\partial z^{[2]}}{\partial W^{[2]}}}_{a^{[1]}} = (a^{[3]} - y) W^{[3]} g'(z^{[2]}) a^{[1]} \tag{3.28}$$

While we have greatly simplified the process, we are not done yet. Because we are computing derivatives in higher dimensions, the exact order of matrix multiplication required to compute Equation (3.28) is not clear. We must reorder the terms in Equation (3.28) such that the dimensions align. First, we note the dimensions of all the terms:

$$\underbrace{\frac{\partial \mathcal{L}}{\partial W^{[2]}}}_{2\times 3} = \underbrace{(a^{[3]} - y)}_{1\times 1} \underbrace{W^{[3]}}_{1\times 2} \underbrace{g'(z^{[2]})}_{2\times 1} \underbrace{a^{[1]}}_{3\times 1} \tag{3.29}$$

Notice how the terms do not align their shapes properly. We must rearrange the terms by using properties of matrix algebra such that the matrix operations produce a result with the correct output shape. The correct ordering is below:

$$\underbrace{\frac{\partial \mathcal{L}}{\partial W^{[2]}}}_{2\times 3} = \underbrace{W^{[3]^T}}_{2\times 1} \circ \underbrace{g'(z^{[2]})}_{2\times 1} \underbrace{(a^{[3]} - y)}_{1\times 1} \underbrace{a^{[1]^T}}_{1\times 3} \tag{3.30}$$

We leave the remaining gradients as an exercise to the reader. In calculating the gradients for the remaining parameters, it is important to use the intermediate results we have computed for $\partial \mathcal{L}/\partial W^{[2]}$ and $\partial \mathcal{L}/\partial W^{[3]}$, as these will be directly useful for computing the gradient.

Returning to optimization, we previously discussed stochastic gradient descent. Now we will talk about gradient descent. For any single layer ℓ, the update rule is defined as:

$$W^{[\ell]} = W^{[\ell]} - \alpha \frac{\partial J}{\partial W^{[\ell]}} \tag{3.31}$$

where J is the cost function $J = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}^{(i)}$ and $\mathcal{L}^{(i)}$ is the loss for a single example. The difference between the gradient descent update versus the stochastic gradient descent version is that the cost function J gives more accurate gradients whereas $\mathcal{L}^{(i)}$ may be noisy. Stochastic gradient descent attempts to approximate the gradient from (full) gradient descent. The disadvantage of gradient descent is that it can be difficult to compute all activations for all examples in a single forward or backwards propagation phase.

In practice, research and applications use *mini-batch gradient descent*. This is a compromise between gradient descent and stochastic gradient descent. In the case mini-batch gradient descent, the cost function J_{mb} is defined as follows:

$$J_{\text{mb}} = \frac{1}{\text{B}}\sum_{i=1}^{\text{B}}\mathcal{L}^{(i)} \tag{3.32}$$

where B is the number of examples in the mini-batch.

There is another optimization method called *momentum*. Consider mini-batch stochastic gradient. For any single layer ℓ, the update rule is as follows:

$$\begin{cases} v_{dW^{[\ell]}} = \beta v_{dW^{[\ell]}} + (1-\beta)\frac{\partial J}{\partial W^{[\ell]}} \\ W^{[\ell]} = W^{[\ell]} - \alpha v_{dW^{[\ell]}} \end{cases} \tag{3.33}$$

Notice how there are now two stages instead of a single stage. The weight update now depends on the cost J at this update step and the *velocity* $v_{dW^{[\ell]}}$. The relative importance is controlled by β. Consider the analogy to a human driving a car. While in motion, the car has momentum. If the car were to use the brakes (or not push accelerator throttle), the car would continue moving due to its momentum. Returning to optimization, the velocity $v_{dW^{[\ell]}}$ will keep track of the gradient over time. This technique has significantly helped neural networks during the training phase.

3.3 Analyzing the Parameters

At this point, we have initialized the parameters and have optimized the parameters. Suppose we evaluate the trained model and observe that it achieves 96% accuracy on the training set but only 64% on the testing set. Some solutions include: collecting more data, employing regularization, or making the model shallower. Let us briefly look at regularization techniques.

3.3.1 L2 Regularization

Let W below denote *all* the parameters in a model. In the case of neural networks, you may think of applying the 2nd term to all layer weights $W^{[\ell]}$. For convenience, we simply write W. The L2 regularization adds another term to the cost function:

$$J_{L2} = J + \frac{\lambda}{2}||W||^2 \tag{3.34}$$

$$= J + \frac{\lambda}{2}\sum_{ij}|W_{ij}|^2 \tag{3.35}$$

$$= J + \frac{\lambda}{2}W^T W \tag{3.36}$$

where J is the standard cost function from before, λ is an arbitrary value with a larger value indicating more regularization and W contains all the weight matrices, and where Equations (3.34), (3.35) and (3.36) are equivalent. The update rule with L2 regularization becomes:

$$W = W - \alpha\frac{\partial J}{\partial W} - \alpha\frac{\lambda}{2}\frac{\partial W^T W}{\partial W} \tag{3.37}$$

$$= (1 - \alpha\lambda)W - \alpha\frac{\partial J}{\partial W} \tag{3.38}$$

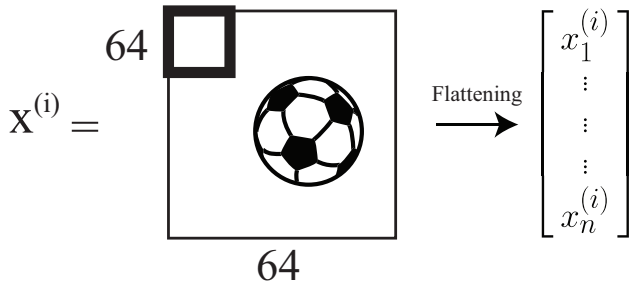
When we were updating our parameters using gradient descent, we did not have the $(1 - \alpha\lambda)W$ term. This means with L2 regularization, every update will include some penalization, depending on W. This penalization increases the cost J, which encourages individual parameters to be small in magnitude, which is a way to reduce overfitting.

3.3.2 Parameter Sharing

Recall logistic regression. It can be represented as a neural network, as shown in Figure 3. The parameter vector $\theta = (\theta_1, ..., \theta_n)$ must have the same number of elements as the input vector $x = (x_1, ..., x_n)$. In our image soccer ball example, this means θ_1 always looks at the top left pixel of the image no matter what. However, we know that a soccer ball might appear in any region of the image and not always the center. It is possible that θ_1 was never trained on a soccer ball in the top left of the image. As a result, during test time, if an image of a soccer ball in the top left appears, the logistic regression will likely predict *no soccer ball*. This is a problem.

This leads us to *convolutional neural networks*. Suppose θ is no longer a vector but instead is a matrix. For our soccer ball example, suppose $\theta = \mathbb{R}^{4\times4}$.

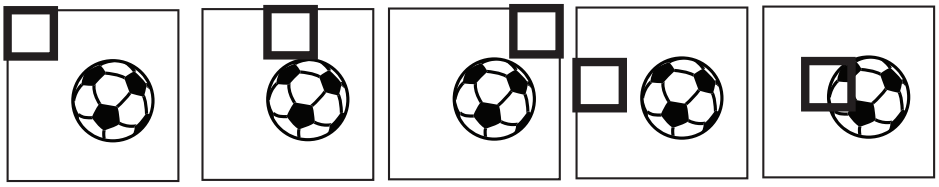
For simplicity, we show the image as 64×64 but recall it is actually three-



dimensional and contains 3 channels. We now take our matrix of parameters θ and slide it over the image. This is shown above by the thick square in the upper left of the image. To compute the activation a, we compute the element-wise product between θ and $x_{1:4,1:4}$, where the subscripts for x indicate we are taking the top left 4×4 region in the image x. We then collapse the matrix into a single scalar by summing all the elements resulting from the element-wise product. Formally:

$$a = \sum_{i=1}^{4} \sum_{j=1}^{4} \theta_{ij} x_{ij} \tag{3.39}$$

We then move this window slightly to the right in the image and repeat this process. Once we have reached the end of the row, we start at the beginning of the second row.



Once we have reached the end of the image, the parameters θ have "seen" all pixels of the image: θ_1 is no longer related to only the top left pixel. As a result, whether the soccer ball appears in the bottom right or top left of the image, the neural network will successfully detect the soccer ball.

CS229: Additional Notes on Backpropagation

1 Forward propagation

Recall that given input x, we define $a^{[0]} = x$. Then for layer $\ell = 1, 2, \ldots, N$, where N is the number of layers of the network, we have

1. $z^{[\ell]} = W^{[\ell]} a^{[\ell-1]} + b^{[\ell]}$
2. $a^{[\ell]} = g^{[\ell]}(z^{[\ell]})$

In these notes we assume the nonlinearities $g^{[\ell]}$ are the same for all layers besides layer N. This is because in the output layer we may be doing regression [hence we might use $g(x) = x$] or binary classification [$g(x) = \text{sigmoid}(x)$] or multiclass classification [$g(x) = \text{softmax}(x)$]. Hence we distinguish $g^{[N]}$ from g, and assume g is used for all layers besides layer N.

Finally, given the output of the network $a^{[N]}$, which we will more simply denote as $\hat{y}$, we measure the loss $J(W, b) = \mathcal{L}(a^{[N]}, y) = \mathcal{L}(\hat{y}, y)$. For example, for real-valued regression we might use the squared loss

$$\mathcal{L}(\hat{y}, y) = \frac{1}{2}(\hat{y} - y)^2$$

and for binary classification using logistic regression we use

$$\mathcal{L}(\hat{y}, y) = -(y \log \hat{y} + (1 - y) \log(1 - \hat{y}))$$

or negative log-likelihood. Finally, for softmax regression over k classes, we use the cross entropy loss

$$\mathcal{L}(\hat{y}, y) = -\sum_{j=1}^{k} \mathbf{1}\{y = j\} \log \hat{y}_j$$

which is simply negative log-likelihood extended to the multiclass setting. Note that $\hat{y}$ is a k-dimensional vector in this case. If we use y to instead denote the k-dimensional vector of zeros with a single 1 at the lth position, where the true label is l, we can also express the cross entropy loss as

$$\mathcal{L}(\hat{y}, y) = -\sum_{j=1}^{k} y_j \log \hat{y}_j$$

2 Backpropagation

Let's define one more piece of notation that'll be useful for backpropagation.[1] We will define

$$\delta^{[\ell]} = \nabla_{z^{[\ell]}} \mathcal{L}(\hat{y}, y)$$

We can then define a three-step "recipe" for computing the gradients with respect to every $W^{[\ell]}, b^{[\ell]}$ as follows:

1. For output layer N, we have

   $$\delta^{[N]} = \nabla_{z^{[N]}} \mathcal{L}(\hat{y}, y)$$

   Sometimes we may want to compute $\nabla_{z^{[N]}} \mathcal{L}(\hat{y}, y)$ directly (e.g. if $g^{[N]}$ is the softmax function), whereas other times (e.g. when $g^{[N]}$ is the sigmoid function σ) we can apply the chain rule:

   $$\nabla_{z^{[N]}} \mathcal{L}(\hat{y}, y) = \nabla_{\hat{y}} \mathcal{L}(\hat{y}, y) \circ (g^{[N]})'(z^{[N]})$$

   Note $(g^{[N]})'(z^{[N]})$ denotes the elementwise derivative w.r.t. $z^{[N]}$.

2. For $\ell = N-1, N-2, \ldots, 1$, we have

   $$\delta^{[\ell]} = (W^{[\ell+1]\top} \delta^{[\ell+1]}]) \circ g'(z^{[\ell]})$$

3. Finally, we can compute the gradients for layer ℓ as

   $$\nabla_{W^{[\ell]}} J(W, b) = \delta^{[\ell]} a^{[\ell-1]\top}$$
   $$\nabla_{b^{[\ell]}} J(W, b) = \delta^{[\ell]}$$

where we use $\circ$ to indicate the elementwise product. Note the above procedure is for a single training example.

You can try applying the above algorithm to logistic regression ($N = 1$, $g^{[1]}$ is the sigmoid function σ) to sanity check steps (1) and (3). Recall that $\sigma'(z) = \sigma(z) \circ (1 - \sigma(z))$ and $\sigma(z^{[1]})$ is simply $a^{[1]}$. Note that for logistic regression, if x is a column vector in $\mathbb{R}^{n \times 1}$, then $W^{[1]} \in \mathbb{R}^{1 \times n}$, and hence $\nabla_{W^{[1]}} J(W, b) \in \mathbb{R}^{1 \times n}$. Example code for two layers is also given at:

`http://cs229.stanford.edu/notes/backprop.py`

[1] These notes are closely adapted from:
`http://ufldl.stanford.edu/tutorial/supervised/MultiLayerNeuralNetworks/`
Scribe: Ziang Xie

CS229 Lecture notes

Raphael John Lamarre Townshend

Decision Trees

We now turn our attention to decision trees, a simple yet flexible class of algorithms. We will first consider the non-linear, region-based nature of decision trees, continue on to define and contrast region-based loss functions, and close off with an investigation of some of the specific advantages and disadvantages of such methods. Once finished with their nuts and bolts, we will move on to investigating different ensembling methods through the lens of decision trees, due to their suitability for such techniques.

1 Non-linearity

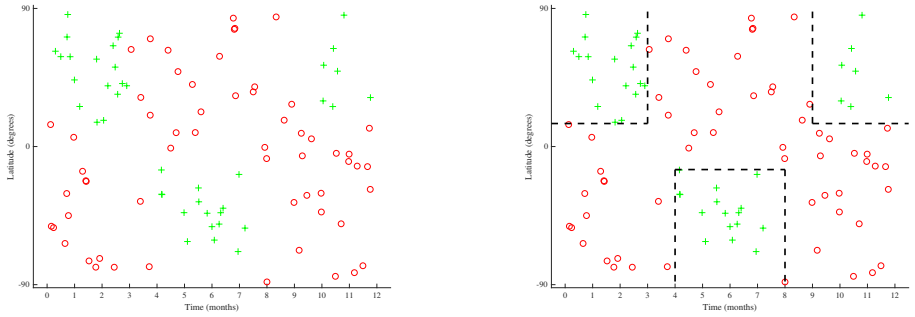
Importantly, decision trees are one of the first inherently **non-linear** machine learning techniques we will cover, as compared to methods such as vanilla SVMs or GLMs. Formally, a method is linear if for an input $x \in \mathbb{R}^n$ (with interecept term $x_0 = 1$) it only produces hypothesis functions h of the form:

$$h(x) = \theta^T x$$

where $\theta \in \mathbb{R}^n$. Hypothesis functions that cannot be reduced to the form above are called non-linear, and if a method can produce non-linear hypothesis functions then it is also non-linear. We have already seen that kernelization of a linear method is one such method by which we can achieve non-linear hypothesis functions, via a feature mapping $\phi(x)$.

Decision trees, on the other hand, can directly produce non-linear hypothesis functions without the need for first coming up with an appropriate feature mapping. As a motivating (and very Canadien) example, let us say we want to build a classifier that, given a time and a location, can predict whether or not it would be possible to ski nearby. To keep things simple, the time is represented as month of the year and the location is represented as

a latitude (how far North or South we are with $-90°$, $0°$, and $90°$ being the South Pole, Equator, and North Pole, respectively).



A representative dataset is shown above left. There is no linear boundary that would correctly split this dataset. However, we can recognize that there are different areas of positive and negative space we wish to isolate, one such division being shown above right. We accomplish this by partitioning the input space $\mathcal{X}$ into disjoint subsets (or **regions**) R_i:

$$\mathcal{X} = \bigcup_{i=0}^{n} R_i$$
$$\text{s.t.} \qquad R_i \cap R_j = \emptyset \text{ for } i \neq j$$

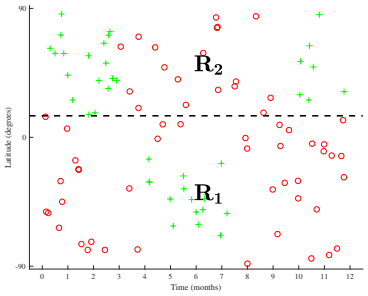
where $n \in \mathbb{Z}^+$.

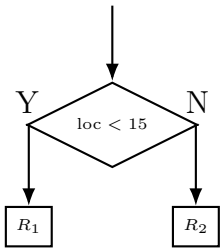
2 Selecting Regions

In general, selecting optimal regions is intractable. Decision trees generate an approximate solution via **greedy, top-down, recursive partitioning**. The method is **top-down** because we start with the original input space $\mathcal{X}$ and split it into two child regions by thresholding on a single feature. We then take one of these child regions and can partition via a new threshold. We continue the training of our model in a **recursive** manner, always selecting a leaf node, a feature, and a threshold to form a new split. Formally, given a parent region R_p, a feature index j, and a threshold $t \in \mathbb{R}$, we obtain two child regions R_1 and R_2 as follows:

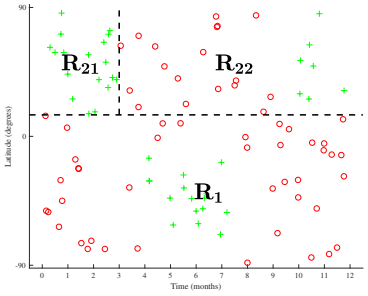
$$R_1 = \{X \mid X_j < t, X \in R_p\}$$
$$R_2 = \{X \mid X_j \geq t, X \in R_p\}$$

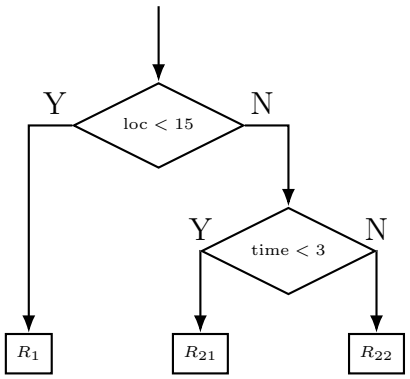
The beginning of one such process is shown below applied to the skiing dataset. In step a, we split the input space $\mathcal{X}$ by the location feature, with a threshold of 15, creating child regions R_1 and R_2. In step b, we then recursively select one of these child regions (in this case R_2) and select a feature (time) and threshold (3), generating two more child regions (R_{21} and R_{22}). In step c, we select any one of the remaining leaf nodes (R_1, R_{21}, R_{22}). We can continue in such a manner until we a meet a given stop criterion (more on this later), and then predict the majority class at each leaf node.



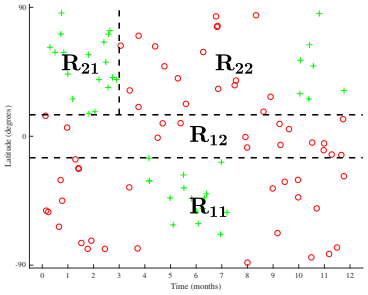


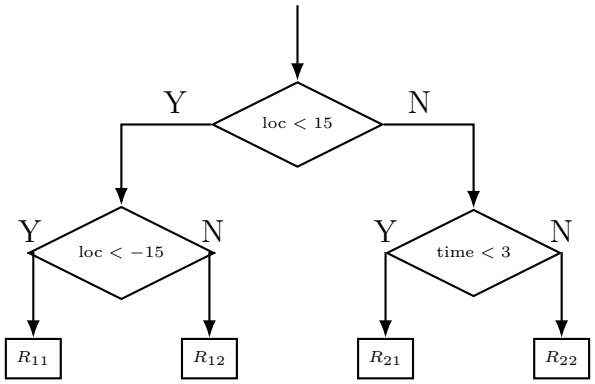
(a)





(b)





(c)

3 Defining a Loss Function

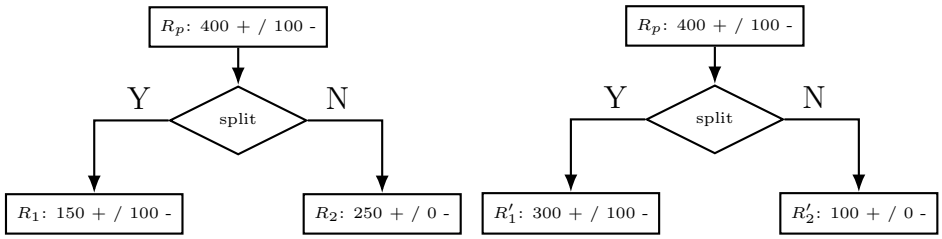
A natural question to ask at this point is how to choose our splits. To do so, it is first useful to define our loss L as a set function on a region R. Given a split of a parent R_p into two child regions R_1 and R_2, we can compute the loss of the parent $L(R_p)$ as well as the cardinality-weighted loss of the children $\frac{|R_1|L(R_1)+|R_2|L(R_2)}{|R_1|+|R_2|}$. Within our **greedy** partitioning framework, we want to select the leaf region, feature, and threshold that will maximize our decrease in loss:

$$L(R_p) - \frac{|R_1|L(R_1) + |R_2|L(R_2)}{|R_1| + |R_2|}$$

For a classification problem, we are interested in the **misclassification loss** $L_{misclass}$. For a region R let $\hat{p}_c$ be the proportion of examples in R that are of class c. Misclassification loss on R can be written as:

$$L_{misclass}(R) = 1 - \max_c(\hat{p}_c)$$

We can understand this as being the number of examples that would be misclassified if we predicted the majority class for region R (which is exactly what we do). While misclassification loss is the final value we are interested in, it is not very sensitive to changes in class probabilities. As a representative example, we show a binary classification case below. We explicitly depict the parent region R_p as well as the positive and negative counts in each region.



The first split is isolating out more of the positives, but we note that:

$$L(R_p) = \frac{|R_1|L(R_1) + |R_2|L(R_2)}{|R_1| + |R_2|} = \frac{|R'_1|L(R'_1) + |R'_2|L(R'_2)}{|R'_1 + |R'_2|} = 100$$

.

Thus, not only can we not only are the losses of the two splits identical, but neither of the splits decrease the loss over that of the parent.

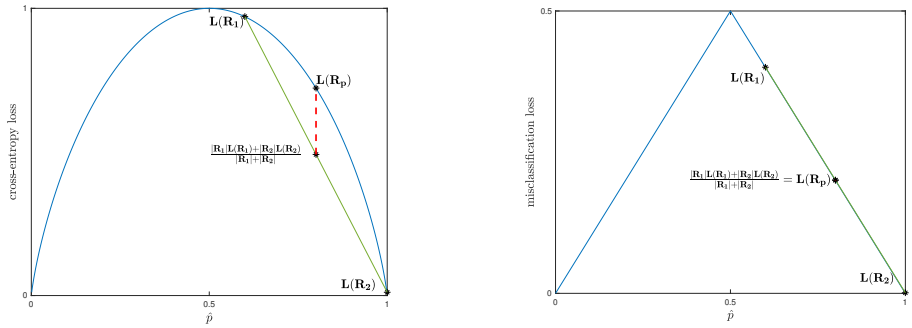
We therefore are interested in defining a more sensitive loss. While several have been proposed, we will focus here on the **cross-entropy** loss L_{cross}:

$$L_{cross}(R) = -\sum_c \hat{p}_c \log_2 \hat{p}_c$$

With $\hat{p} \log_2 \hat{p} \equiv 0$ if $\hat{p} = 0$. From an information-theoretic perspective, cross-entropy measure the number of bits needed to specify the outcome (or class) given that the distribution is known. Furthermore, the reduction in loss from parent to child is known as information gain.

To understand the relative sensitivity of cross-entropy loss with respect to misclassification loss, let us look at plots of both loss functions for the binary classification case. For these cases, we can simplify our loss functions to depend on just the proportion of positive examples $\hat{p}_i$ in a region R_i:

$$L_{misclass}(R) = L_{misclass}(\hat{p}) = 1 - \max(\hat{p}, 1 - \hat{p})$$
$$L_{cross}(R) = L_{cross}(\hat{p}) = -\hat{p} \log \hat{p} - (1 - \hat{p}) \log (1 - \hat{p})$$



In the figure above on the left, we see the cross-entropy loss plotted over $\hat{p}$. We take the regions (R_p, R_1, R_2) from the previous page's example's first split, and plot their losses as well. As cross-entropy loss is strictly concave, it can be seen from the plot (and easily proven) that as long as $\hat{p}_1 \neq \hat{p}_2$ and both child regions are non-empty, then the weighted sum of the children losses will always be less than that of the parent.

Misclassification loss, on the other hand, is not strictly concave, and therefore there is no guarantee that the weighted sum of the children will be less than that of the parent, as shown above right, with the same partition. Due to this added sensitivity, cross-entropy loss (or the closely related Gini loss) are used when growing decision trees for classification.

Before fully moving away from loss functions, we briefly cover the regression setting for decision trees. For each data point x_i we now instead have an

associated value $y_i \in \mathbb{R}$ we wish to predict. Much of the tree growth process remains the same, with the differences being that the final prediction for a region R is the mean of all the values:

$$\hat{y} = \frac{\sum_{i \in R} y_i}{|R|}$$

And in this case we can directly use the **squared loss** to select our splits:

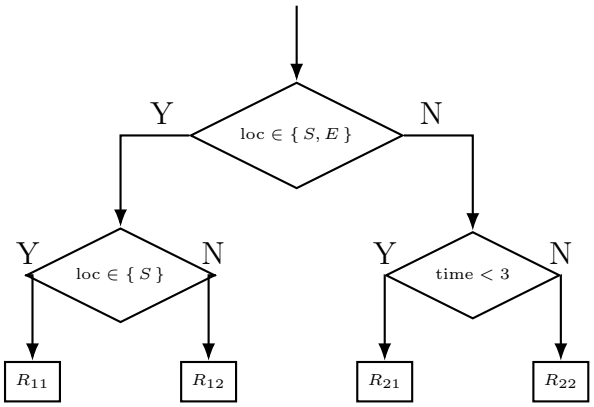
$$L_{squared}(R) = \frac{\sum_{i \in R} (y_i - \hat{y})^2}{|R|}$$

4 Other Considerations

The popularity of decision trees can in large part be attributed to the ease by which they are explained and understood, as well as the high degree of interpretability they exhibit: we can look at the generated set of thresholds to understand why a model made specific predictions. However, that is not the full picture – we will now cover some additional salient points.

4.1 Categorical Variables

Another advantage of decision trees is that they can easily deal with categorical variables. As an example, our location in the skiing dataset could instead be represented as a categorical variable (one of Northern Hemisphere, Southern Hemisphere, or Equator (i.e. loc $\in \{N, S, E\}$)). Rather than use a one-hot encoding or similar preprocessing step to transform the data into a quantitative feature, as would be necessary for the other algorithms we have seen, we can directly probe subset membership. The final tree in Section 2 can be re-written as:



A caveat to the above is that we must take care to not allow a variable to have too many categories. For a set of categories S, our set of possible questions is the power set $\mathcal{P}(S)$, of cardinality $2^{|S|}$. Thus, a large number of categories makes question selectioin computationally intractable. Optimizations are possible for the binary classification, though even in this case serious consideration should be given to whether the feature can be re-formulated as a quantitative one instead as the large number of possible thresholds lend themselves to a high degree of overfitting.

4.2 Regularization

In Section 2 we alluded to various stopping criteria we could use to determine when to halt the growth of a tree. The simplest criteria involves "fully" growning the tree: we continue until each leaf region contains exactly one training data point. This technique however leads to a high variance and low bias model, and we therefore turn to various stopping heuristics for regularization. Some common ones include:

- **Minimum Leaf Size** – Do not split R if its cardinality falls below a fixed threshold.
- **Maximum Depth** – Do not split R if more than a fixed threshold of splits were already taken to reach R.
- **Maximum Number of Nodes** – Stop if a tree has more than a fixed threshold of leaf nodes.

A tempting heuristic to use would be to enforce a minimum decrease in loss after splits. This is a problematic approach as the greedy, single-feature at a time approach of decision trees could mean missing higher order interactions. If we require thresholding on multiple features to achieve a good split, we might be unable to achieve a good decrease in loss on the initial splits and therefore prematurely terminate. A better approach involves fully growing out the tree, and then pruning away nodes that minimally decrease misclassification or squared error, as measured on a validation set.

4.3 Runtime

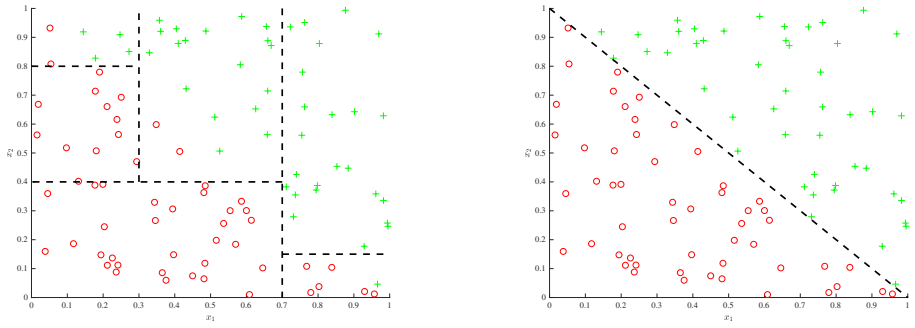
We briefly turn to considering the runtime of decision trees. For ease of analysis, we will consider binary classification with n examples, f features, and a tree of depth d. At test time, for a data point we traverse the tree

until we reach a leaf node and then output its prediction, for a runtime of $O(d)$. Note that if our tree is balanced than $d = O(\log n)$, and thus test time performance is generally quite fast.

At training time, we note that each data point can only appear in at most $O(d)$ nodes. Through sorting and intelligent caching of intermediate values, we can achieve an amortized runtime of $O(1)$ at each node for a single data point for a single feature. Thus, overall runtime is $O(nfd)$ – a fairly fast runtime as the data matrix alone is of size nf.

4.4 Lack of Additive Structure

One important downside to consider is that decision trees can not easily capture additive structure. For example, as seen below on the left, a simple decision boundary of the form $x_1 + x_2$ could only be approximately modeled through the use of many splits, as each split can only consider one of x_1 or x_2 at a time. A linear model on the other hand could directly derive this boundary, as shown below right.



While there has been some work in allowing for decision boundaries that factor in many features at once, they have the downside of further increasing variance and reducing interpretability.

5 Recap

To summarize, some of the primary benefits of decision trees are:

+ Easy to explain
+ Interpretable
+ Categorical variable support
+ Fast

While some of the disadvantages include:

– High variance
– Poor additive modeling

Unfortunately, these problems tend to cause individual decision trees to have low overall predictive accuracy. A common (and successful) way to address these issues is through ensembling methods – our next topic of discussion.

CS229 Lecture notes

Raphael John Lamarre Townshend

Ensembling Methods

We now cover methods by which we can aggregate the output of trained models. We will use Bias-Variance analysis as well as the example of decision trees to probe some of the trade-offs of each of these methods.

To understand why we can derive benefit from ensembling, let us first recall some basic probability theory. Say we have n independent, identically distributed (i.i.d.) random variables X_i for $0 \leq i < n$. Assume $\text{Var}(X_i) = \sigma^2$ for all X_i. Then we have that the variance of the mean is:

$$\text{Var}(\bar{X}) = \text{Var}(\frac{1}{n}\sum_i X_i) = \frac{\sigma^2}{n}$$

Now, if we drop the independence assumption (so the variables are only i.d.), and instead say that the X_i's are correlated by a factor ρ, we can show that:

$$\begin{aligned}
\text{Var}(\bar{X}) =& \text{Var}(\frac{1}{n}\sum_i X_i) && (1)\\
=& \frac{1}{n^2}\sum_{i,j}\text{Cov}(X_i, X_j) && (2)\\
=& \frac{n\sigma^2}{n^2} + \frac{n(n-1)\rho\sigma^2}{n^2} && (3)\\
=& \rho\sigma^2 + \frac{1-\rho}{n}\sigma^2 && (4)
\end{aligned}$$

Where in Step 3 we use the definition of pearson correlation coefficient $\rho_{X,Y} = \frac{\text{Cov}(X,Y)}{\sigma_x\sigma_y}$ and that $\text{Cov}(X, X) = \text{Var}(X)$.

Now, if we consider each random variable to be the error of a given model, we can see that both increasing the number of models used (causing the

second term to vanish) as well as decreasing the correlation between models (causing the first term to vanish and returning us to the i.i.d. definition) leads to an overall decrease in variance of the error of the ensemble.

There are several ways by which we can generate de-correlated models, including:

- Using different algorithms
- Using different training sets
- Bagging
- Boosting

While the first two are fairly straightforward, they involve large amounts of additional work. In the following sections, we will cover the latter two techniques, boosting and bagging, as well as their specific uses in the context of decision trees.

1 Bagging

1.1 Boostrap

Bagging stands for "Boostrap Aggregation" and is a **variance reduction** ensembling method. **Bootstrap** is a method from statistics traditionally used to measure uncertainty of some estimator (e.g. mean).

Say we have a true population P that we wish to compute an estimator for, as well a training set S sampled from P ($S \sim P$). While we can find an approximation by computing the estimator on S, we cannot know what the error is with respect to the true value. To do so we would need multiple independent training sets S_1, S_2, ... all sampled from P.

However, if we make the assumption that $S = P$, we can generate a new bootstrap set Z sampled with replacement from S ($Z \sim S$, $|Z| = |S|$). In fact we can generate many such samples $Z_1, Z_2, ..., Z_M$. We can then look at the variability of our estimate across these bootstrap sets to obtain a measure of error.

1.2 Aggregation

Now, returning to ensembling, we can take each Z_m and train a machine learning model G_m on each, and define a new **aggregate predictor**:

$$G(X) = \sum_m \frac{G_m(x)}{M}$$

This process is called **bagging**. Referring back to equation (4), we have that the variance of M correlated predictors is:

$$Var(\bar{X}) = \rho\sigma^2 + \frac{1-\rho}{M}\sigma^2$$

Bagging creates less correlated predictors than if they were all simply trained on S, thereby decreasing ρ. While the bias of each individual predictor increases due to each bootstrap set not having the full training set available, in practice it has been found that the decrease in variance outweighs the increase in bias. Also note that increasing the number of predictors M can't lead to additional overfitting, as ρ is insensitive to M and therefore overall variance can only decrease.

An additional advantage of bagging is called **out-of-bag estimation**. It can be shown that each bootstrapped sample only contains approximately $\frac{2}{3}$ of S, and thus we can use the other $\frac{1}{3}$ as an estimate of error, called out-of-bag error. In the limit, as $M \to \infty$, out-of-bag error gives an equivalent result to leave-one-out cross-validation.

1.3 Bagging + Decision Trees

Recall that fully-grown decision trees are high variance, low bias models, and therefore the variance-reducing effects of bagging work well in conjunction with them. Bagging also allows for handling of missing features: if a feature is missing, exclude trees in the ensemble that use that feature in our of their splits. Though if certain features are particularly powerful predictors they may still be included in most if not all trees.

A downside to bagged trees is that we lose the interpretability inherent in the single decision tree. One method by which to re-gain some amount of insight is through a technique called **variable importance measure**. For each feature, find each split that uses it in the ensemble and average the decrease in loss across all such splits. Note that this is not the same as measuring how much performance would degrade if we did not have this feature, as other features might be correlated and could substitute.

A final but important aspect of bagged decision trees to cover is the method of **random forests**. If our dataset contained one very strong predictor, then our bagged trees would always use that feature in their splits and end up correlated. With random forests, we instead only allow a subset

of features to be used at each split. By doing so, we achieve a decrease in correlation ρ which leads to a decrease in variance. Again, there is also an increase in bias due to the restriction of the feature space, but as with vanilla bagged decision trees this proves to not often be an issue. Finally, even powerful predictors will no longer be present in every tree (assuming sufficient number of trees and sufficient restriction of features at each split), allowing for more graceful handling of missing predictors.

1.4 Recap

To summarize, some of the primary benefits of bagging, in the context of decision trees, are:

+ Decrease in variance (even more so for random forests)
+ Better accuracy
+ Free validation set
+ Support for missing values

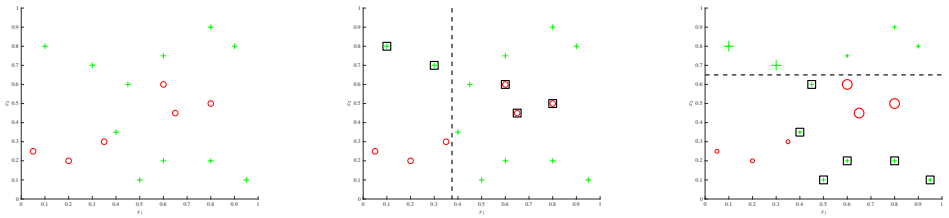
While some of the disadvantages include:

– Incrase in bias (even more so for random forests)
– Harder to interpret
– Still not additive
– More expensive

2 Boosting

2.1 Intuition

Bagging is a variance-reducing technique, whereas boosting is used for **bias-reduction**. We therefore want high bias, low variance models, also known as **weak learners**. Continuing our exploration via the use of decision trees, we can make them into weak learners by allowing each tree to only make one decision before making a prediction; these are known as **decision stumps**.



We explore the intuition behind boosting via the example above. We start with a dataset on the left, and allow a single decision stump to be trained, as seen in the middle panel. The key idea is that we then track which examples the classifier got wrong, and increase their relative weight compared to the correctly classified examples. We then train a new decision stump which will be more incentivized to correctly classify these "hard negatives." We continue as such, incrementally re-weighting examples at each step, and at the end we output a combination of these weak learners as an ensemble classifier.

2.2 Adaboost

Having covered the intuition, let us look at one of the most popular boosting algorithms, **Adaboost**, reproduced below:

Algorithm 0: Adaboost

Input: Labeled training data (x_1, y_1), (x_2, y_2), ... (x_N, y_N)

Output: Ensemble classifer $f(x)$

$w_i \leftarrow \frac{1}{N}$ for $i = 1, 2..., N$

for $m = 0$ ***to*** M **do**

  Fit weak classifier G_m to training data weighted by w_i

  Compute weighted error $err_m = \frac{\sum_i w_i \mathbb{1}(y_i \neq G_m(x_i))}{\sum w_i}$

  Compute weight $\alpha_m = \log(\frac{1-err_m}{err_m})$

  $w_i \leftarrow w_i * \exp(\alpha_m \mathbb{1}(y_i \neq G_m(x_i)))$

end

$f(x) = \text{sign}(\sum_m \alpha_m G_m(x))$

The weightings for each example begin out even, with misclassified examples being further up-weighted at each step, in a cumulative fashion. The final aggregate classifier is a summation of all the weak learners, weighted by the negative log-odds of the weighted error.

We can also see that due to the final summation, this ensembling method allows for modeling of additive terms, increasing the overall modeling capability (and variance) of the final model. Each new weak learner is no longer

independent of the previous models in the sequence, meaning that increasing M leads to an increase in the risk of overfitting.

The exact weightings used for Adaboost appear to be somewhat arbitrary at first glance, but can be shown to be well justified. We shall approach this in the next section through a more general framework of which Adaboost is a special case.

2.3 Forward Stagewise Additive Modeling

The **Forward Stagewise Additive Modeling** algorithm reproduced below is a framework for ensembling :

Algorithm 1: Forward Stagewise Additive Modeling

Input: Labeled training data (x_1, y_1), (x_2, y_2), ... (x_N, y_N)
Output: Ensemble classifer $f(x)$

Initialize $f_0(x) = 0$
for $m = 0$ ***to*** M **do**
  Compute $(\beta_m, \gamma_m) = \text{argmin}_{\beta,\gamma} \sum_{i=1}^{N} L(y_i, f_{m-1}(x_i) + \beta G(x_i; \gamma))$
  Set $f_m(x) = f_{m-1}(x) + \beta_m G(x; y_i)$
end
$f(x) = f_m(x)$

Close inspection reveals that few assumptions are made about the learning problem at hand, the only major ones being the additive nature of the ensembling as well as the fixing of all previous weightings and parameters after a given step. We again have weak classifiers $G(x)$, though this time we explicitly parameterize them by their parameters γ. At each step we are trying to find the next weak learner's parameters and weighting so to best match the remaining error of the current ensemble.

As a concrete implementation of this algorithm, using a squared loss would be the same as fitting individual classifiers to the residual $y_i - f_{m-1}(x_i)$. Furthermore, it can be shown that Adaboost is a special case of this formulation, specifically for 2-class classification and exponential loss:

$$L(y, \hat{y}) = \exp(-y\hat{y})$$

For further details regarding the connection between Adaboost and Forward Stagewise Additive Modeling, the interested reader is referred to 10.4 Elements of Statistical Learning.

2.4 Gradient Boosting

In general, it is not always easy to write out a closed-form solution to the minimization problem presented in Forward Stagewise Additive Modeling. High-performing methods such as **xgboost** resolve this issue by turning to numerical optimization.

One of the most obvious things to do in this case would be to take the derivative of the loss and perform gradient descent. However, the complication is that we are restricted to taking steps in our model class – we can only add in parameterized weak learners $G(x, \gamma)$, not make arbitrary moves in the input space.

In **gradient boosting**, we instead compute the gradient at each training point with respect to the current predictor (typically a decision stump):

$$g_i = \frac{\partial L(y, f(x_i))}{\partial f(x_i)}$$

We then train a new regression predictor to match this gradient and use it as the gradient step. In Forward Stagewise Additive Modeling, this works out to:

$$\gamma_i = \text{argmin}_\gamma \sum_{i=1}^{N} (g_i - G(x_i; \gamma))^2$$

2.5 Recap

To summarize, some of the primary benefits of boosting are:

+ Decrease in bias

+ Better accuracy

+ Additive modeling

While some of the disadvantages include:

– Increase in variance

– Prone to overfitting

For more on the theory behind boosting, John Duchi's excellent supplemental lecture notes are recommended.

CS229 Supplemental Lecture notes

John Duchi

1 Boosting

We have seen so far how to solve classification (and other) problems when we have a data representation already chosen. We now talk about a procedure, known as *boosting*, which was originally discovered by Rob Schapire, and further developed by Schapire and Yoav Freund, that automatically chooses feature representations. We take an optimization-based perspective, which is somewhat different from the original interpretation and justification of Freund and Schapire, but which lends itself to our approach of (1) choose a representation, (2) choose a loss, and (3) minimize the loss.

Before formulating the problem, we give a little intuition for what we are going to do. Roughly, the idea of boosting is to take a *weak learning* algorithm—any learning algorithm that gives a classifier that is slightly better than random—and transforms it into a *strong* classifier, which does much much better than random. To build a bit of intuition for what this means, consider a hypothetical digit recognition experiment, where we wish to distinguish 0s from 1s, and we receive images we must classify. Then a natural weak learner might be to take the middle pixel of the image, and if it is colored, call the image a 1, and if it is blank, call the image a 0. This classifier may be far from perfect, but it is likely better than random. Boosting procedures proceed by taking a collection of such weak classifiers, and then reweighting their contributions to form a classifier with much better accuracy than any individual classifier.

With that in mind, let us formulate the problem. Our interpretation of boosting is as a coordinate descent method in an infinite dimensional space, which—while it sounds complex—is not so bad as it seems. First, we assume we have raw input examples $x \in \mathbb{R}^n$ with labels $y \in \{-1, 1\}$, as is usual in binary classification. We also assume we have an infinite collection of *feature* functions $\phi_j : \mathbb{R}^n \to \{-1, 1\}$ and an infinite vector $\theta = [\theta_1 \ \theta_2 \ \cdots]^T$, but

which we assume always has only a finite number of non-zero entries. For our classifier we use

$$h_\theta(x) = \text{sign}\left(\sum_{j=1}^{\infty} \theta_j \phi_j(x)\right).$$

We will abuse notation, and define $\theta^T \phi(x) = \sum_{j=1}^{\infty} \theta_j \phi_j(x)$.

In boosting, one usually calls the features ϕ_j *weak hypotheses*. Given a training set $(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})$, we call a vector $p = (p^{(1)}, \ldots, p^{(m)})$ a distribution on the examples if $p^{(i)} \geq 0$ for all i and

$$\sum_{i=1}^{m} p^{(i)} = 1.$$

Then we say that there is a *weak learner with margin* $\gamma > 0$ if for any distribution p on the m training examples there exists one weak hypothesis ϕ_j such that

$$\sum_{i=1}^{m} p^{(i)} 1\left\{y^{(i)} \neq \phi_j(x^{(i)})\right\} \leq \frac{1}{2} - \gamma. \tag{1}$$

That is, we assume that there is *some* classifier that does slightly better than random guessing on the dataset. The existence of a weak learning algorithm is an assumption, but the surprising thing is that we can transform any weak learning algorithm into one with perfect accuracy.

In more generality, we assume we have access to a *weak learner*, which is an algorithm that takes as input a distribution (weights) p on the training examples and returns a classifier doing slightly better than random. We will

(i) **Input:** A distribution $p^{(1)}, \ldots, p^{(m)}$ and training set $\{(x^{(i)}, y^{(i)})\}_{i=1}^m$ with $\sum_{i=1}^m p^{(i)} = 1$ and $p^{(i)} \geq 0$

(ii) **Return:** A weak classifier $\phi_j : \mathbb{R}^n \to \{-1, 1\}$ such that

$$\sum_{i=1}^{m} p^{(i)} 1\left\{y^{(i)} \neq \phi_j(x^{(i)})\right\} \leq \frac{1}{2} - \gamma.$$

Figure 1: Weak learning algorithm

show how, given access to a weak learning algorithm, boosting can return a classifier with perfect accuracy on the training data. (Admittedly, we would like the classifer to generalize well to unseen data, but for now, we ignore this issue.)

1.1 The boosting algorithm

Roughly, boosting begins by assigning each training example equal weight in the dataset. It then receives a weak-hypothesis that does well according to the current weights on training examples, which it incorporates into its current classification model. It then reweights the training examples so that examples on which it makes mistakes receive higher weight—so that the weak learning algorithm focuses on a classifier doing well on those examples—while examples with no mistakes receive lower weight. This repeated reweighting of the training data coupled with a weak learner doing well on examples for which the classifier currently does poorly yields classifiers with good performance.

The boosting algorithm specifically performs *coordinate descent* on the exponential loss for classification problems, where the objective is

$$J(\theta) = \frac{1}{m} \sum_{i=1}^{m} \exp(-y^{(i)} \theta^T \phi(x^{(i)})).$$

We first show how to compute the exact form of the coordinate descent update for the risk $J(\theta)$. Coordinate descent iterates as follows:

(i) Choose a coordinate $j \in \mathbb{N}$

(ii) Update θ_j to

$$\theta_j = \arg\min_{\theta_j} J(\theta)$$

while leaving θ_k identical for all $k \neq j$.

We iterate the above procedure until convergence.

In the case of boosting, the coordinate updates are not too challenging to derive because of the analytic convenience of the exp function. We now show how to derive the update. Suppose we wish to update coordinate k. Define

$$w^{(i)} = \exp\left(-y^{(i)} \sum_{j \neq k} \theta_j \phi_j(x^{(i)})\right)$$

to be a weight, and note that optimizing coordinate k corresponds to minimizing

$$\sum_{i=1}^{m} w^{(i)} \exp(-y^{(i)}\phi_k(x^{(i)})\alpha)$$

in $\alpha = \theta_k$. Now, define

$$W^+ := \sum_{i:y^{(i)}\phi_k(x^{(i)})=1} w^{(i)} \quad \text{and} \quad W^- := \sum_{i:y^{(i)}\phi_k(x^{(i)})=-1} w^{(i)}$$

to be the sums of the weights of examples that ϕ_k classifies correctly and incorrectly, respectively. Then finding θ_k is the same as choosing

$$\alpha = \arg\min_{\alpha} \left\{W^+ e^{-\alpha} + W^- e^{\alpha}\right\} = \frac{1}{2} \log \frac{W^+}{W^-}.$$

To see the final equality, take derivatives and set the resulting equation to zero, so we have $-W^+ e^{-\alpha} + W^- e^{\alpha} = 0$. That is, $W^- e^{2\alpha} = W^+$, or $\alpha = \frac{1}{2} \log \frac{W^+}{W^-}$.

What remains is to choose the particular coordinate to perform coordinate descent on. We assume we have access to a weak-learning algorithm as in Figure 1, which at iteration t takes as input a distribution p on the training set and returns a weak hypothesis ϕ_t satisfying the margin condition (1). We present the full boosting algorithm in Figure 2. It proceeds in iterations $t = 1, 2, 3, \ldots$. We represent the set of hypotheses returned by the weak learning algorithm at time t by $\{\phi_1, \ldots, \phi_t\}$.

2 The convergence of Boosting

We now argue that the boosting procedure achieves 0 training error, and we also provide a rate of convergence to zero. To do so, we present a lemma that guarantees progress is made.

Lemma 2.1. *Let*

$$J(\theta^{(t)}) = \frac{1}{m} \sum_{i=1}^{m} \exp\left(- y^{(i)} \sum_{\tau=1}^{t} \theta_\tau \phi_\tau(x^{(i)}) \right).$$

Then

$$J(\theta^{(t)}) \leq \sqrt{1 - 4\gamma^2} J(\theta^{(t-1)}).$$

For each iteration $t = 1, 2, \ldots$:

(i) Define weights

$$w^{(i)} = \exp\left(- y^{(i)} \sum_{\tau=1}^{t-1} \theta_\tau \phi_\tau(x^{(i)}) \right)$$

and distribution $p^{(i)} = w^{(i)} / \sum_{j=1}^m w^{(j)}$

(ii) Construct a weak hypothesis $\phi_t : \mathbb{R}^n \to \{-1, 1\}$ from the distribution $p = (p^{(1)}, \ldots, p^{(m)})$ on the training set

(iii) Compute $W_t^+ = \sum_{i:y^{(i)}\phi_t(x^{(i)})=1} w^{(i)}$ and $W_t^- = \sum_{i:y^{(i)}\phi_t(x^{(i)})=-1} w^{(i)}$ and set

$$\theta_t = \frac{1}{2} \log \frac{W_t^+}{W_t^-}.$$

Figure 2: Boosting algorithm

As the proof of the lemma is somewhat involved and not the central focus of these notes—though it is important to know one's algorithm will converge!—we defer the proof to Appendix A.1. Let us describe how it guarantees convergence of the boosting procedure to a classifier with zero training error.

We initialize the procedure at $\theta^{(0)} = \vec{0}$, so that the initial empirical risk $J(\theta^{(0)}) = 1$. Now, we note that for any θ, the misclassification error satisfies

$$1\left\{\text{sign}(\theta^T \phi(x)) \neq y\right\} = 1\left\{y\theta^T\phi(x) \leq 0\right\} \leq \exp\left(-y\theta^T\phi(x)\right)$$

because $e^z \geq 1$ for all $z \geq 0$. Thus, we have that the misclassification error rate has upper bound

$$\frac{1}{m} \sum_{i=1}^m 1\left\{\text{sign}(\theta^T\phi(x^{(i)})) \neq y^{(i)}\right\} \leq J(\theta),$$

and so if $J(\theta) < \frac{1}{m}$ then the vector θ makes *no* mistakes on the training data. After t iterations of boosting, we find that the empirical risk satisfies

$$J(\theta^{(t)}) \leq (1 - 4\gamma^2)^{\frac{t}{2}} J(\theta^{(0)}) = (1 - 4\gamma^2)^{\frac{t}{2}}.$$

To find how many iterations are required to guarantee $J(\theta^{(t)}) < \frac{1}{m}$, we take logarithms to find that $J(\theta^{(t)}) < 1/m$ if

$$\frac{t}{2}\log(1-4\gamma^2) < \log\frac{1}{m}, \quad \text{or} \quad t > \frac{2\log m}{-\log(1-4\gamma^2)}.$$

Using a first order Taylor expansion, that is, that $\log(1-4\gamma^2) \le -4\gamma^2$, we see that if the number of rounds of boosting—the number of weak classifiers we use—satisfies

$$t > \frac{\log m}{2\gamma^2} \ge \frac{2\log m}{-\log(1-4\gamma^2)},$$

then $J(\theta^{(t)}) < \frac{1}{m}$.

3 Implementing weak-learners

One of the major advantages of boosting algorithms is that they automatically generate features from raw data for us. Moreover, because the weak hypotheses always return values in $\{-1, 1\}$, there is no need to normalize features to have similar scales when using learning algorithms, which in practice can make a large difference. Additionally, and while this is not theoretically well-understood, many types of weak-learning procedures introduce non-linearities intelligently into our classifiers, which can yield much more expressive models than the simpler linear models of the form $\theta^T x$ that we have seen so far.

3.1 Decision stumps

There are a number of strategies for weak learners, and here we focus on one, known as *decision stumps*. For concreteness in this description, let us suppose that the input variables $x \in \mathbb{R}^n$ are real-valued. A decision stump is a function f, which is parameterized by a threshold s and index $j \in \{1, 2, \ldots, n\}$, and returns

$$\phi_{j,s}(x) = \text{sign}(x_j - s) = \begin{cases} 1 & \text{if } x_j \ge s \\ -1 & \text{otherwise.} \end{cases} \tag{2}$$

These classifiers are simple enough that we can fit them efficiently even to a weighted dataset, as we now describe.

Indeed, a decision stump weak learner proceeds as follows. We begin with a distribution—set of weights $p^{(1)}, \ldots, p^{(m)}$ summing to 1—on the training set, and we wish to choose a decision stump of the form (2) to minimize the error on the training set. That is, we wish to find a threshold $s \in \mathbb{R}$ and index j such that

$$\widehat{\mathrm{Err}}(\phi_{j,s}, p) = \sum_{i=1}^{m} p^{(i)} 1\left\{\phi_{j,s}(x^{(i)}) \neq y^{(i)}\right\} = \sum_{i=1}^{m} p^{(i)} 1\left\{y^{(i)}(x_j^{(i)} - s) \leq 0\right\} \quad (3)$$

is minimized. Naively, this could be an inefficient calculation, but a more intelligent procedure allows us to solve this problem in roughly $O(nm \log m)$ time. For each feature $j = 1, 2, \ldots, n$, we sort the raw input features so that

$$x_j^{(i_1)} \geq x_j^{(i_2)} \geq \cdots \geq x_j^{(i_m)}.$$

As the only values s for which the error of the decision stump can change are the values $x_j^{(i)}$, a bit of clever book-keeping allows us to compute

$$\sum_{i=1}^{m} p^{(i)} 1\left\{y^{(i)}(x_j^{(i)} - s) \leq 0\right\} = \sum_{k=1}^{m} p^{(i_k)} 1\left\{y^{(i_k)}(x_j^{(i_k)} - s) \leq 0\right\}$$

efficiently by incrementally modifying the sum in sorted order, which takes time $O(m)$ after we have already sorted the values $x_j^{(i)}$. (We do not describe the algorithm in detail here, leaving that to the interested reader.) Thus, performing this calcuation for each of the n input features takes total time $O(nm \log m)$, and we may choose the index j and threshold s that give the best decision stump for the error (3).

One *very* important issue to note is that by flipping the sign of the thresholded decision stump $\phi_{j,s}$, we achieve error $1 - \widehat{\mathrm{Err}}(\phi_{j,s}, p)$, that is, the error of

$$\widehat{\mathrm{Err}}(-\phi_{j,s}, p) = 1 - \widehat{\mathrm{Err}}(\phi_{j,s}, p).$$

(You should convince yourself that this is true.) Thus, it is important to also track the smallest value of $1 - \widehat{\mathrm{Err}}(\phi_{j,s}, p)$ over all thresholds, because this may be smaller than $\widehat{\mathrm{Err}}(\phi_{j,s}, p)$, which gives a better weak learner. Using this procedure for our weak learner (Fig. 1) gives the basic, but extremely useful, boosting classifier.

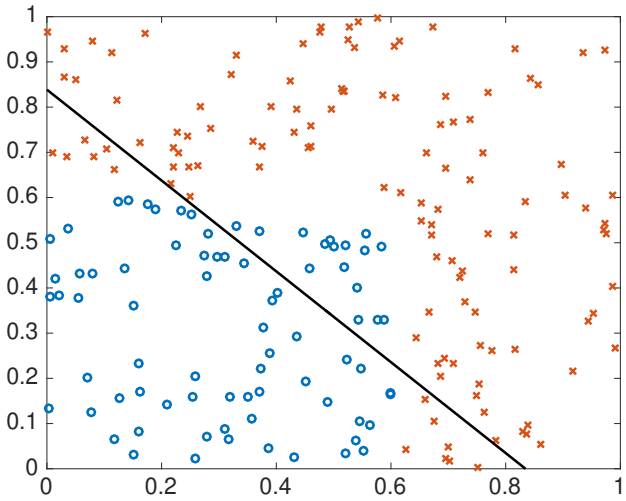


Figure 3: Best logistic regression classifier using the raw features $x \in \mathbb{R}^2$ (and a bias term $x_0 = 1$) for the example considered here.

3.2 Example

We now give an example showing the behavior of boosting on a simple dataset. In particular, we consider a problem with data points $x \in \mathbb{R}^2$, where the optimal classifier is

$$y = \begin{cases} 1 & \text{if } x_1 < .6 \text{ and } x_2 < .6 \\ -1 & \text{otherwise.} \end{cases} \tag{4}$$

This is a simple non-linear decision rule, but it is impossible for standard linear classifiers, such as logistic regression, to learn. In Figure 3, we show the best decision line that logistic regression learns, where positive examples are circles and negative examples are x's. It is clear that logistic regression is not fitting the data particularly well.

With boosted decision stumps, however, we can achieve a much better fit for the simple nonlinear classification problem (4). Figure 4 shows the boosted classifiers we have learned after different numbers of iterations of boosting, using a training set of size $m = 150$. From the figure, we see that the first decision stump is to threshold the feature x_1 at the value $s \approx .23$, that is, $\phi(x) = \text{sign}(x_1 - s)$ for $s \approx .23$.

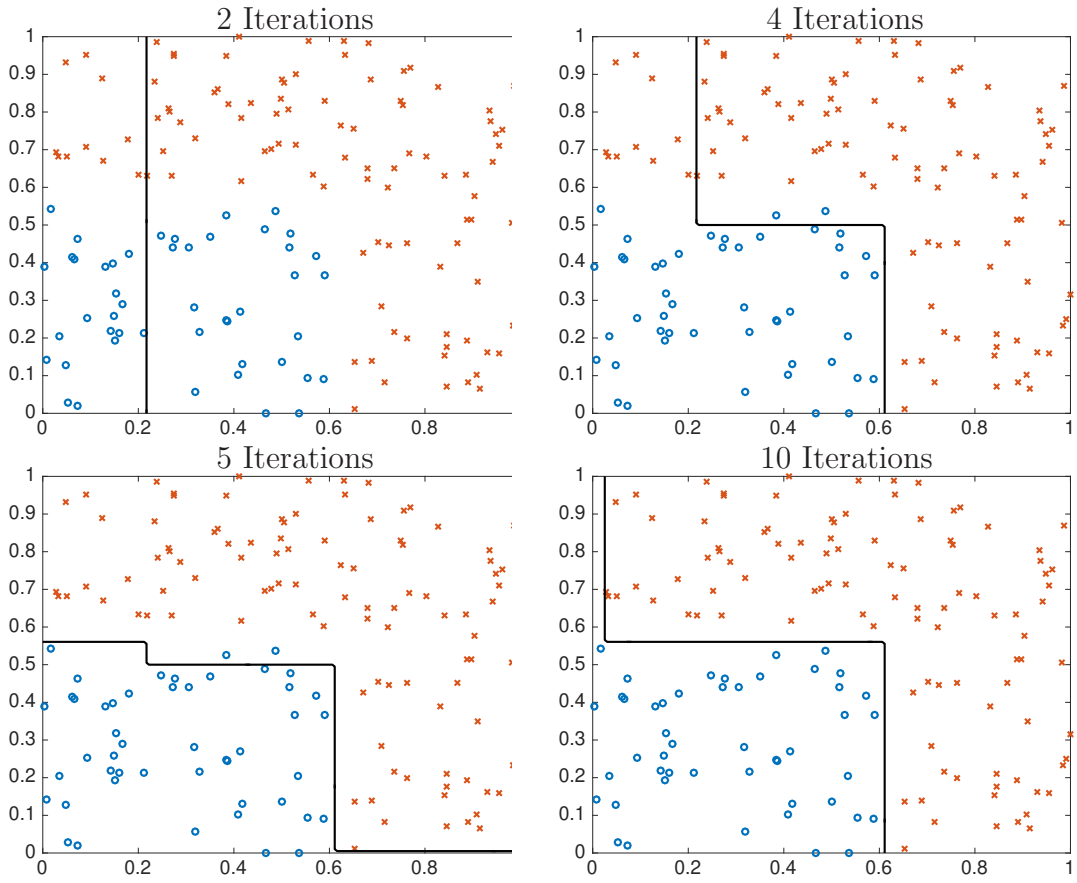


Figure 4: Boosted decision stumps after $t = 2, 4, 5$, and 10 iterations of boosting, respectively.

3.3 Other strategies

There are a huge number of variations on the basic boosted decision stumps idea. First, we do not require that the input features x_j be real-valued. Some of them may be categorical, meaning that $x_j \in \{1, 2, \ldots, k\}$ for some k, in which case natural decision stumps are of the form

$$\phi_j(x) = \begin{cases} 1 & \text{if } x_j = l \\ -1 & \text{otherwise,} \end{cases}$$

as well as variants setting $\phi_j(x) = 1$ if $x_j \in C$ for some set $C \subset \{1, \ldots, k\}$ of categories.

Another natural variation is the *boosted decision tree*, in which instead of a single level decision for the weak learners, we consider conjuctions of features or trees of decisions. Google can help you find examples and information on these types of problems.

A Appendices

A.1 Proof of Lemma 2.1

We now return to prove the progress lemma. We prove this result by directly showing the relationship of the weights at time t to those at time $t-1$. In particular, we note by inspection that

$$J(\theta^{(t)}) = \min_{\alpha} \{W_t^+ e^{-\alpha} + W_t^- e^{\alpha}\} = 2\sqrt{W_t^+ W_t^-}$$

while

$$J(\theta^{(t-1)}) = \frac{1}{m} \sum_{i=1}^{m} \exp\left(- y^{(i)} \sum_{\tau=1}^{t-1} \theta_\tau \phi_\tau(x^{(i)}) \right) = W_t^+ + W_t^-.$$

We know by the weak-learning assumption that

$$\sum_{i=1}^{m} p^{(i)} 1\left\{y^{(i)} \neq \phi_t(x^{(i)})\right\} \leq \frac{1}{2} - \gamma, \quad \text{or} \quad \frac{1}{W_t^+ + W_t^-} \underbrace{\sum_{i : y^{(i)} \phi_t(x^{(i)}) = -1} w^{(i)}}_{= W_t^-} \leq \frac{1}{2} - \gamma.$$

Rewriting this expression by noting that the sum on the right is nothing but W_t^-, we have

$$W_t^- \leq \left(\frac{1}{2} - \gamma\right)(W_t^+ + W_t^-), \quad \text{or} \quad W_t^+ \geq \frac{1+2\gamma}{1-2\gamma} W_t^-.$$

By substituting $\alpha = \frac{1}{2}\log\frac{1+2\gamma}{1-2\gamma}$ in the minimum defining $J(\theta^{(t)})$, we obtain

$$\begin{aligned}
J(\theta^{(t)}) &\leq W_t^+ \sqrt{\frac{1-2\gamma}{1+2\gamma}} + W_t^- \sqrt{\frac{1+2\gamma}{1-2\gamma}} \\
&= W_t^+ \sqrt{\frac{1-2\gamma}{1+2\gamma}} + W_t^-(1-2\gamma+2\gamma)\sqrt{\frac{1+2\gamma}{1-2\gamma}} \\
&\leq W_t^+ \sqrt{\frac{1-2\gamma}{1+2\gamma}} + W_t^-(1-2\gamma)\sqrt{\frac{1+2\gamma}{1-2\gamma}} + 2\gamma\frac{1-2\gamma}{1+2\gamma}\sqrt{\frac{1+2\gamma}{1-2\gamma}} W_t^+ \\
&= W_t^+ \left[\sqrt{\frac{1-2\gamma}{1+2\gamma}} + 2\gamma\sqrt{\frac{1-2\gamma}{1+2\gamma}}\right] + W_t^- \sqrt{1-4\gamma^2},
\end{aligned}$$

where we used that $W_t^- \leq \frac{1-2\gamma}{1+2\gamma} W_t^+$. Performing a few algebraic manipulations, we see that the final expression is equal to

$$\sqrt{1-4\gamma^2}(W_t^+ + W_t^-).$$

That is, $J(\theta^{(t)}) \leq \sqrt{1-4\gamma^2} J(\theta^{(t-1)})$. □