

Stanford Online Course

# CS229 NOTES

Thomas Lin

# CS229 Lecture Notes

Andrew Ng

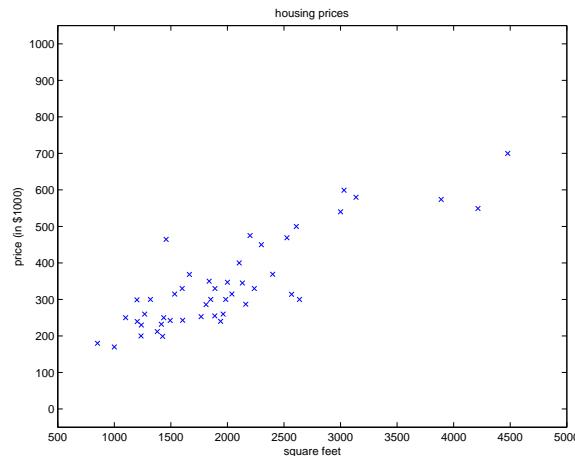
slightly updated by TM on June 28, 2019

## 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 <sup>2</sup> )	Price (1000\$s)
2104	400
1600	330
2400	369
1416	232
3000	540
:	:

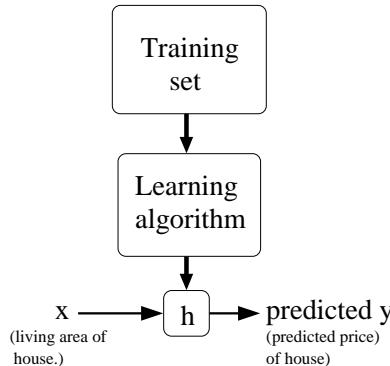
We can plot this data:



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  $n$  training examples  $\{(x^{(i)}, y^{(i)}); i = 1, \dots, n\}$ —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 <sup>2</sup> )	#bedrooms	Price (1000\$s)
2104	3	400
1600	3	330
2400	3	369
1416	2	232
3000	4	540
:	:	:

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  $\theta_i$ '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  $\theta$  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

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

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

Now, given a training set, how do we pick, or learn, the parameters  $\theta$ ? 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  $\theta$ 's, how close the  $h(x^{(i)})$ 's are to the corresponding  $y^{(i)}$ 's. We define the **cost function**:

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

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  $\theta$  so as to minimize  $J(\theta)$ . To do so, let's use a search algorithm that starts with some "initial guess" for  $\theta$ , and that repeatedly changes  $\theta$  to make  $J(\theta)$  smaller, until hopefully we converge to a value of  $\theta$  that minimizes  $J(\theta)$ . Specifically, let's consider the **gradient descent** algorithm, which starts with some initial  $\theta$ , 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  $j = 0, \dots, d$ .) Here,  $\alpha$  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:

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

For a single training example, this gives the update rule:<sup>1</sup>

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

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 {

$$\theta_j := \theta_j + \alpha \sum_{i=1}^n (y^{(i)} - h_\theta(x^{(i)})) x_j^{(i)}, \text{ (for every } j) \quad (1)$$

}

By grouping the updates of the coordinates into an update of the vector  $\theta$ , we can rewrite update (1) in a slightly more succinct way:

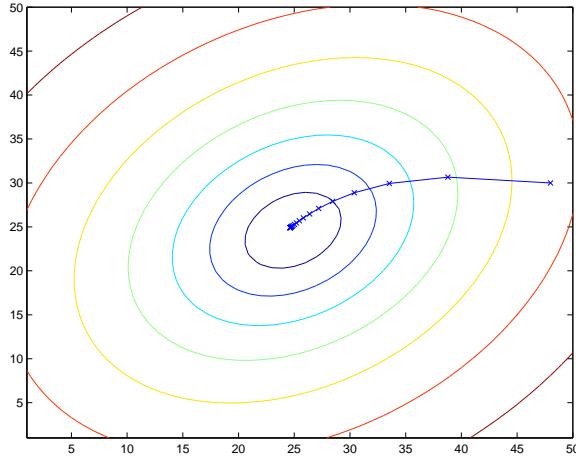
$$\theta := \theta + \alpha \sum_{i=1}^n (y^{(i)} - h_\theta(x^{(i)})) x^{(i)}$$

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

---

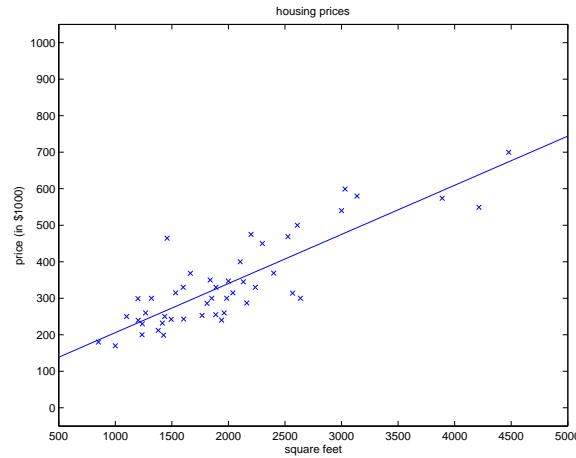
<sup>1</sup>We use the notation “ $a := b$ ” to denote an operation (in a computer program) in which we *set* the value of a variable  $a$  to be equal to the value of  $b$ . In other words, this operation overwrites  $a$  with the value of  $b$ . In contrast, we will write “ $a = b$ ” when we are asserting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

for linear regression has only one global, and no other local, optima; thus gradient descent always converges (assuming the learning rate  $\alpha$  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.



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  $\theta$  that gradient descent went through.

When we run batch gradient descent to fit  $\theta$  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 n, {
        
$$\theta_j := \theta_j + \alpha (y^{(i)} - h_{\theta}(x^{(i)})) x_j^{(i)}, \quad (\text{for every } j) \quad (2)$$

    }
}

```

By grouping the updates of the coordinates into an update of the vector  $\theta$ , we can rewrite update (2) in a slightly more succinct way:

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

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  $n$  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  $\theta$  “close” to the minimum much faster than batch gradient descent. (Note however that it may never “converge” to the minimum, and the parameters  $\theta$  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.<sup>2</sup>) For these reasons, particularly when the training set is large, stochastic gradient descent is often preferred over batch gradient descent.

---

<sup>2</sup>By slowly letting the learning rate  $\alpha$  decrease to zero as the algorithm runs, it is also possible to ensure that the parameters will converge to the global minimum rather than merely oscillate around the minimum.

## 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  $\theta_j$ '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.

### 2.1 Matrix derivatives

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

$$\nabla_A f(A) = \begin{bmatrix} \frac{\partial f}{\partial A_{11}} & \cdots & \frac{\partial f}{\partial A_{1d}} \\ \vdots & \ddots & \vdots \\ \frac{\partial f}{\partial A_{n1}} & \cdots & \frac{\partial f}{\partial A_{nd}} \end{bmatrix}$$

Thus, the gradient  $\nabla_A f(A)$  is itself an  $n$ -by- $d$  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

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

### 2.2 Least squares revisited

Armed with the tools of matrix derivatives, let us now proceed to find in closed-form the value of  $\theta$  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  $n$ -by- $d$  matrix (actually  $n$ -by- $d + 1$ , if we include the intercept term) that contains the

training examples' input values in its rows:

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

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

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

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

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

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

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

Finally, to minimize  $J$ , let's find its derivatives with respect to  $\theta$ . Hence,

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

In the third step, we used the fact that  $a^T b = b^T a$ , and in the fifth step used the facts  $\nabla_x b^T x = b$  and  $\nabla_x x^T A x = 2Ax$  for symmetric matrix  $A$  (for more details, see Section 4.3 of “Linear Algebra Review and Reference”). 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  $\theta$  that minimizes  $J(\theta)$  is given in closed form by the equation

$$\theta = (X^T X)^{-1} X^T \vec{y}.^3$$

### 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  $\sigma^2$ . 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

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

---

<sup>3</sup>Note that in the above step, we are implicitly assuming that  $X^T X$  is an invertible matrix. This can be checked before calculating the inverse. If either the number of linearly independent examples is fewer than the number of features, or if the features are not linearly independent, then  $X^T X$  will not be invertible. Even in such cases, it is possible to “fix” the situation with additional techniques, which we skip here for the sake of simplicity.

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

Given  $X$  (the design matrix, which contains all the  $x^{(i)}$ 's) and  $\theta$ , 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  $\theta$ . When we wish to explicitly view this as a function of  $\theta$ , 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

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

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  $\theta$ ? The principle of **maximum likelihood** says that we should choose  $\theta$  so as to make the data as high probability as possible. I.e., we should choose  $\theta$  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**  $\ell(\theta)$ :

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

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

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

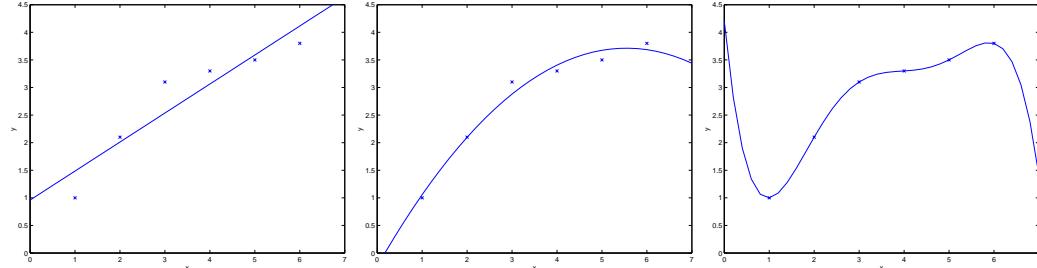
which we recognize to be  $J(\theta)$ , our original least-squares cost function.

To summarize: Under the previous probabilistic assumptions on the data, least-squares regression corresponds to finding the maximum likelihood estimate of  $\theta$ . 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  $\theta$  did not depend on what was  $\sigma^2$ , and indeed we'd have arrived at the same result even if  $\sigma^2$  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  $\theta$  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  $\theta$  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  $\theta$ , 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<sup>4</sup>

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

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,  $\theta$  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

---

<sup>4</sup>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  $\tau$  or  $\Sigma$ .

$\tau$  controls how quickly the weight of a training example falls off with distance of its  $x^{(i)}$  from the query point  $x$ ;  $\tau$  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  $\theta_i$ 's), which are fit to the data. Once we've fit the  $\theta_i$ '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.

## 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  $y \in \{0, 1\}$ .

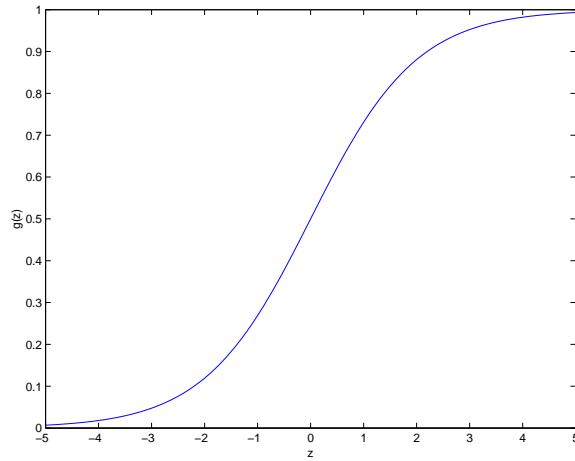
To fix this, let's change the form for our hypotheses  $h_\theta(x)$ . We will choose

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

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  $z \rightarrow \infty$ , and  $g(z)$  tends towards 0 as  $z \rightarrow -\infty$ . 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  $\theta^T x = \theta_0 + \sum_{j=1}^d \theta_j x_j$ .

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  $g'$ :

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

So, given the logistic regression model, how do we fit  $\theta$  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  $n$  training examples were generated independently, we can then write down the likelihood of the parameters as

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

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

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

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:

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

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

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

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 we 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

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

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 $\ell(\theta)$

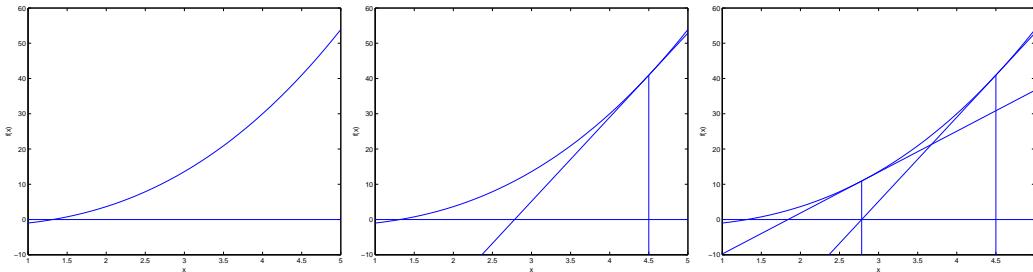
Returning to logistic regression with  $g(z)$  being the sigmoid function, let’s now talk about a different algorithm for maximizing  $\ell(\theta)$ .

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  $\theta$  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  $\theta$ , solving for where that linear function equals to zero, and letting the next guess for  $\theta$  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  $\theta$  so that  $f(\theta) = 0$ ; the value of  $\theta$  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  $\theta$ , which is about 2.8. The rightmost figure shows the result of running one more iteration, which updates  $\theta$  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  $\ell$ ? The maxima of  $\ell$  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  $\ell$ , 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,  $\theta$  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  $\theta_i$ 's; and  $H$  is an  $d$ -by- $d$  matrix (actually,  $d+1$ -by- $d+1$ , 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  $d$ -by- $d$  Hessian; but so long as  $d$  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<sup>5</sup>

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  $\mu$  and  $\phi$  as functions of  $x$  and  $\theta$ . 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

$$p(y; \eta) = b(y) \exp(\eta^T T(y) - a(\eta)) \quad (3)$$

Here,  $\eta$  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  $\eta$ ; as we vary  $\eta$ , 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  $\phi$ , 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  $\phi$ , we obtain Bernoulli distributions with different means. We now show that this class of Bernoulli distributions, ones obtained by varying  $\phi$ , is in the exponential family; i.e., that there is a choice of  $T$ ,  $a$  and  $b$  so that Equation (3) becomes exactly the class of Bernoulli distributions.

---

<sup>5</sup>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  $\eta$  by solving for  $\phi$  in terms of  $\eta$ , 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 (3), 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  $\sigma^2$  had no effect on our final choice of  $\theta$  and  $h_\theta(x)$ . Thus, we can choose an arbitrary value for  $\sigma^2$  without changing anything. To simplify the derivation below, let's set  $\sigma^2 = 1$ .<sup>6</sup> We then have:

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

---

<sup>6</sup>If we leave  $\sigma^2$  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  $\mu$  and  $\sigma$ . For the purposes of GLMs, however, the  $\sigma^2$  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,  $\tau$  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  $\theta$ ) 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.  $y | x; \theta \sim \text{ExponentialFamily}(\eta)$ . I.e., given  $x$  and  $\theta$ , the distribution of  $y$  follows some exponential family distribution, with parameter  $\eta$ .
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  $h(x) = E[y|x]$ . (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  $h_\theta(x) = p(y=1|x; \theta) = 0 \cdot p(y=0|x; \theta) + 1 \cdot p(y=1|x; \theta) = E[y|x; \theta].$ )

3. The natural parameter  $\eta$  and the inputs  $x$  are related linearly:  $\eta = \theta^T x$ .  
(Or, if  $\eta$  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 a Gaussian  $\mathcal{N}(\mu, \sigma^2)$ . (Here,  $\mu$  may depend  $x$ .) So, we let the  $\text{ExponentialFamily}(\eta)$  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  $\mu$ ; 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  $E[y|x; \theta] = \phi$ . 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 ( $g(\eta) = E[T(y); \eta]$ ) 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 identity function; and the canonical response function for the Bernoulli is the logistic function.<sup>7</sup>

## 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  $y \in \{1, 2, \dots, k\}$ . 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.

---

<sup>7</sup>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  $\phi_1, \dots, \phi_k$  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  $\phi_i$ '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,  $\phi_1, \dots, \phi_{k-1}$ , 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  $\phi_1, \dots, \phi_{k-1}$ .

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

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

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  $E[(T(y))_i] = P(y = i) = \phi_i$ .

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

exponential family. We have:

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

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  $i = 1, \dots, k$ ) 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

$$\begin{aligned}
e^{\eta_i} &= \frac{\phi_i}{\phi_k} \\
\phi_k e^{\eta_i} &= \phi_i \\
\phi_k \sum_{i=1}^k e^{\eta_i} &= \sum_{i=1}^k \phi_i = 1
\end{aligned} \tag{4}$$

This implies that  $\phi_k = 1 / \sum_{i=1}^k e^{\eta_i}$ , which can be substituted back into Equation (4) to give the response function

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

This function mapping from the  $\eta$ 's to the  $\phi$ 's is called the **softmax** function.

To complete our model, we use Assumption 3, given earlier, that the  $\eta_i$ 's are linearly related to the  $x$ 's. So, have  $\eta_i = \theta_i^T x$  (for  $i = 1, \dots, k-1$ ), where  $\theta_1, \dots, \theta_{k-1} \in \mathbb{R}^{d+1}$  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

$$\begin{aligned} p(y = i|x; \theta) &= \phi_i \\ &= \frac{e^{\eta_i}}{\sum_{j=1}^k e^{\eta_j}} \\ &= \frac{e^{\theta_i^T x}}{\sum_{j=1}^k e^{\theta_j^T x}} \end{aligned} \tag{5}$$

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

Our hypothesis will output

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

In other words, our hypothesis will output the estimated probability that  $p(y = i|x; \theta)$ , for every value of  $i = 1, \dots, k$ . (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  $n$  examples  $\{(x^{(i)}, y^{(i)}); i = 1, \dots, n\}$  and would like to learn the parameters  $\theta_i$  of this model, we would begin by writing down the log-likelihood

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

To obtain the second line above, we used the definition for  $p(y|x; \theta)$  given in Equation (5). We can now obtain the maximum likelihood estimate of the parameters by maximizing  $\ell(\theta)$  in terms of  $\theta$ , 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  $d$ -dimensions, also called the multivariate Gaussian distribution, is parameterized by a **mean vector**  $\mu \in \mathbb{R}^d$  and a **covariance matrix**  $\Sigma \in \mathbb{R}^{d \times d}$ , where  $\Sigma \geq 0$  is symmetric and positive semi-definite. Also written " $\mathcal{N}(\mu, \Sigma)$ ", its density is given by:

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

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

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

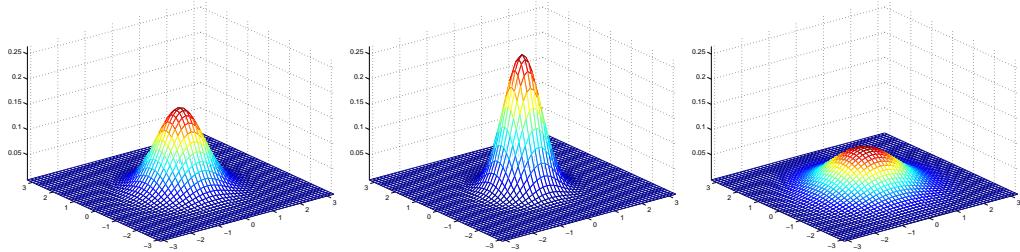
$$\mathbb{E}[X] = \int_x x p(x; \mu, \Sigma) dx = \mu$$

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

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

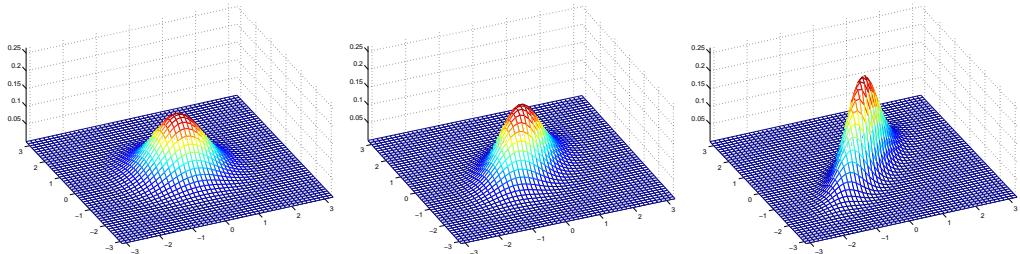
$$\text{Cov}(X) = \Sigma.$$

Here are 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  $2 \times 1$  zero-vector) and covariance matrix  $\Sigma = I$  (the  $2 \times 2$  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  $\Sigma$  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.

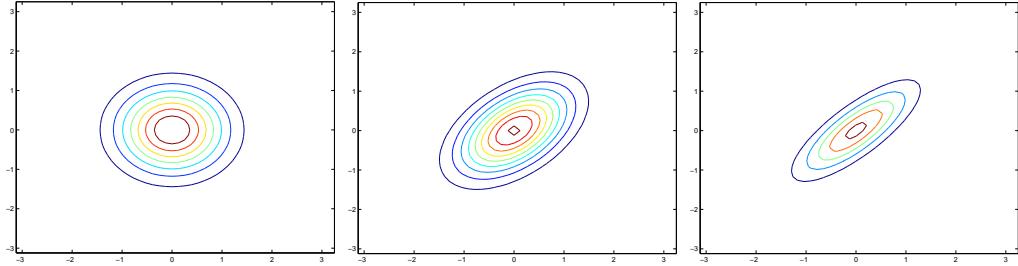


The figures above show Gaussians with mean 0, and with covariance matrices respectively

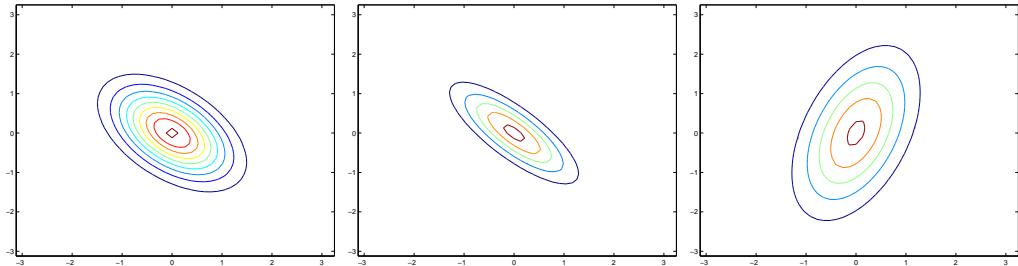
$$\Sigma = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}; \quad \Sigma = \begin{bmatrix} 1 & 0.5 \\ 0.5 & 1 \end{bmatrix}; \quad \Sigma = \begin{bmatrix} 1 & 0.8 \\ 0.8 & 1 \end{bmatrix}.$$

The leftmost figure shows the familiar standard normal distribution, and we see that as we increase the off-diagonal entry in  $\Sigma$ , the density becomes more

“compressed” towards the  $45^\circ$  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  $\Sigma$ :

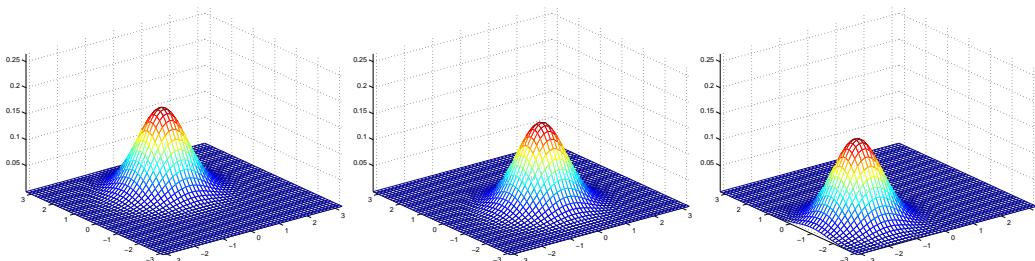


The plots above used, respectively,

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

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  $\mu$ , 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:

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

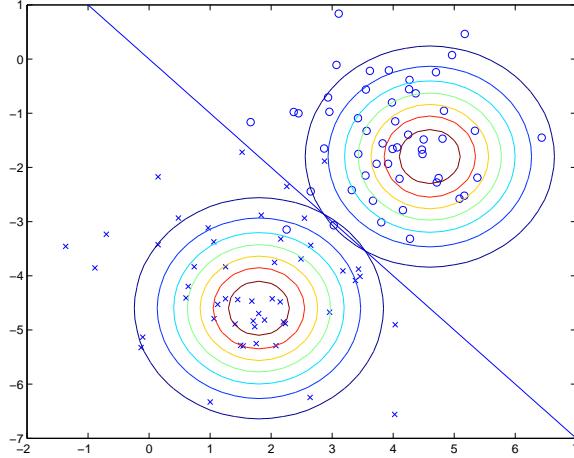
Here, the parameters of our model are  $\phi$ ,  $\Sigma$ ,  $\mu_0$  and  $\mu_1$ . (Note that while there're two different mean vectors  $\mu_0$  and  $\mu_1$ , this model is usually applied using only one covariance matrix  $\Sigma$ .) The log-likelihood of the data is given by

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

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

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

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  $\Sigma$ , but they have different means  $\mu_0$  and  $\mu_1$ . 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  $\theta$  is some appropriate function of  $\phi, \Sigma, \mu_0, \mu_1$ .<sup>1</sup> 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  $\Sigma$ ), 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  $\Sigma$ ), then GDA is **asymptotically efficient**. Informally, this means that in the limit of very large training sets (large  $n$ ), 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

---

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

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 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  $x_j$ '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  $x_j$  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  $j$ -th word of the dictionary, then we will set  $x_j = 1$ ; otherwise, we let  $x_j = 0$ . For instance, the vector

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

is used to represent an email that contains the words "a" and "buy," but not

“aardvark,” “aardwolf” or “zygmurgy.”<sup>2</sup> The set of words encoded into 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:

$$\begin{aligned} & p(x_1, \dots, x_{50000}|y) \\ &= p(x_1|y)p(x_2|y, x_1)p(x_3|y, x_1, x_2) \cdots p(x_{50000}|y, x_1, \dots, x_{49999}) \\ &= p(x_1|y)p(x_2|y)p(x_3|y) \cdots p(x_{50000}|y) \\ &= \prod_{j=1}^d p(x_j|y) \end{aligned}$$

The first equality simply follows from the usual properties of probabilities, and the second equality used the NB assumption. We note that even though

---

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

the Naive Bayes assumption is an extremely strong assumptions, the resulting algorithm works well on many problems.

Our model is parameterized by  $\phi_{j|y=1} = p(x_j = 1|y = 1)$ ,  $\phi_{j|y=0} = p(x_j = 1|y = 0)$ , and  $\phi_y = p(y = 1)$ . As usual, given a training set  $\{(x^{(i)}, y^{(i)}); i = 1, \dots, n\}$ , we can write down the joint likelihood of the data:

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

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

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

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

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

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  $x_j$  are binary-valued, the generalization to where  $x_j$  can take values in  $\{1, 2, \dots, k_j\}$  is straightforward. Here, we would simply model  $p(x_j|y)$  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  $x_j$  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  $x_j$  to 3. We can then apply the Naive Bayes algorithm, and model  $p(x_j|y)$  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, we are in the year of 20xx, after completing CS229 and having done excellent work on the project, you decide around May 20xx to submit work you did to the NeurIPS conference for publication.<sup>3</sup> Because you end up discussing the conference in your emails, you also start getting messages with the word “neurips” in it. But this is your first NeurIPS paper, and until this time, you had not previously seen any emails containing the word “neurips”; in particular “neurips” did not ever appear in your training set of spam/non-spam emails. Assuming that “neurips” 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

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

I.e., because it has never seen “neurips” 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

---

<sup>3</sup>NeurIPS is one of the top machine learning conferences. The deadline for submitting a paper is typically in May-June.

“neurips” is spam, it calculates the class posterior probabilities, and obtains

$$\begin{aligned} p(y=1|x) &= \frac{\prod_{j=1}^d p(x_j|y=1)p(y=1)}{\prod_{j=1}^d p(x_j|y=1)p(y=1) + \prod_{j=1}^d p(x_j|y=0)p(y=0)} \\ &= \frac{0}{0}. \end{aligned}$$

This is because each of the terms “ $\prod_{j=1}^d p(x_j|y)$ ” 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  $\{1, \dots, k\}$ . We can parameterize our multinomial with  $\phi_j = p(z=j)$ . Given a set of  $n$  independent observations  $\{z^{(1)}, \dots, z^{(n)}\}$ , the maximum likelihood estimates are given by

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

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

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

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  $\phi_j$ ’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  $\phi_j$ ’s.

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

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

(In practice, it usually doesn't matter much whether we apply Laplace smoothing to  $\phi_y$  or not, since we will typically have a fair fraction each of spam and non-spam messages, so  $\phi_y$  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 **Bernoulli event model** (or sometimes **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  $j$  in that email independently and according to the probabilities  $p(x_j = 1|y) = \phi_{j|y}$ . Thus, the probability of a message was given by  $p(y) \prod_{j=1}^d p(x_j|y)$ .

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  $x_j$  denote the identity of the  $j$ -th word in the email. Thus,  $x_j$  is now an integer taking values in  $\{1, \dots, |V|\}$ , where  $|V|$  is the size of our vocabulary (dictionary). An email of  $d$  words is now represented by a vector  $(x_1, x_2, \dots, x_d)$  of length  $d$ ; note that  $d$  can vary for different documents. For instance, if an email starts with "A NeurIPS ...," then  $x_1 = 1$  ("a" is the first word in the dictionary), and  $x_2 = 35000$  ("neurips" 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  $d$  words of the email have been generated. Thus, the overall probability of a message is given by  $p(y) \prod_{j=1}^d p(x_j|y)$ . Note that this formula looks like the one we had earlier for the probability of a message under the Bernoulli event model, but that the terms in the formula now mean very different things. In particular  $x_j|y$  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  $\phi_{k|y=0} = p(x_j = k|y = 0)$ . 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  $\{(x^{(i)}, y^{(i)}); i = 1, \dots, n\}$  where  $x^{(i)} = (x_1^{(i)}, x_2^{(i)}, \dots, x_{d_i}^{(i)})$  (here,  $d_i$  is the number of words in the  $i$ -training example), the likelihood of the data is given by

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

Maximizing this yields the maximum likelihood estimates of the parameters:

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

If we were to apply Laplace smoothing (which is 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:

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

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

updated by Tengyu Ma on April 21, 2019

## Part V Kernel Methods

### 1.1 Feature maps

Recall that in our discussion about linear regression, we considered the problem of predicting the price of a house (denoted by  $y$ ) from the living area of the house (denoted by  $x$ ), and we fit a linear function of  $x$  to the training data. What if the price  $y$  can be more accurately represented as a *non-linear* function of  $x$ ? In this case, we need a more expressive family of models than linear models.

We start by considering fitting cubic functions  $y = \theta_3x^3 + \theta_2x^2 + \theta_1x + \theta_0$ . It turns out that we can view the cubic function as a linear function over the a different set of feature variables (defined below). Concretely, let the function  $\phi : \mathbb{R} \rightarrow \mathbb{R}^4$  be defined as

$$\phi(x) = \begin{bmatrix} 1 \\ x \\ x^2 \\ x^3 \end{bmatrix} \in \mathbb{R}^4. \quad (1)$$

Let  $\theta \in \mathbb{R}^4$  be the vector containing  $\theta_0, \theta_1, \theta_2, \theta_3$  as entries. Then we can rewrite the cubic function in  $x$  as:

$$\theta_3x^3 + \theta_2x^2 + \theta_1x + \theta_0 = \theta^T\phi(x)$$

Thus, a cubic function of the variable  $x$  can be viewed as a linear function over the variables  $\phi(x)$ . To distinguish between these two sets of variables,

in the context of kernel methods, we will call the “original” input value the input **attributes** of a problem (in this case,  $x$ , the living area). When the original input is mapped to some new set of quantities  $\phi(x)$ , we will call those new quantities the **features** variables. (Unfortunately, different authors use different terms to describe these two things in different contexts.) We will call  $\phi$  a **feature map**, which maps the attributes to the features.

## 1.2 LMS (least mean squares) with features

We will derive the gradient descent algorithm for fitting the model  $\theta^T \phi(x)$ . First recall that for ordinary least square problem where we were to fit  $\theta^T x$ , the batch gradient descent update is (see the first lecture note for its derivation):

$$\begin{aligned}\theta &:= \theta + \alpha \sum_{i=1}^n (y^{(i)} - h_\theta(x^{(i)})) x^{(i)} \\ &:= \theta + \alpha \sum_{i=1}^n (y^{(i)} - \theta^T x^{(i)}) x^{(i)}.\end{aligned}\tag{2}$$

Let  $\phi : \mathbb{R}^d \rightarrow \mathbb{R}^p$  be a feature map that maps attribute  $x$  (in  $\mathbb{R}^d$ ) to the features  $\phi(x)$  in  $\mathbb{R}^p$ . (In the motivating example in the previous subsection, we have  $d = 1$  and  $p = 4$ .) Now our goal is to fit the function  $\theta^T \phi(x)$ , with  $\theta$  being a vector in  $\mathbb{R}^p$  instead of  $\mathbb{R}^d$ . We can replace all the occurrences of  $x^{(i)}$  in the algorithm above by  $\phi(x^{(i)})$  to obtain the new update:

$$\theta := \theta + \alpha \sum_{i=1}^n (y^{(i)} - \theta^T \phi(x^{(i)})) \phi(x^{(i)})\tag{3}$$

Similarly, the corresponding stochastic gradient descent update rule is

$$\theta := \theta + \alpha (y^{(i)} - \theta^T \phi(x^{(i)})) \phi(x^{(i)})\tag{4}$$

## 1.3 LMS with the kernel trick

The gradient descent update, or stochastic gradient update above becomes computationally expensive when the features  $\phi(x)$  is high-dimensional. For example, consider the direct extension of the feature map in equation (1) to

high-dimensional input  $x$ : suppose  $x \in \mathbb{R}^d$ , and let  $\phi(x)$  be the vector that contains all the monomials of  $x$  with degree  $\leq 3$

$$\phi(x) = \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ \vdots \\ x_1^2 \\ x_1 x_2 \\ x_1 x_3 \\ \vdots \\ x_2 x_1 \\ \vdots \\ x_1^3 \\ x_1^2 x_2 \\ \vdots \end{bmatrix}. \quad (5)$$

The dimension of the features  $\phi(x)$  is on the order of  $d^3$ .<sup>1</sup> This is a prohibitively long vector for computational purpose — when  $d = 1000$ , each update requires at least computing and storing a  $1000^3 = 10^9$  dimensional vector, which is  $10^6$  times slower than the update rule for ordinary least squares updates (2).

It may appear at first that such  $d^3$  runtime per update and memory usage are inevitable, because the vector  $\theta$  itself is of dimension  $p \approx d^3$ , and we may need to update every entry of  $\theta$  and store it. However, we will introduce the kernel trick with which we will not need to store  $\theta$  explicitly, and the runtime can be significantly improved.

For simplicity, we assume the initialize the value  $\theta = 0$ , and we focus on the iterative update (3). The main observation is that at any time,  $\theta$  can be represented as a linear combination of the vectors  $\phi(x^{(1)}), \dots, \phi(x^{(n)})$ . Indeed, we can show this inductively as follows. At initialization,  $\theta = 0 = \sum_{i=1}^n 0 \cdot \phi(x^{(i)})$ . Assume at some point,  $\theta$  can be represented as

$$\theta = \sum_{i=1}^n \beta_i \phi(x^{(i)}) \quad (6)$$

---

<sup>1</sup>Here, for simplicity, we include all the monomials with repetitions (so that, e.g.,  $x_1 x_2 x_3$  and  $x_2 x_3 x_1$  both appear in  $\phi(x)$ ). Therefore, there are totally  $1 + d + d^2 + d^3$  entries in  $\phi(x)$ .

for some  $\beta_1, \dots, \beta_n \in \mathbb{R}$ . Then we claim that in the next round,  $\theta$  is still a linear combination of  $\phi(x^{(1)}), \dots, \phi(x^{(n)})$  because

$$\begin{aligned}\theta &:= \theta + \alpha \sum_{i=1}^n (y^{(i)} - \theta^T \phi(x^{(i)})) \phi(x^{(i)}) \\ &= \sum_{i=1}^n \beta_i \phi(x^{(i)}) + \alpha \sum_{i=1}^n (y^{(i)} - \theta^T \phi(x^{(i)})) \phi(x^{(i)}) \\ &= \sum_{i=1}^n \underbrace{(\beta_i + \alpha (y^{(i)} - \theta^T \phi(x^{(i)})))}_{\text{new } \beta_i} \phi(x^{(i)})\end{aligned}\tag{7}$$

You may realize that our general strategy is to implicitly represent the  $p$ -dimensional vector  $\theta$  by a set of coefficients  $\beta_1, \dots, \beta_n$ . Towards doing this, we derive the update rule of the coefficients  $\beta_1, \dots, \beta_n$ . Using the equation above, we see that the new  $\beta_i$  depends on the old one via

$$\beta_i := \beta_i + \alpha (y^{(i)} - \theta^T \phi(x^{(i)}))\tag{8}$$

Here we still have the old  $\theta$  on the RHS of the equation. Replacing  $\theta$  by  $\theta = \sum_{j=1}^n \beta_j \phi(x^{(j)})$  gives

$$\forall i \in \{1, \dots, n\}, \beta_i := \beta_i + \alpha \left( y^{(i)} - \sum_{j=1}^n \beta_j \phi(x^{(j)})^T \phi(x^{(i)}) \right)$$

We often rewrite  $\phi(x^{(j)})^T \phi(x^{(i)})$  as  $\langle \phi(x^{(j)}), \phi(x^{(i)}) \rangle$  to emphasize that it's the inner product of the two feature vectors. Viewing  $\beta_i$ 's as the new representation of  $\theta$ , we have successfully translated the batch gradient descent algorithm into an algorithm that updates the value of  $\beta$  iteratively. It may appear that at every iteration, we still need to compute the values of  $\langle \phi(x^{(j)}), \phi(x^{(i)}) \rangle$  for all pairs of  $i, j$ , each of which may take roughly  $O(p)$  operation. However, two important properties come to rescue:

1. We can pre-compute the pairwise inner products  $\langle \phi(x^{(j)}), \phi(x^{(i)}) \rangle$  for all pairs of  $i, j$  before the loop starts.
2. For the feature map  $\phi$  defined in (5) (or many other interesting feature maps), computing  $\langle \phi(x^{(j)}), \phi(x^{(i)}) \rangle$  can be efficient and does not

necessarily require computing  $\phi(x^{(i)})$  explicitly. This is because:

$$\begin{aligned}
\langle \phi(x), \phi(z) \rangle &= 1 + \sum_{i=1}^d x_i z_i + \sum_{i,j \in \{1, \dots, d\}} x_i x_j z_i z_j + \sum_{i,j,k \in \{1, \dots, d\}} x_i x_j x_k z_i z_j z_k \\
&= 1 + \sum_{i=1}^d x_i z_i + \left( \sum_{i=1}^d x_i z_i \right)^2 + \left( \sum_{i=1}^d x_i z_i \right)^3 \\
&= 1 + \langle x, z \rangle + \langle x, z \rangle^2 + \langle x, z \rangle^3
\end{aligned} \tag{9}$$

Therefore, to compute  $\langle \phi(x), \phi(z) \rangle$ , we can first compute  $\langle x, z \rangle$  with  $O(d)$  time and then take another constant number of operations to compute  $1 + \langle x, z \rangle + \langle x, z \rangle^2 + \langle x, z \rangle^3$ .

As you will see, the inner products between the features  $\langle \phi(x), \phi(z) \rangle$  are essential here. We define the **Kernel** corresponding to the feature map  $\phi$  as a function that maps  $\mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}$  satisfying:<sup>2</sup>

$$K(x, z) \triangleq \langle \phi(x), \phi(z) \rangle \tag{10}$$

To wrap up the discussion, we write down the final algorithm as follows:

- 
1. Compute all the values  $K(x^{(i)}, x^{(j)}) \triangleq \langle \phi(x^{(i)}), \phi(x^{(j)}) \rangle$  using equation (9) for all  $i, j \in \{1, \dots, n\}$ . Set  $\beta := 0$ .

## 2. Loop:

$$\forall i \in \{1, \dots, n\}, \beta_i := \beta_i + \alpha \left( y^{(i)} - \sum_{j=1}^n \beta_j K(x^{(i)}, x^{(j)}) \right) \tag{11}$$

Or in vector notation, letting  $K$  be the  $n \times n$  matrix with  $K_{ij} = K(x^{(i)}, x^{(j)})$ , we have

$$\beta := \beta + \alpha(\vec{y} - K\beta)$$


---

With the algorithm above, we can update the representation  $\beta$  of the vector  $\theta$  efficiently with  $O(n)$  time per update. Finally, we need to show that

---

<sup>2</sup>Recall that  $\mathcal{X}$  is the space of the input  $x$ . In our running example,  $\mathcal{X} = \mathbb{R}^d$

the knowledge of the representation  $\beta$  suffices to compute the prediction  $\theta^T \phi(x)$ . Indeed, we have

$$\theta^T \phi(x) = \sum_{i=1}^n \beta_i \phi(x^{(i)})^T \phi(x) = \sum_{i=1}^n \beta_i K(x^{(i)}, x) \quad (12)$$

You may realize that fundamentally all we need to know about the feature map  $\phi(\cdot)$  is encapsulated in the corresponding kernel function  $K(\cdot, \cdot)$ . We will expand on this in the next section.

## 1.4 Properties of kernels

In the last subsection, we started with an explicitly defined feature map  $\phi$ , which induces the kernel function  $K(x, z) \triangleq \langle \phi(x), \phi(z) \rangle$ . Then we saw that the kernel function is so intrinsic so that as long as the kernel function is defined, the whole training algorithm can be written entirely in the language of the kernel without referring to the feature map  $\phi$ , so can the prediction of a test example  $x$  (equation (12).)

Therefore, it would be tempted to define other kernel function  $K(\cdot, \cdot)$  and run the algorithm (11). Note that the algorithm (11) does not need to explicitly access the feature map  $\phi$ , and therefore we only need to ensure the existence of the feature map  $\phi$ , but do not necessarily need to be able to explicitly write  $\phi$  down.

What kinds of functions  $K(\cdot, \cdot)$  can correspond to some feature map  $\phi$ ? In other words, can we tell if there is some feature mapping  $\phi$  so that  $K(x, z) = \phi(x)^T \phi(z)$  for all  $x, z$ ?

If we can answer this question by giving a precise characterization of valid kernel functions, then we can completely change the interface of selecting feature maps  $\phi$  to the interface of selecting kernel function  $K$ . Concretely, we can pick a function  $K$ , verify that it satisfies the characterization (so that there exists a feature map  $\phi$  that  $K$  corresponds to), and then we can run update rule (11). The benefit here is that we don't have to be able to compute  $\phi$  or write it down analytically, and we only need to know its existence. We will answer this question at the end of this subsection after we go through several concrete examples of kernels.

Suppose  $x, z \in \mathbb{R}^d$ , and let's first consider the function  $K(\cdot, \cdot)$  defined as:

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

We can also write this as

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

Thus, we see that  $K(x, z) = \langle \phi(x), \phi(z) \rangle$  is the kernel function that corresponds to the feature mapping  $\phi$  given (shown here for the case of  $d = 3$ ) by

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

Revisiting the computational efficiency perspective of kernel, note that whereas calculating the high-dimensional  $\phi(x)$  requires  $O(d^2)$  time, finding  $K(x, z)$  takes only  $O(d)$  time—linear in the dimension of the input attributes.

For another related example, also consider  $K(\cdot, \cdot)$  defined by

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

(Check this yourself.) This function  $K$  is a kernel function that corresponds

to the feature mapping (again shown for  $d = 3$ )

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

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

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

**Kernels as similarity metrics.** 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 reasonable 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. Does there exist

a feature map  $\phi$  such that the kernel  $K$  defined above satisfies  $K(x, z) = \phi(x)^T \phi(z)$ ? In this particular example, the answer is yes. This kernel is called the **Gaussian kernel**, and corresponds to an infinite dimensional feature mapping  $\phi$ . We will give a precise characterization about what properties a function  $K$  needs to satisfy so that it can be a valid kernel function that corresponds to some feature map  $\phi$ .

**Necessary conditions for valid kernels.** Suppose for now that  $K$  is indeed a valid kernel corresponding to some feature mapping  $\phi$ , and we will first see what properties it satisfies. Now, consider some finite set of  $n$  points (not necessarily the training set)  $\{x^{(1)}, \dots, x^{(n)}\}$ , and let a square,  $n$ -by- $n$  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 uses the fact that  $\sum_{i,j} a_i a_j = (\sum_i a_i)^2$  for  $a_i = z_i \phi_k(x^{(i)})$ . 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  $\phi$ ), then the corresponding kernel matrix  $K \in \mathbb{R}^{n \times n}$  is symmetric positive semidefinite.

**Sufficient conditions for valid kernels.** More generally, the condition above 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.<sup>3</sup>

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

Given a function  $K$ , apart from trying to find a feature mapping  $\phi$  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  $K(x, z) = (x^T z)^k$  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 256-dimensional vectors 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.

---

<sup>3</sup>Many texts present Mercer's theorem in a slightly more complicated form involving  $L^2$  functions, but when the input attributes take values in  $\mathbb{R}^d$ , the version given here is equivalent.

**Application of kernel methods:** We've seen the application of kernels to linear regression. In the next part, we will introduce the support vector machines to which kernels can be directly applied. dwell too much longer on it here. In fact, the idea of kernels has significantly broader applicability than linear regression and 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 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."

## Part VI

# 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 algorithms. 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.

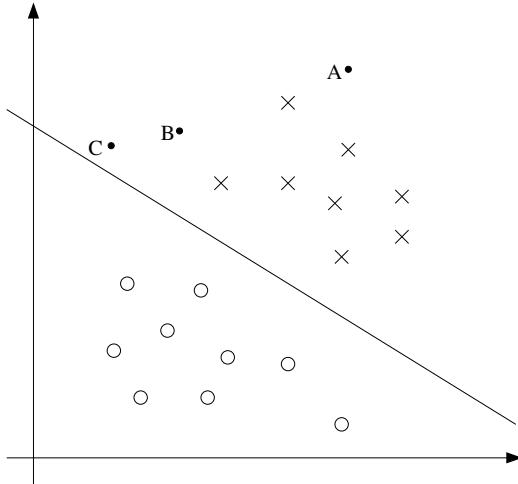
## 2 Margins: Intuition

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 4.

Consider logistic regression, where the probability  $p(y = 1|x; \theta)$  is modeled by  $h_\theta(x) = g(\theta^T x)$ . We 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  $h_\theta(x) = p(y = 1|x; \theta)$ , and thus also the higher our degree of "confidence" that the label is 1. Thus, informally we can think of our prediction as being very confident that

$y = 1$  if  $\theta^T x \gg 0$ . Similarly, we think of logistic regression as confidently predicting  $y = 0$ , 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  $\theta$  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 would 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.

### 3 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  $\theta$ , 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  $\theta_0$ , and  $w$  takes the role of  $[\theta_1 \dots \theta_d]^T$ .

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  $p(y = 1)$  (which is what logistic regression does).

### 4 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 as

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

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

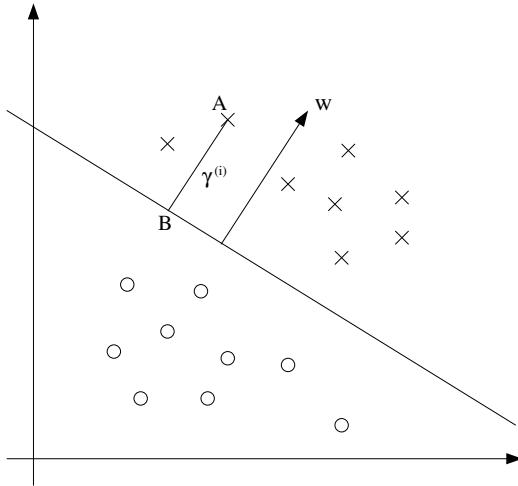
For a linear classifier with the choice of  $g$  given above (taking values in  $\{-1, 1\}$ ), there's one property of the functional margin that makes it not a very good measure of confidence, however. Given our choice of  $g$ , we note that if we replace  $w$  with  $2w$  and  $b$  with  $2b$ , then 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  $S = \{(x^{(i)}, y^{(i)}); i = 1, \dots, n\}$ , we also define the function margin of  $(w, b)$  with respect to  $S$  as the smallest of the functional margins of the individual training examples. Denoted by  $\hat{\gamma}$ , this can therefore be written:

$$\hat{\gamma} = \min_{i=1,\dots,n} \hat{\gamma}^{(i)}.$$

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  $90^\circ$ ) 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  $S = \{(x^{(i)}, y^{(i)}); i = 1, \dots, n\}$ , 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:

$$\gamma = \min_{i=1, \dots, n} \gamma^{(i)}.$$

## 5 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 will we find the one that achieves the maximum geometric margin? We can pose the following optimization problem:

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

I.e., we want to maximize  $\gamma$ , subject to each training example having functional margin at least  $\gamma$ . 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  $\gamma$ . 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:

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

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:

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

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.<sup>4</sup>

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.

## 6 Lagrange duality (optional reading)

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

Consider a problem of the following form:

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

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

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

---

<sup>4</sup>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  $\beta_i$ '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  $\beta$ .

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,<sup>5</sup> 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:

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

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  $\alpha_i$ 's and  $\beta_i$ '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

$$\theta_{\mathcal{P}}(w) = \max_{\alpha, \beta: \alpha_i \geq 0} f(w) + \sum_{i=1}^k \alpha_i g_i(w) + \sum_{i=1}^l \beta_i h_i(w) \tag{13}$$

$$= \infty. \tag{14}$$

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

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

---

<sup>5</sup>Readers interested in learning more about this topic are encouraged to read, e.g., R. T. Rockafellar (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  $\alpha, \beta$ , here we 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  $d^* = \max_{\alpha, \beta : \alpha_i \geq 0} \theta_{\mathcal{D}}(\alpha, \beta)$ .

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,<sup>6</sup> and the  $h_i$ 's are affine.<sup>7</sup> 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  $i$ .

---

<sup>6</sup>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.)

<sup>7</sup>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  $w^*, \alpha^*, \beta^*$  so that  $w^*$  is the solution to the primal problem,  $\alpha^*, \beta^*$  are the solution to the dual problem, and moreover  $p^* = d^* = \mathcal{L}(w^*, \alpha^*, \beta^*)$ . Moreover,  $w^*, \alpha^*$  and  $\beta^*$  satisfy the **Karush-Kuhn-Tucker (KKT) conditions**, which are as follows:

$$\frac{\partial}{\partial w_i} \mathcal{L}(w^*, \alpha^*, \beta^*) = 0, \quad i = 1, \dots, d \quad (15)$$

$$\frac{\partial}{\partial \beta_i} \mathcal{L}(w^*, \alpha^*, \beta^*) = 0, \quad i = 1, \dots, l \quad (16)$$

$$\alpha_i^* g_i(w^*) = 0, \quad i = 1, \dots, k \quad (17)$$

$$g_i(w^*) \leq 0, \quad i = 1, \dots, k \quad (18)$$

$$\alpha^* \geq 0, \quad i = 1, \dots, k \quad (19)$$

Moreover, if some  $w^*, \alpha^*, \beta^*$  satisfy the KKT conditions, then it is also a solution to the primal and dual problems.

We draw attention to Equation (17), 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.

## 7 Optimal margin classifiers

**Note:** *The equivalence of optimization problem (20) and the optimization problem (24), and the relationship between the primary and dual variables in equation (22) are the most important take home messages of this section.*

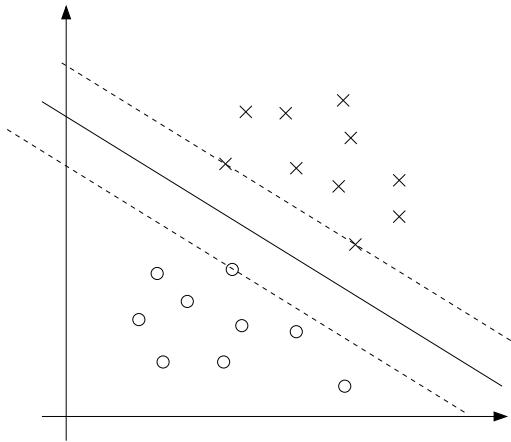
Previously, we posed the following (primal) optimization problem for finding the optimal margin classifier:

$$\begin{aligned} \min_{w,b} \quad & \frac{1}{2} \|w\|^2 \\ \text{s.t.} \quad & y^{(i)}(w^T x^{(i)} + b) \geq 1, \quad i = 1, \dots, n \end{aligned} \quad (20)$$

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  $\alpha_i$ '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 of 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:

$$\mathcal{L}(w, b, \alpha) = \frac{1}{2} \|w\|^2 - \sum_{i=1}^n \alpha_i [y^{(i)}(w^T x^{(i)} + b) - 1]. \quad (21)$$

Note that there're only “ $\alpha_i$ ” but no “ $\beta_i$ ” 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  $\alpha$ ), 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:

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

This implies that

$$w = \sum_{i=1}^n \alpha_i y^{(i)} x^{(i)}. \quad (22)$$

As for the derivative with respect to  $b$ , we obtain

$$\frac{\partial}{\partial b} \mathcal{L}(w, b, \alpha) = \sum_{i=1}^n \alpha_i y^{(i)} = 0. \quad (23)$$

If we take the definition of  $w$  in Equation (22) and plug that back into the Lagrangian (Equation 21), and simplify, we get

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

But from Equation (23), the last term must be zero, so we obtain

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

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 (23), we obtain the following dual optimization problem:

$$\begin{aligned} \max_{\alpha} \quad & W(\alpha) = \sum_{i=1}^n \alpha_i - \frac{1}{2} \sum_{i,j=1}^n y^{(i)} y^{(j)} \alpha_i \alpha_j \langle x^{(i)}, x^{(j)} \rangle. \\ \text{s.t.} \quad & \alpha_i \geq 0, \quad i = 1, \dots, n \\ & \sum_{i=1}^n \alpha_i y^{(i)} = 0, \end{aligned} \quad (24)$$

You should also be able to verify that the conditions required for  $p^* = d^*$  and the KKT conditions (Equations 15–19) 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  $\alpha_i$ '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  $\alpha$ 's that maximize  $W(\alpha)$  subject to the constraints), then we can use Equation (22) to go back and find the optimal  $w$ 's as a function of the  $\alpha$ 's. Having found  $w^*$ , by considering the primal problem, it is also straightforward to find the optimal value for the intercept term  $b$  as

$$b^* = -\frac{\max_{i:y^{(i)}=-1} w^{*T} x^{(i)} + \min_{i:y^{(i)}=1} w^{*T} x^{(i)}}{2}. \quad (25)$$

(Check for yourself that this is correct.)

Before moving on, let's also take a more careful look at Equation (22), which gives the optimal value of  $w$  in terms of (the optimal value of)  $\alpha$ . 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 (22), this quantity can also be written:

$$w^T x + b = \left( \sum_{i=1}^n \alpha_i y^{(i)} x^{(i)} \right)^T x + b \quad (26)$$

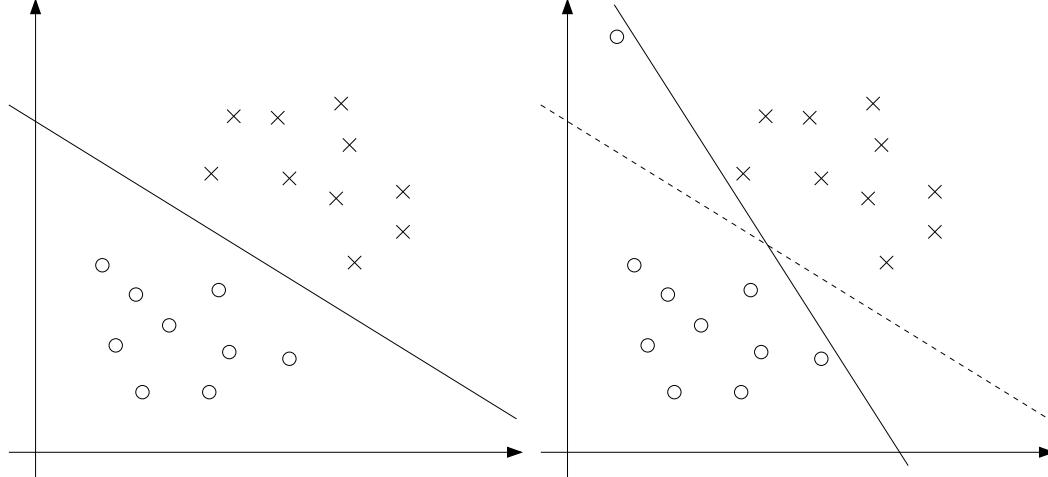
$$= \sum_{i=1}^n \alpha_i y^{(i)} \langle x^{(i)}, x \rangle + b. \quad (27)$$

Hence, if we've found the  $\alpha_i$ '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  $\alpha_i$ '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 (27) 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.

## 8 Regularization and the non-separable case (optional reading)

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  $\phi$  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  $\ell_1$  regularization) as follows:

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

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:

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

Here, the  $\alpha_i$ 's and  $r_i$ 's are our Lagrange multipliers (constrained to be  $\geq 0$ ). 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:

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

As before, we also have that  $w$  can be expressed in terms of the  $\alpha_i$ 's as given in Equation (22), so that after solving the dual problem, we can continue to use Equation (27) to make our predictions. Note that, somewhat surprisingly, in adding  $\ell_1$  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 25 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:

$$\alpha_i = 0 \Rightarrow y^{(i)}(w^T x^{(i)} + b) \geq 1 \tag{28}$$

$$\alpha_i = C \Rightarrow y^{(i)}(w^T x^{(i)} + b) \leq 1 \tag{29}$$

$$0 < \alpha_i < C \Rightarrow y^{(i)}(w^T x^{(i)} + b) = 1. \tag{30}$$

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 (optional reading)

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

$$\max_{\alpha} W(\alpha_1, \alpha_2, \dots, \alpha_n).$$

Here, we think of  $W$  as just some function of the parameters  $\alpha_i$ '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  $i = 1, \dots, n$ , {

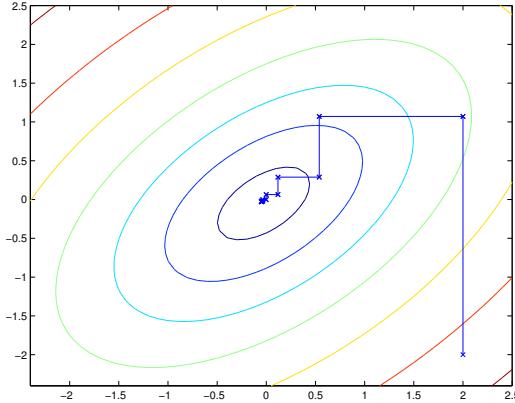
$$\alpha_i := \arg \max_{\hat{\alpha}_i} W(\alpha_1, \dots, \alpha_{i-1}, \hat{\alpha}_i, \alpha_{i+1}, \dots, \alpha_n).$$

}

}

Thus, in the innermost loop of this algorithm, we will hold all the variables except for some  $\alpha_i$  fixed, and reoptimize  $W$  with respect to just the parameter  $\alpha_i$ . In the version of this method presented here, the inner-loop reoptimizes the variables in order  $\alpha_1, \alpha_2, \dots, \alpha_n, \alpha_1, \alpha_2, \dots$  (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:

$$\max_{\alpha} \quad W(\alpha) = \sum_{i=1}^n \alpha_i - \frac{1}{2} \sum_{i,j=1}^n y^{(i)} y^{(j)} \alpha_i \alpha_j \langle x^{(i)}, x^{(j)} \rangle. \quad (31)$$

$$\text{s.t. } 0 \leq \alpha_i \leq C, \quad i = 1, \dots, n \quad (32)$$

$$\sum_{i=1}^n \alpha_i y^{(i)} = 0. \quad (33)$$

Let's say we have set of  $\alpha_i$ 's that satisfy the constraints (32-33). Now, suppose we want to hold  $\alpha_2, \dots, \alpha_n$  fixed, and take a coordinate ascent step and reoptimize the objective with respect to  $\alpha_1$ . Can we make any progress? The answer is no, because the constraint (33) ensures that

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

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

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

(This step used the fact that  $y^{(1)} \in \{-1, 1\}$ , and hence  $(y^{(1)})^2 = 1$ .) Hence,  $\alpha_1$  is exactly determined by the other  $\alpha_i$ 's, and if we were to hold  $\alpha_2, \dots, \alpha_n$  fixed, then we can't make any change to  $\alpha_1$  without violating the constraint (33) in the optimization problem.

Thus, if we want to update some subject of the  $\alpha_i$ '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  $\alpha_i$  and  $\alpha_j$  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  $\alpha_i$  and  $\alpha_j$ , while holding all the other  $\alpha_k$ 's ( $k \neq i, j$ ) fixed.

}

To test for convergence of this algorithm, we can check whether the KKT conditions (Equations 28-30) 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  $\alpha_i, \alpha_j$  can be computed very efficiently. Let's now briefly sketch the main ideas for deriving the efficient update.

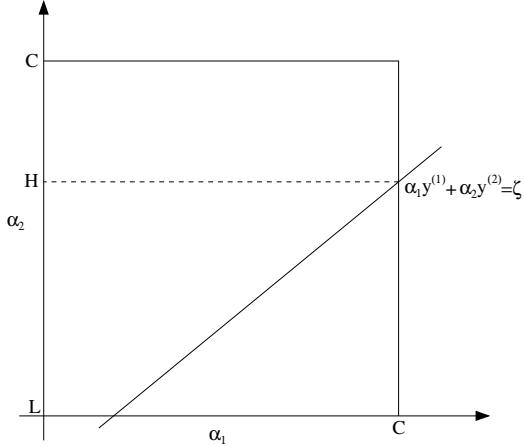
Let's say we currently have some setting of the  $\alpha_i$ 's that satisfy the constraints (32-33), and suppose we've decided to hold  $\alpha_3, \dots, \alpha_n$  fixed, and want to reoptimize  $W(\alpha_1, \alpha_2, \dots, \alpha_n)$  with respect to  $\alpha_1$  and  $\alpha_2$  (subject to the constraints). From (33), we require that

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

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

$$\alpha_1 y^{(1)} + \alpha_2 y^{(2)} = \zeta. \quad (34)$$

We can thus picture the constraints on  $\alpha_1$  and  $\alpha_2$  as follows:



From the constraints (32), we know that  $\alpha_1$  and  $\alpha_2$  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  $\alpha_1$  and  $\alpha_2$  must lie. Note also that, from these constraints, we know  $L \leq \alpha_2 \leq H$ ; otherwise,  $(\alpha_1, \alpha_2)$  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 permissible values for  $\alpha_2$  that will ensure that  $\alpha_1, \alpha_2$  lie within the box  $[0, C] \times [0, C]$ .

Using Equation (34), we can also write  $\alpha_1$  as a function of  $\alpha_2$ :

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

(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

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

Treating  $\alpha_3, \dots, \alpha_n$  as constants, you should be able to verify that this is just some quadratic function in  $\alpha_2$ . 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 (32) (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  $\alpha_2$ . You should also be able to convince yourself that if we had instead wanted to maximize  $W$  with respect to  $\alpha_2$  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  $\alpha_2^{new}$ , we can use Equation (34) to go back and find the optimal value of  $\alpha_1^{new}$ .

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  $\alpha_i, \alpha_j$  to update; the other is how to update  $b$  as the SMO algorithm is run.

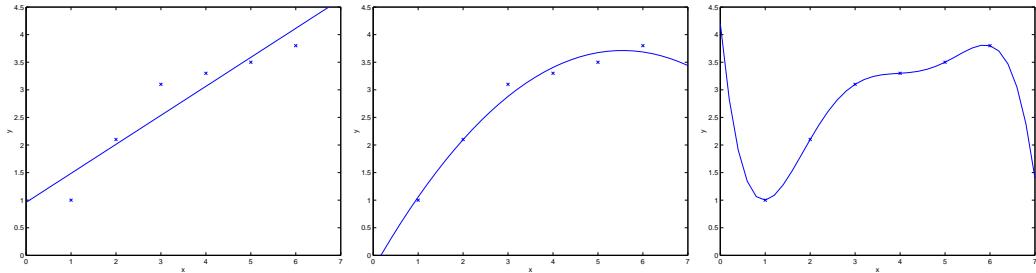
# CS229 Lecture notes

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 “ $y = \theta_0 + \theta_1 x + \dots + \theta_5 x^5$ .” 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 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.<sup>1</sup>

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? This 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

---

<sup>1</sup>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  $A_1, A_2, \dots, A_k$  be  $k$  different events (that may not be independent). Then

$$P(A_1 \cup \dots \cup A_k) \leq P(A_1) + \dots + P(A_k).$$

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 sum of the probabilities of the  $k$  different events.

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

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

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

We assume we are given a training set  $S = \{(x^{(i)}, y^{(i)}); i = 1, \dots, n\}$  of size  $n$ , 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

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

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.<sup>2</sup>

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  $\theta$ ? 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,  $\mathcal{H} = \{h_\theta : h_\theta(x) = 1\{\theta^T x \geq 0\}, \theta \in \mathbb{R}^{d+1}\}$  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:

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

---

<sup>2</sup>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  $\mathcal{H} = \{h_1, \dots, h_k\}$  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

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

Thus,  $\hat{\varepsilon}(h_i)$  is exactly the mean of the  $n$  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

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

This shows that, for our particular  $h_i$ , training error will be close to generalization error with high probability, assuming  $n$  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 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 shown that, for any particular  $A_i$ , it holds true

that  $P(A_i) \leq 2 \exp(-2\gamma^2 n)$ . Thus, using the union bound, we have that

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

If we subtract both sides from 1, we find that

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

(The “ $\neg$ ” symbol means “not.”) So, with probability at least  $1 - 2k \exp(-2\gamma^2 n)$ , we have that  $\varepsilon(h)$  will be within  $\gamma$  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  $n$  and  $\gamma$ , 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:  $n$ ,  $\gamma$ , 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  $\gamma$  and some  $\delta > 0$ , how large must  $n$  be before we can guarantee that with probability at least  $1 - \delta$ , training error will be within  $\gamma$  of generalization error? By setting  $\delta = 2k \exp(-2\gamma^2 n)$  and solving for  $n$ , [you should convince yourself this is the right thing to do!], we find that if

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

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  $\delta$ .) This bound tells us how many training examples we need in order to make a guarantee. The training set size  $n$  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  $n$  and  $\delta$  fixed and solve for  $\gamma$  in the previous equation, and show [again, convince yourself that this is right!] that with probability  $1 - \delta$ , we have that for all  $h \in \mathcal{H}$ ,

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

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  $2\gamma$  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  $n, \delta$  be fixed. Then with probability at least  $1 - \delta$ , we have that

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

This is proved by letting  $\gamma$  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  $2\gamma$  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  $\gamma$  and  $\delta$  fixed and solving for  $n$  like we did before, we can also obtain the following sample complexity bound:

**Corollary.** Let  $|\mathcal{H}| = k$ , and let any  $\delta, \gamma$  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

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

## 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  $n \geq O\left(\frac{1}{\gamma^2} \log \frac{2^{64d}}{\delta}\right) = O\left(\frac{d}{\gamma^2} \log \frac{1}{\delta}\right) = O_{\gamma, \delta}(d)$ . (The  $\gamma, \delta$  subscripts indicate that the last big- $O$  is hiding constants that may depend on  $\gamma$  and  $\delta$ .) 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 try to do is 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.)

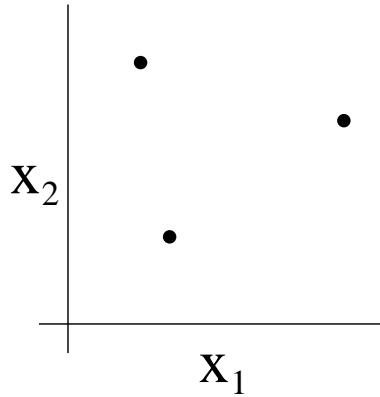
The other part of our previous argument that's slightly unsatisfying is that it relies on the parameterization of  $\mathcal{H}$ . Intuitively, this doesn't seem like it should matter: We had written the class of linear classifiers as  $h_\theta(x) = 1\{\theta_0 + \theta_1x_1 + \dots + \theta_dx_d \geq 0\}$ , with  $n+1$  parameters  $\theta_0, \dots, \theta_d$ . But it could also be written  $h_{u,v}(x) = 1\{(u_0^2 - v_0^2) + (u_1^2 - v_1^2)x_1 + \dots + (u_d^2 - v_d^2)x_d \geq 0\}$  with  $2d+2$  parameters  $u_i, v_i$ . Yet, both of these are just defining the same  $\mathcal{H}$ : The set of linear classifiers in  $d$  dimensions.

To derive a more satisfying argument, let's define a few more things.

Given a set  $S = \{x^{(1)}, \dots, x^{(\mathbf{D})}\}$  (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  $\{y^{(1)}, \dots, y^{(\mathbf{D})}\}$ , there exists some  $h \in \mathcal{H}$  so that  $h(x^{(i)}) = y^{(i)}$  for all  $i = 1, \dots, \mathbf{D}$ .

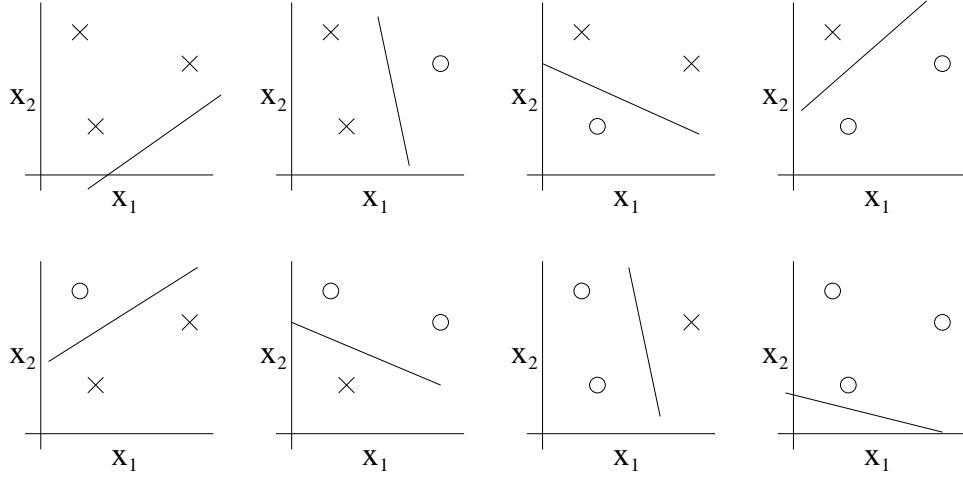
Given a hypothesis class  $\mathcal{H}$ , we then define its **Vapnik-Chervonenkis dimension**, written  $\text{VC}(\mathcal{H})$ , to be the size of the largest set that is shattered by  $\mathcal{H}$ . (If  $\mathcal{H}$  can shatter arbitrarily large sets, then  $\text{VC}(\mathcal{H}) = \infty$ .)

For instance, consider the following set of three points:



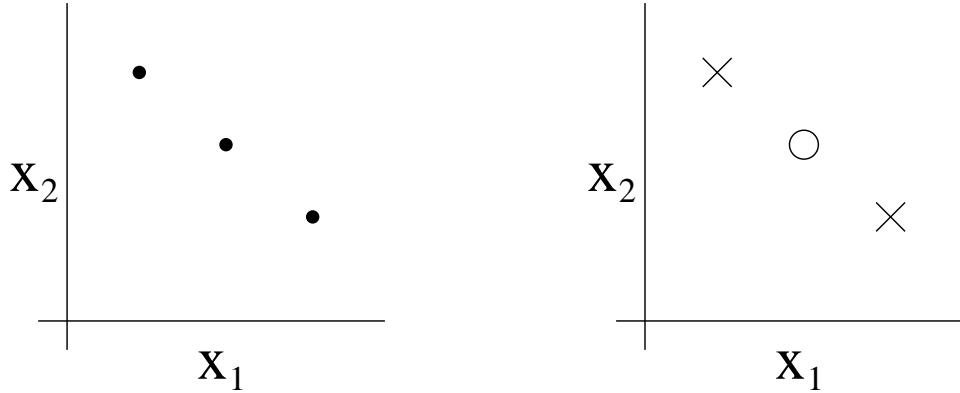
Can the set  $\mathcal{H}$  of linear classifiers in two dimensions ( $h(x) = 1\{\theta_0 + \theta_1x_1 + \theta_2x_2 \geq 0\}$ ) can shatter the set above? The answer is yes. Specifically, we

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  $VC(\mathcal{H}) = 3$ .

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 other words, under the definition of the VC dimension, in order to prove that  $VC(\mathcal{H})$  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  $\mathbf{D} = \text{VC}(\mathcal{H})$ . Then with probability at least  $1 - \delta$ , we have that for all  $h \in \mathcal{H}$ ,

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

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

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

In other words, if a hypothesis class has finite VC dimension, then uniform convergence occurs as  $n$  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  $n = O_{\gamma, \delta}(\mathbf{D})$ .

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 a given hypothesis class  $\mathcal{H}$  (and for an algorithm that tries to minimize training error), the number of training examples needed to achieve generalization error close to that of the optimal classifier is usually roughly linear in the number of parameters of  $\mathcal{H}$ .

# CS229 Lecture notes

Andrew Ng

## Part VI

# Regularization and model selection

Suppose we are trying select among several different models for a learning problem. For instance, we might be using a polynomial regression model  $h_\theta(x) = g(\theta_0 + \theta_1x + \theta_2x^2 + \dots + \theta_kx^k)$ , 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<sup>1</sup>? Alternatively, suppose we want to automatically choose the bandwidth parameter  $\tau$  for locally weighted regression, or the parameter  $C$  for our  $\ell_1$ -regularized SVM. How can we do that?

For the sake of concreteness, in these notes we assume we have some finite set of models  $\mathcal{M} = \{M_1, \dots, M_d\}$  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.<sup>2</sup>) Alternatively, if we are trying to decide between using an SVM, a neural network or logistic regression, then  $\mathcal{M}$  may contain these models.

---

<sup>1</sup>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.

<sup>2</sup>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  $\tau$  and consider only a finite number of possible values for it. More generally, most of the algorithms described here can all be viewed as performing optimization search in the space of models, and we can perform this search over infinite model classes as well.

## 1 Cross validation

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

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

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

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

1. Randomly split  $S$  into  $S_{\text{train}}$  (say, 70% of the data) and  $S_{\text{cv}}$  (the remaining 30%). Here,  $S_{\text{cv}}$  is called the hold-out cross validation set.
2. Train each model  $M_i$  on  $S_{\text{train}}$  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  $S_{\text{cv}}$ .)

By testing on a set of examples  $S_{\text{cv}}$  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  $S_{\text{train}}$  does not necessarily mean it will also do well on  $S_{\text{cv}}$ , 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  $n$  training examples, since we're testing models that were trained on only  $0.7m$  examples each time. While this is fine if data is abundant and/or cheap, in learning problems in which data is scarce (consider a problem with  $m = 20$ , say), we'd like to do something better.

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

1. Randomly split  $S$  into  $k$  disjoint subsets of  $m/k$  training examples each. Lets call these subsets  $S_1, \dots, S_k$ .
2. For each model  $M_i$ , we evaluate it as follows:

For  $j = 1, \dots, k$

Train the model  $M_i$  on  $S_1 \cup \dots \cup S_{j-1} \cup S_{j+1} \cup \dots \cup S_k$  (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  $\hat{\epsilon}_{S_j}(h_{ij})$ .

The estimated generalization error of model  $M_i$  is then calculated as the average of the  $\hat{\epsilon}_{S_j}(h_{ij})$ '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  $d$  is very large (perhaps  $n \gg d$ ), but you suspect that there is only a small number of features that are “relevant” to the learning task. Even if you use the a simple linear classifier (such as the perceptron) over the  $d$  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  $d$  features, there are  $2^d$  possible feature subsets (since each of the  $d$  features can either be included or excluded from the subset), and thus feature selection can be posed as a model selection problem over  $2^d$  possible models. For large values of  $d$ , it’s usually too expensive to explicitly enumerate over and compare all  $2^d$  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  $i = 1, \dots, d$  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  $\mathcal{F} = \{1, \dots, d\}$  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  $\mathcal{F} = \{1, \dots, d\}$  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  $\mathcal{F} = \{1, \dots, d\}$ ) 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**  $\text{MI}(x_i, y)$  between  $x_i$  and  $y$ :

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

(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:

$$\text{MI}(x_i, y) = \text{KL}(p(x_i, y) || p(x_i)p(y))$$

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  $\text{MI}(x_i, y)$  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  $d$ , 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 estimation (MLE), and chose our parameters according to

$$\theta_{\text{MLE}} = \arg \max_{\theta} \prod_{i=1}^n p(y^{(i)} | x^{(i)}; \theta).$$

Throughout our subsequent discussions, we viewed  $\theta$  as an unknown parameter of the world. This view of the  $\theta$  as being *constant-valued but unknown* is taken in **frequentist** statistics. In the frequentist this view of the world,  $\theta$  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  $\theta$  as being a *random variable* whose value is unknown. In this approach, we would specify a **prior distribution**  $p(\theta)$  on  $\theta$  that expresses our “prior beliefs” about the parameters. Given a training set  $S = \{(x^{(i)}, y^{(i)})\}_{i=1}^n$ , when we are asked to make a prediction on a new value of  $x$ , we can then compute the posterior

distribution on the parameters

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

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  $h_{\theta}(x^{(i)}) = 1/(1 + \exp(-\theta^T x^{(i)}))$ .<sup>3</sup>

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  $\theta$ :

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

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

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

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  $\theta$ . 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)  $\theta$  as in Equation (1), and this typically cannot be done in closed-form.

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

$$\theta_{\text{MAP}} = \arg \max_{\theta} \prod_{i=1}^n p(y^{(i)}|x^{(i)}, \theta)p(\theta). \quad (3)$$

---

<sup>3</sup>Since we are now viewing  $\theta$  as a random variable, it is okay to condition on its value, and write “ $p(y|x, \theta)$ ” instead of “ $p(y|x; \theta)$ .”

<sup>4</sup>The integral below would be replaced by a summation if  $y$  is discrete-valued.

Note that this is the same formulas as for the MLE (maximum likelihood) estimate for  $\theta$ , 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  $\theta_{\text{MAP}}$  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  $d \gg n$ .

# CS229 Lecture notes

Andrew Ng

## The $k$ -means clustering algorithm

In the clustering problem, we are given a training set  $\{x^{(1)}, \dots, x^{(n)}\}$ , and want to group the data into a few cohesive “clusters.” Here,  $x^{(i)} \in \mathbb{R}^d$  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**  $\mu_1, \mu_2, \dots, \mu_k \in \mathbb{R}^d$  randomly.
2. Repeat until convergence: {

For every  $i$ , set

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

For each  $j$ , set

$$\mu_j := \frac{\sum_{i=1}^n 1\{c^{(i)} = j\} x^{(i)}}{\sum_{i=1}^n 1\{c^{(i)} = j\}}.$$

}

In the algorithm above,  $k$  (a parameter of the algorithm) is the number of clusters we want to find; and the cluster centroids  $\mu_j$  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  $\mu_j$ , and (ii) Moving each cluster centroid  $\mu_j$  to the mean of the points assigned to it. Figure 1 shows an illustration of running  $k$ -means.

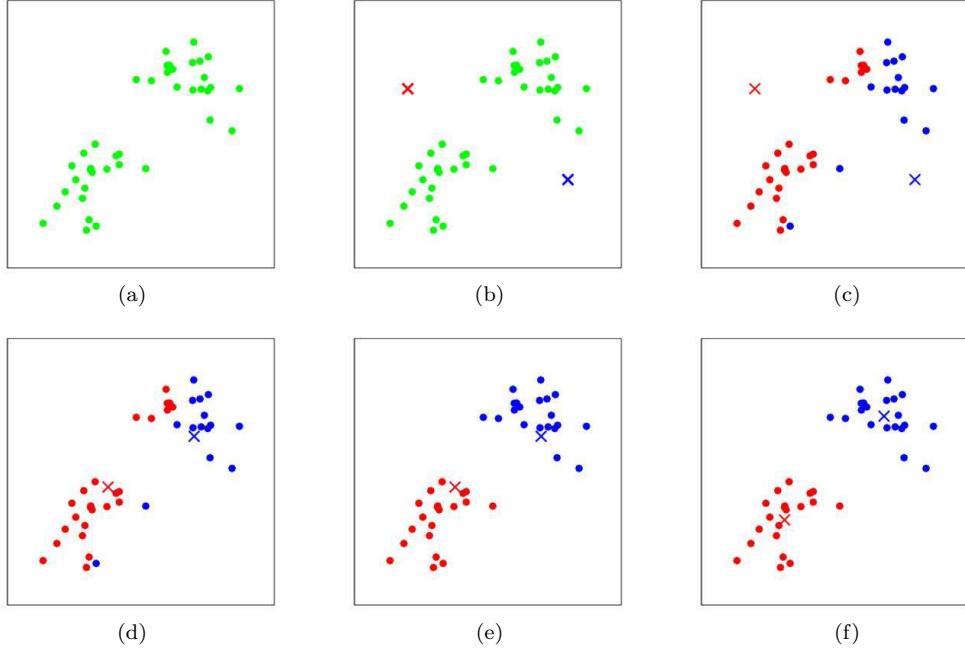


Figure 1: K-means algorithm. Training examples are shown as dots, and cluster centroids are shown as crosses. (a) Original dataset. (b) Random initial cluster centroids (in this instance, not chosen to be equal to two training examples). (c-f) Illustration of running two iterations of  $k$ -means. In 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:

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

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  $\mu$  fixed, and then minimizes  $J$  with respect to  $\mu$  while holding  $c$  fixed. Thus,  $J$  must monotonically decrease, and the value of  $J$  must converge. (Usually, this implies that  $c$  and  $\mu$  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  $\mu$ —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  $\mu_j$ ). 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) algorithm for density estimation.

Suppose that we are given a training set  $\{x^{(1)}, \dots, x^{(n)}\}$  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  $\phi_j$  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  $\{1, \dots, k\}$ , 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  $\phi$ ,  $\mu$  and  $\Sigma$ . To estimate them, we can write down the likelihood of our data:

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

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

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

Maximizing this with respect to  $\phi$ ,  $\mu$  and  $\Sigma$  gives the parameters:

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

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.<sup>1</sup>

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

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

---

<sup>1</sup>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  $\Sigma_j$  for each Gaussian.

(M-step) Update the parameters:

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

}

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  $\mu_j$  and covariance  $\Sigma_j$  at  $x^{(i)}$ ;  $p(z^{(i)} = j; \phi)$  is given by  $\phi_j$ , and so on. The values  $w_j^{(i)}$  calculated in the E-step represent our “soft” guesses<sup>2</sup> 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

---

<sup>2</sup>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  $\{1, \dots, k\}$ ).

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

Tengyu Ma and Andrew Ng

May 13, 2019

## 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:

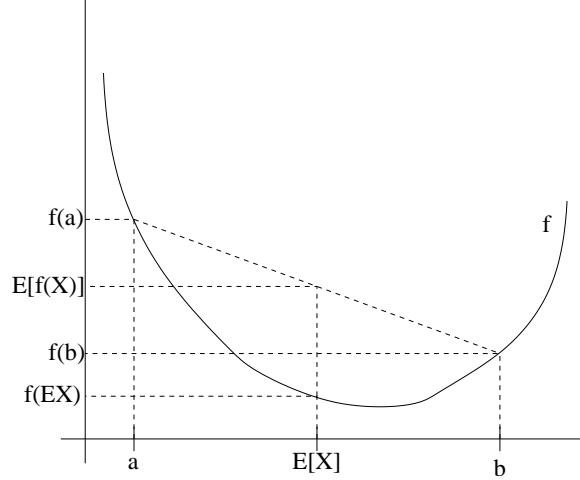
**Theorem.** Let  $f$  be a convex function, and let  $X$  be a random variable. Then:

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

Moreover, if  $f$  is strictly convex, then  $\mathbb{E}[f(X)] = f(\mathbb{E}X)$  holds true if and only if  $X = \mathbb{E}[X]$  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,  $f(\mathbb{E}X) = f(\mathbb{E}[X])$ .

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  $f(E[X])$  indicated on the  $y$ -axis. Moreover, the value  $E[f(X)]$  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  $E[f(X)] \geq f(EX)$ .

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 ( $E[f(X)] \leq f(EX)$ , etc.).

## 2 The EM algorithm

Suppose we have an estimation problem in which we have a training set  $\{x^{(1)}, \dots, x^{(n)}\}$  consisting of  $n$  independent examples. We have a latent variable model  $p(x, z; \theta)$  with  $z$  being the latent variable (which for simplicity is assumed to take finite number of values). The density for  $x$  can be obtained by marginalized over the latent variable  $z$ :

$$p(x; \theta) = \sum_z p(x, z; \theta) \quad (1)$$

We wish to fit the parameters  $\theta$  by maximizing the log-likelihood of the data, defined by

$$\ell(\theta) = \sum_{i=1}^n \log p(x^{(i)}; \theta) \quad (2)$$

We can rewrite the objective in terms of the joint density  $p(x, z; \theta)$  by

$$\ell(\theta) = \sum_{i=1}^n \log p(x^{(i)}; \theta) \quad (3)$$

$$= \sum_{i=1}^n \log \sum_{z^{(i)}} p(x^{(i)}, z^{(i)}; \theta). \quad (4)$$

But, explicitly finding the maximum likelihood estimates of the parameters  $\theta$  may be hard since it will result in difficult non-convex optimization problems.<sup>1</sup> 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  $\ell$  (E-step), and then optimize that lower-bound (M-step).<sup>2</sup>

It turns out that the summation  $\sum_{i=1}^n$  is not essential here, and towards a simpler exposition of the EM algorithm, we will first consider optimizing the likelihood  $\log p(x)$  for a **single example**  $x$ . After we derive the algorithm for optimizing  $\log p(x)$ , we will convert it to an algorithm that works for  $n$  examples by adding back the sum to each of the relevant equations. Thus, now we aim to optimize  $\log p(x; \theta)$  which can be rewritten as

$$\log p(x; \theta) = \log \sum_z p(x, z; \theta) \quad (5)$$

---

<sup>1</sup>It's mostly an empirical observation that the optimization problem is difficult to optimize.

<sup>2</sup>Empirically, the E-step and M-step can often be computed more efficiently than optimizing the function  $\ell(\cdot)$  directly. However, it doesn't necessarily mean that alternating the two steps can always converge to the global optimum of  $\ell(\cdot)$ . Even for mixture of Gaussians, the EM algorithm can either converge to a global optimum or get stuck, depending on the properties of the training data. Empirically, for real-world data, often EM can converge to a solution with relatively high likelihood (if not the optimum), and the theory behind it is still largely not understood.

Let  $Q$  be a distribution over the possible values of  $z$ . That is,  $\sum_z Q(z) = 1$ ,  $Q(z) \geq 0$ .

Consider the following:<sup>3</sup>

$$\begin{aligned} \log p(x; \theta) &= \log \sum_z p(x, z; \theta) \\ &= \log \sum_z Q(z) \frac{p(x, z; \theta)}{Q(z)} \end{aligned} \quad (6)$$

$$\geq \sum_z Q(z) \log \frac{p(x, z; \theta)}{Q(z)} \quad (7)$$

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

$$\sum_z Q(z) \left[ \frac{p(x, z; \theta)}{Q(z)} \right]$$

in the summation is just an expectation of the quantity  $[p(x, z; \theta)/Q(z)]$  with respect to  $z$  drawn according to the distribution given by  $Q$ .<sup>4</sup> By Jensen's inequality, we have

$$f \left( \mathbb{E}_{z \sim Q} \left[ \frac{p(x, z; \theta)}{Q(z)} \right] \right) \geq \mathbb{E}_{z \sim Q} \left[ f \left( \frac{p(x, z; \theta)}{Q(z)} \right) \right],$$

where the “ $z \sim Q$ ” subscripts above indicate that the expectations are with respect to  $z$  drawn from  $Q$ . This allowed us to go from Equation (6) to Equation (7).

Now, for **any** distribution  $Q$ , the formula (7) gives a lower-bound on  $\log p(x; \theta)$ . There are many possible choices for the  $Q$ 's. Which should we choose? Well, if we have some current guess  $\theta$  of the parameters, it seems natural to try to make the lower-bound tight at that value of  $\theta$ . I.e., we will make the inequality above hold with equality at our particular value of  $\theta$ .

To make the bound tight for a particular value of  $\theta$ , we need for the step involving Jensen's inequality in our derivation above to hold with equality.

---

<sup>3</sup>If  $z$  were continuous, then  $Q$  would be a density, and the summations over  $z$  in our discussion are replaced with integrals over  $z$ .

<sup>4</sup>We note that the notion  $\frac{p(x, z; \theta)}{Q(z)}$  only makes sense if  $Q(z) \neq 0$  whenever  $p(x, z; \theta) \neq 0$ . Here we implicitly assume that we only consider those  $Q$  with such a property.

For this to be true, we know it is sufficient that the expectation be taken over a “constant”-valued random variable. I.e., we require that

$$\frac{p(x, z; \theta)}{Q(z)} = c$$

for some constant  $c$  that does not depend on  $z$ . This is easily accomplished by choosing

$$Q(z) \propto p(x, z; \theta).$$

Actually, since we know  $\sum_z Q(z) = 1$  (because it is a distribution), this further tells us that

$$\begin{aligned} Q(z) &= \frac{p(x, z; \theta)}{\sum_z p(x, z; \theta)} \\ &= \frac{p(x, z; \theta)}{p(x; \theta)} \\ &= p(z|x; \theta) \end{aligned} \tag{8}$$

Thus, we simply set the  $Q$ ’s to be the posterior distribution of the  $z$ ’s given  $x$  and the setting of the parameters  $\theta$ .

Indeed, we can directly verify that when  $Q(z) = p(z|x; \theta)$ , then equation (7) is an equality because

$$\begin{aligned} \sum_z Q(z) \log \frac{p(x, z; \theta)}{Q(z)} &= \sum_z p(z|x; \theta) \log \frac{p(x, z; \theta)}{p(z|x; \theta)} \\ &= \sum_z p(z|x; \theta) \log \frac{p(z|x; \theta)p(x; \theta)}{p(z|x; \theta)} \\ &= \sum_z p(z|x; \theta) \log p(x; \theta) \\ &= \log p(x; \theta) \sum_z p(z|x; \theta) \\ &= \log p(x; \theta) \quad (\text{because } \sum_z p(z|x; \theta) = 1) \end{aligned}$$

For convenience, we call the expression in Equation (7) the **evidence lower bound** (ELBO) and we denote it by

$$\text{ELBO}(x; Q, \theta) = \sum_z Q(z) \log \frac{p(x, z; \theta)}{Q(z)} \tag{9}$$

With this equation, we can re-write equation (7) as

$$\forall Q, \theta, x, \quad \log p(x; \theta) \geq \text{ELBO}(x; Q, \theta) \quad (10)$$

Intuitively, the EM algorithm alternatively updates  $Q$  and  $\theta$  by a) setting  $Q(z) = p(z|x; \theta)$  following Equation (8) so that  $\text{ELBO}(x; Q, \theta) = \log p(x; \theta)$  for  $x$  and the current  $\theta$ , and b) maximizing  $\text{ELBO}(x; Q, \theta)$  w.r.t  $\theta$  while fixing the choice of  $Q$ .

Recall that all the discussion above was under the assumption that we aim to optimize the log-likelihood  $\log p(x; \theta)$  for a single example  $x$ . It turns out that with multiple training examples, the basic idea is the same and we only need to take a sum over examples at relevant places. Next, we will build the evidence lower bound for multiple training examples and make the EM algorithm formal.

Recall we have a training set  $\{x^{(1)}, \dots, x^{(n)}\}$ . Note that the optimal choice of  $Q$  is  $p(z|x; \theta)$ , and it depends on the particular example  $x$ . Therefore here we will introduce  $n$  distributions  $Q_1, \dots, Q_n$ , one for each example  $x^{(i)}$ . For each example  $x^{(i)}$ , we can build the evidence lower bound

$$\log p(x^{(i)}; \theta) \geq \text{ELBO}(x^{(i)}; Q_i, \theta) = \sum_{z^{(i)}} Q_i(z^{(i)}) \log \frac{p(x^{(i)}, z^{(i)}; \theta)}{Q_i(z^{(i)})}$$

Taking sum over all the examples, we obtain a lower bound for the log-likelihood

$$\begin{aligned} \ell(\theta) &\geq \sum_i \text{ELBO}(x^{(i)}; Q_i, \theta) \\ &= \sum_i \sum_{z^{(i)}} Q_i(z^{(i)}) \log \frac{p(x^{(i)}, z^{(i)}; \theta)}{Q_i(z^{(i)})} \end{aligned} \quad (11)$$

For *any* set of distributions  $Q_1, \dots, Q_n$ , the formula (11) gives a lower-bound on  $\ell(\theta)$ , and analogous to the argument around equation (8), the  $Q_i$  that attains equality satisfies

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

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

Now, for this choice of the  $Q_i$ 's, Equation (11) gives a lower-bound on the loglikelihood  $\ell$  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 (11) with respect to the parameters to obtain a new setting of the  $\theta$ '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

$$\begin{aligned} \theta &:= \arg \max_{\theta} \sum_{i=1}^n \text{ELBO}(x^{(i)}; Q_i, \theta) \\ &= \arg \max_{\theta} \sum_i \sum_{z^{(i)}} Q_i(z^{(i)}) \log \frac{p(x^{(i)}, z^{(i)}; \theta)}{Q_i(z^{(i)})}. \end{aligned} \quad (12)$$

}

How do 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 (11), holds with equality, and hence

$$\ell(\theta^{(t)}) = \sum_{i=1}^n \text{ELBO}(x^{(i)}; Q_i^{(t)}, \theta^{(t)}) \quad (13)$$

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

$$\begin{aligned} \ell(\theta^{(t+1)}) &\geq \sum_{i=1}^n \text{ELBO}(x^{(i)}; Q_i^{(t)}, \theta^{(t+1)}) \\ &\quad (\text{because inequality (11) holds for all } Q \text{ and } \theta) \\ &\geq \sum_{i=1}^n \text{ELBO}(x^{(i)}; Q_i^{(t)}, \theta^{(t)}) \quad (\text{see reason below}) \\ &= \ell(\theta^{(t)}) \quad (\text{by equation (13)}) \end{aligned}$$

where the last inequality follows from that  $\theta^{(t+1)}$  is chosen explicitly to be

$$\arg \max_{\theta} \sum_{i=1}^n \text{ELBO}(x^{(i)}; Q_i^{(t)}, \theta)$$

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 (by overloading  $\text{ELBO}(\cdot)$ )

$$\text{ELBO}(Q, \theta) = \sum_{i=1}^n \text{ELBO}(x^{(i)}; Q_i, \theta) = \sum_i \sum_{z^{(i)}} Q_i(z^{(i)}) \log \frac{p(x^{(i)}, z^{(i)}; \theta)}{Q_i(z^{(i)})} \quad (14)$$

then we know  $\ell(\theta) \geq \text{ELBO}(Q, \theta)$  from our previous derivation. The EM can also be viewed an alternating maximization algorithm on  $\text{ELBO}(Q, \theta)$ , in which the E-step maximizes it with respect to  $Q$  (check this yourself), and the M-step maximizes it with respect to  $\theta$ .

## 2.1 Other interpretation of ELBO

Let  $\text{ELBO}(x; Q, \theta) = \sum_z Q(z) \log \frac{p(x, z; \theta)}{Q(z)}$  be defined as in equation (9). There are several other forms of ELBO. First, we can rewrite

$$\begin{aligned} \text{ELBO}(x; Q, \theta) &= \mathbb{E}_{z \sim Q} [\log p(x, z; \theta)] - \mathbb{E}_{z \sim Q} [\log Q(z)] \\ &= \mathbb{E}_{z \sim Q} [\log p(x|z; \theta)] - D_{KL}(Q \| p_z) \end{aligned} \quad (15)$$

where we use  $p_z$  to denote the marginal distribution of  $z$  (under the distribution  $p(x, z; \theta)$ ), and  $D_{KL}()$  denotes the KL divergence

$$D_{KL}(Q \| p_z) = \sum_z Q(z) \log \frac{Q(z)}{p(z)} \quad (16)$$

In many cases, the marginal distribution of  $z$  does not depend on the parameter  $\theta$ . In this case, we can see that maximizing ELBO over  $\theta$  is equivalent to maximizing the first term in (15). This corresponds to maximizing the conditional likelihood of  $x$  conditioned on  $z$ , which is often a simpler question than the original question.

Another form of  $\text{ELBO}(\cdot)$  is (please verify yourself)

$$\text{ELBO}(x; Q, \theta) = \log p(x) - D_{KL}(Q \| p_{z|x}) \quad (17)$$

where  $p_{z|x}$  is the conditional distribution of  $z$  given  $x$  under the parameter  $\theta$ . This forms shows that the maximizer of  $\text{ELBO}(Q, \theta)$  over  $Q$  is obtained when  $Q = p_{z|x}$ , which was shown in equation (8) before.

### 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  $\phi$ ,  $\mu$  and  $\Sigma$  in a mixture of Gaussians. For the sake of brevity, we carry out the derivations for the M-step updates only for  $\phi$  and  $\mu_j$ , and leave the updates for  $\Sigma_j$  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  $\phi, \mu, \Sigma$ , the quantity

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

Let's maximize this with respect to  $\mu_l$ . If we take the derivative with respect to  $\mu_l$ , we find

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

Setting this to zero and solving for  $\mu_l$  therefore yields the update rule

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

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  $\phi_j$ . Grouping together only the terms that depend on  $\phi_j$ , we find that we need to maximize

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

However, there is an additional constraint that the  $\phi_j$ '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

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

where  $\beta$  is the Lagrange multiplier.<sup>5</sup> Taking derivatives, we find

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

Setting this to zero and solving, we get

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

I.e.,  $\phi_j \propto \sum_{i=1}^n w_j^{(i)}$ . Using the constraint that  $\sum_j \phi_j = 1$ , we easily find that  $-\beta = \sum_{i=1}^n \sum_{j=1}^k w_j^{(i)} = \sum_{i=1}^n 1 = n$ . (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  $\phi_j$ :

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

The derivation for the M-step updates to  $\Sigma_j$  are also entirely straightforward.

---

<sup>5</sup>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.

## 4 Variational inference and variational auto-encoder

Loosely speaking, variational auto-encoder [2] generally refers to a family of algorithms that extend the EM algorithms to more complex models parameterized by neural networks. It extends the technique of variational inference with the additional “re-parametrization trick” which will be introduced below. Variational auto-encoder may not give the best performance for many datasets, but it contains several central ideas about how to extend EM algorithms to high-dimensional continuous latent variables with non-linear models. Understanding it will likely give you the language and backgrounds to understand various recent papers related to it.

As a running example, we will consider the following parameterization of  $p(x, z; \theta)$  by a neural network. Let  $\theta$  be the collection of the weights of a neural network  $g(z; \theta)$  that maps  $z \in \mathbb{R}^k$  to  $\mathbb{R}^d$ . Let

$$z \sim \mathcal{N}(0, I_{k \times k}) \quad (18)$$

$$x|z \sim \mathcal{N}(g(z; \theta), \sigma^2 I_{d \times d}) \quad (19)$$

Here  $I_{k \times k}$  denotes identity matrix of dimension  $k$  by  $k$ , and  $\sigma$  is a scalar that we assume to be known for simplicity.

For the Gaussian mixture models in Section 3, the optimal choice of  $Q(z) = p(z|x; \theta)$  for each fixed  $\theta$ , that is the posterior distribution of  $z$ , can be analytically computed. In many more complex models such as the model (19), it’s intractable to compute the exact the posterior distribution  $p(z|x; \theta)$ .

Recall that from equation (10), ELBO is always a lower bound for any choice of  $Q$ , and therefore, we can also aim for finding an **approximation** of the true posterior distribution. Often, one has to use some particular form to approximate the true posterior distribution. Let  $\mathcal{Q}$  be a family of  $Q$ ’s that we are considering, and we will aim to find a  $Q$  within the family of  $\mathcal{Q}$  that is closest to the true posterior distribution. To formalize, recall the definition of the ELBO lower bound as a function of  $Q$  and  $\theta$  defined in equation (14)

$$\text{ELBO}(Q, \theta) = \sum_{i=1}^n \text{ELBO}(x^{(i)}; Q_i, \theta) = \sum_i \sum_{z^{(i)}} Q_i(z^{(i)}) \log \frac{p(x^{(i)}, z^{(i)}; \theta)}{Q_i(z^{(i)})}$$

Recall that EM can be viewed as alternating maximization of  $\text{ELBO}(Q, \theta)$ . Here instead, we optimize the EBLO over  $Q \in \mathcal{Q}$

$$\max_{Q \in \mathcal{Q}} \max_{\theta} \text{ELBO}(Q, \theta) \quad (20)$$

Now the next question is what form of  $Q$  (or what structural assumptions to make about  $Q$ ) allows us to efficiently maximize the objective above. When the latent variable  $z$  are high-dimensional discrete variables, one popular assumption is the **mean field assumption**, which assumes that  $Q_i(z)$  gives a distribution with independent coordinates, or in other words,  $Q_i$  can be decomposed into  $Q_i(z) = Q_i^1(z_1) \cdots Q_i^k(z_k)$ . There are tremendous applications of mean field assumptions to learning generative models with discrete latent variables, and we refer to [1] for a survey of these models and their impact to a wide range of applications including computational biology, computational neuroscience, social sciences. We will not get into the details about the discrete latent variable cases, and our main focus is to deal with continuous latent variables, which requires not only mean field assumptions, but additional techniques.

When  $z \in \mathbb{R}^k$  is a continuous latent variable, there are several decisions to make towards successfully optimizing (20). First we need to give a succinct representation of the distribution  $Q_i$  because it is over an infinite number of points. A natural choice is to assume  $Q_i$  is a Gaussian distribution with some mean and variance. We would also like to have more succinct representation of the means of  $Q_i$  of all the examples. Note that  $Q_i(z^{(i)})$  is supposed to approximate  $p(z^{(i)}|x^{(i)}; \theta)$ . It would make sense let all the means of the  $Q_i$ 's be some function of  $x^{(i)}$ . Concretely, let  $q(\cdot; \phi), v(\cdot; \psi)$  be two functions that map from dimension  $d$  to  $k$ , which are parameterized by  $\phi$  and  $\psi$ , we assume that

$$Q_i = \mathcal{N}(q(x^{(i)}; \phi), \text{diag}(v(x^{(i)}; \psi))^2) \quad (21)$$

Here  $\text{diag}(w)$  means the  $k \times k$  matrix with the entries of  $w \in \mathbb{R}^k$  on the diagonal. In other words, the distribution  $Q_i$  is assumed to be a Gaussian distribution with independent coordinates, and the mean and standard deviations are governed by  $q$  and  $v$ . Often in variational auto-encoder,  $q$  and  $v$  are chosen to be neural networks.<sup>6</sup> In recent deep learning literature, often  $q, v$  are called **encoder** (in the sense of encoding the data into latent code), whereas  $g(z; \theta)$  if often referred to as the **decoder**.

We remark that  $Q_i$  of such form in many cases are very far from a good approximation of the true posterior distribution. However, some approximation is necessary for feasible optimization. In fact, the form of  $Q_i$  needs to satisfy other requirements (which happened to be satisfied by the form (21))

Before optimizing the ELBO, let's first verify whether we can efficiently evaluate the value of the ELBO for fixed  $Q$  of the form (21) and  $\theta$ . We

---

<sup>6</sup> $q$  and  $v$  can also share parameters. We sweep this level of details under the rug in this note.

rewrite the ELBO as a function of  $\phi, \psi, \theta$  by

$$\text{ELBO}(\phi, \psi, \theta) = \sum_{i=1}^n \mathbb{E}_{z^{(i)} \sim Q_i} \left[ \log \frac{p(x^{(i)}, z^{(i)}; \theta)}{Q_i(z^{(i)})} \right], \quad (22)$$

where  $Q_i = \mathcal{N}(q(x^{(i)}; \phi), \text{diag}(v(x^{(i)}; \psi))^2)$

Note that to evaluate  $Q_i(z^{(i)})$  inside the expectation, we should be able **to compute the density of  $Q_i$** . To estimate the expectation  $\mathbb{E}_{z^{(i)} \sim Q_i}$ , we should be able **to sample from distribution  $Q_i$**  so that we can build an empirical estimator with samples. It happens that for Gaussian distribution  $Q_i = \mathcal{N}(q(x^{(i)}; \phi), \text{diag}(v(x^{(i)}; \psi))^2)$ , we are able to be both efficiently.

Now let's optimize the ELBO. It turns out that we can run gradient ascent over  $\phi, \psi, \theta$  instead of alternating maximization. There is no strong need to compute the maximum over each variable at a much greater cost. (For Gaussian mixture model in Section 3, computing the maximum is analytically feasible and relatively cheap, and therefore we did alternating maximization.) Mathematically, let  $\eta$  be the learning rate, the gradient ascent step is

$$\begin{aligned} \theta &:= \theta + \eta \nabla_\theta \text{ELBO}(\phi, \psi, \theta) \\ \phi &:= \phi + \eta \nabla_\phi \text{ELBO}(\phi, \psi, \theta) \\ \psi &:= \psi + \eta \nabla_\psi \text{ELBO}(\phi, \psi, \theta) \end{aligned}$$

Computing the gradient over  $\theta$  is simple because

$$\begin{aligned} \nabla_\theta \text{ELBO}(\phi, \psi, \theta) &= \nabla_\theta \sum_{i=1}^n \mathbb{E}_{z^{(i)} \sim Q_i} \left[ \frac{\log p(x^{(i)}, z^{(i)}; \theta)}{Q_i(z^{(i)})} \right] \\ &= \nabla_\theta \sum_{i=1}^n \mathbb{E}_{z^{(i)} \sim Q_i} [\log p(x^{(i)}, z^{(i)}; \theta)] \\ &= \sum_{i=1}^n \mathbb{E}_{z^{(i)} \sim Q_i} [\nabla_\theta \log p(x^{(i)}, z^{(i)}; \theta)], \end{aligned} \quad (23)$$

But computing the gradient over  $\phi$  and  $\psi$  is tricky because the sampling distribution  $Q_i$  depends on  $\phi$  and  $\psi$ . (Abstractly speaking, the issue we face can be simplified as the problem of computing the gradient  $\mathbb{E}_{z \sim Q_\phi}[f(\phi)]$  with respect to variable  $\phi$ . We know that in general,  $\nabla \mathbb{E}_{z \sim Q_\phi}[f(\phi)] \neq \mathbb{E}_{z \sim Q_\phi}[\nabla f(\phi)]$  because the dependency of  $Q_\phi$  on  $\phi$  has to be taken into account as well. )

The idea that comes to rescue is the so-called **re-parameterization trick**: we rewrite  $z^{(i)} \sim Q_i = \mathcal{N}(q(x^{(i)}; \phi), \text{diag}(v(x^{(i)}; \psi))^2)$  in an equivalent

way:

$$z^{(i)} = q(x^{(i)}; \phi) + v(x^{(i)}; \psi) \odot \xi^{(i)} \text{ where } \xi^{(i)} \sim \mathcal{N}(0, I_{k \times k}) \quad (24)$$

Here  $x \odot y$  denotes the entry-wise product of two vectors of the same dimension. Here we used the fact that  $x \sim N(\mu, \sigma^2)$  is equivalent to that  $x = \mu + \xi\sigma$  with  $\xi \sim N(0, 1)$ . We mostly just used this fact in every dimension simultaneously for the random variable  $z^{(i)} \sim Q_i$ .

With this re-parameterization, we have that

$$\begin{aligned} & \mathbb{E}_{z^{(i)} \sim Q_i} \left[ \log \frac{p(x^{(i)}, z^{(i)}; \theta)}{Q_i(z^{(i)})} \right] \\ &= \mathbb{E}_{\xi^{(i)} \sim \mathcal{N}(0, 1)} \left[ \log \frac{p(x^{(i)}, q(x^{(i)}; \phi) + v(x^{(i)}; \psi) \odot \xi^{(i)}; \theta)}{Q_i(q(x^{(i)}; \phi) + v(x^{(i)}; \psi) \odot \xi^{(i)})} \right] \end{aligned} \quad (25)$$

It follows that

$$\begin{aligned} & \nabla_\phi \mathbb{E}_{z^{(i)} \sim Q_i} \left[ \log \frac{p(x^{(i)}, z^{(i)}; \theta)}{Q_i(z^{(i)})} \right] \\ &= \nabla_\phi \mathbb{E}_{\xi^{(i)} \sim \mathcal{N}(0, 1)} \left[ \log \frac{p(x^{(i)}, q(x^{(i)}; \phi) + v(x^{(i)}; \psi) \odot \xi^{(i)}; \theta)}{Q_i(q(x^{(i)}; \phi) + v(x^{(i)}; \psi) \odot \xi^{(i)})} \right] \\ &= \mathbb{E}_{\xi^{(i)} \sim \mathcal{N}(0, 1)} \left[ \nabla_\phi \log \frac{p(x^{(i)}, q(x^{(i)}; \phi) + v(x^{(i)}; \psi) \odot \xi^{(i)}; \theta)}{Q_i(q(x^{(i)}; \phi) + v(x^{(i)}; \psi) \odot \xi^{(i)})} \right] \end{aligned}$$

We can now sample multiple copies of  $\xi^{(i)}$ 's to estimate the expectation in the RHS of the equation above.<sup>7</sup> We can estimate the gradient with respect to  $\psi$  similarly, and with these, we can implement the gradient ascent algorithm to optimize the ELBO over  $\phi, \psi, \theta$ .

There are not many high-dimensional distributions with analytically computable density function known to be re-parameterizable. We refer to [2] for a few other choices that can replace Gaussian distribution.

## References

- [1] David M Blei, Alp Kucukelbir, and Jon D McAuliffe. Variational inference: A review for statisticians. *Journal of the American Statistical Association*, 112(518):859–877, 2017.
- [2] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

---

<sup>7</sup>Empirically people sometimes just use one sample to estimate it for maximum computational efficiency.

# CS229 Lecture notes

Andrew Ng

## Part X

# Factor analysis

When we have data  $x^{(i)} \in \mathbb{R}^d$  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  $n$  was significantly larger than the dimension  $d$  of the data.

Now, consider a setting in which  $d \gg n$ . 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  $n$  data points span only a low-dimensional subspace of  $\mathbb{R}^d$ , if we model the data as Gaussian, and estimate the mean and covariance using the usual maximum likelihood estimators,

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

we would find that the matrix  $\Sigma$  is singular. This means that  $\Sigma^{-1}$  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,<sup>1</sup> and this corresponds to a singular covariance matrix.

---

<sup>1</sup>This is the set of points  $x$  satisfying  $x = \sum_{i=1}^n \alpha_i x^{(i)}$ , for some  $\alpha_i$ 's so that  $\sum_{i=1}^n \alpha_1 = 1$ .

More generally, unless  $n$  exceeds  $d$  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  $\Sigma$  that allow us to fit  $\Sigma$  with small amounts of data but neither 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 conditional distributions of Gaussians. Finally, we present the factor analysis model, and EM for it.

## 1 Restrictions of $\Sigma$

If we do not have sufficient data to fit a full covariance matrix, we may place some restrictions on the space of matrices  $\Sigma$  that we will consider. For instance, we may choose to fit a covariance matrix  $\Sigma$  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  $\Sigma$  satisfying

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

Thus,  $\Sigma_{jj}$  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  $\Sigma$  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  $\sigma^2$  is the parameter under our control. The maximum likelihood estimate of  $\sigma^2$  can be found to be:

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

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

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

However, restricting  $\Sigma$  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  $\Sigma$  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  $\Sigma$  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  $E[x_1] = \mu_1$ , and that  $\text{Cov}(x_1) = E[(x_1 - \mu_1)(x_1 - \mu_1)^T] = \Sigma_{11}$ . To see that the latter is true, note that by definition of the joint covariance of  $x_1$  and  $x_2$ , we have

that

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

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

$$\mu_{1|2} = \mu_1 + \Sigma_{12}\Sigma_{22}^{-1}(x_2 - \mu_2), \quad (1)$$

$$\Sigma_{1|2} = \Sigma_{11} - \Sigma_{12}\Sigma_{22}^{-1}\Sigma_{21}. \quad (2)$$

When we work 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  $\mu \in \mathbb{R}^d$ , the matrix  $\Lambda \in \mathbb{R}^{d \times k}$ , and the diagonal matrix  $\Psi \in \mathbb{R}^{d \times d}$ . The value of  $k$  is usually chosen to be smaller than  $d$ .

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  $d$ -dimensional affine space of  $\mathbb{R}^d$  by computing  $\mu + \Lambda z^{(i)}$ . Lastly,  $x^{(i)}$  is generated by adding covariance  $\Psi$  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  $\epsilon$  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  $\mu_{zx}$  and  $\Sigma$ .

We know that  $E[z] = 0$ , from the fact that  $z \sim \mathcal{N}(0, I)$ . Also, we have that

$$\begin{aligned} E[x] &= E[\mu + \Lambda z + \epsilon] \\ &= \mu + \Lambda E[z] + E[\epsilon] \\ &= \mu. \end{aligned}$$

Putting these together, we obtain

$$\mu_{zx} = \begin{bmatrix} \vec{0} \\ \mu \end{bmatrix}$$

Next, to find  $\Sigma$ , we need to calculate  $\Sigma_{zz} = E[(z - E[z])(z - E[z])^T]$  (the upper-left block of  $\Sigma$ ),  $\Sigma_{zx} = E[(z - E[z])(x - E[x])^T]$  (upper-right block), and  $\Sigma_{xx} = E[(x - E[x])(x - E[x])^T]$  (lower-right block).

Now, since  $z \sim \mathcal{N}(0, I)$ , we easily find that  $\Sigma_{zz} = \text{Cov}(z) = I$ . Also,

$$\begin{aligned} E[(z - E[z])(x - E[x])^T] &= E[z(\mu + \Lambda z + \epsilon - \mu)^T] \\ &= E[zz^T]\Lambda^T + E[z\epsilon^T] \\ &= \Lambda^T. \end{aligned}$$

In the last step, we used the fact that  $E[zz^T] = \text{Cov}(z)$  (since  $z$  has zero mean), and  $E[z\epsilon^T] = E[z]E[\epsilon^T] = 0$  (since  $z$  and  $\epsilon$  are independent, and

hence the expectation of their product is the product of their expectations). Similarly, we can find  $\Sigma_{xx}$  as follows:

$$\begin{aligned} \mathbb{E}[(x - \mathbb{E}[x])(x - \mathbb{E}[x])^T] &= \mathbb{E}[(\mu + \Lambda z + \epsilon - \mu)(\mu + \Lambda z + \epsilon - \mu)^T] \\ &= \mathbb{E}[\Lambda z z^T \Lambda^T + \epsilon z^T \Lambda^T + \Lambda z \epsilon^T + \epsilon \epsilon^T] \\ &= \Lambda \mathbb{E}[z z^T] \Lambda^T + \mathbb{E}[\epsilon \epsilon^T] \\ &= \Lambda \Lambda^T + \Psi. \end{aligned}$$

Putting everything together, we therefore have that

$$\begin{bmatrix} z \\ x \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} \vec{0} \\ \mu \end{bmatrix}, \begin{bmatrix} I & \Lambda^T \\ \Lambda & \Lambda \Lambda^T + \Psi \end{bmatrix}\right). \quad (3)$$

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  $\{x^{(i)}; i = 1, \dots, n\}$ , we can write down the log likelihood of the parameters:

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

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 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

$$\sum_{i=1}^n \int_{z^{(i)}} Q_i(z^{(i)}) \log \frac{p(x^{(i)}, z^{(i)}; \mu, \Lambda, \Psi)}{Q_i(z^{(i)})} dz^{(i)} \quad (4)$$

with respect to the parameters  $\mu, \Lambda, \Psi$ . We will work out only the optimization with respect to  $\Lambda$ , and leave the derivations of the updates for  $\mu$  and  $\Psi$  as an exercise to the reader.

We can simplify Equation (4) as follows:

$$\sum_{i=1}^n \int_{z^{(i)}} Q_i(z^{(i)}) [\log p(x^{(i)}|z^{(i)}; \mu, \Lambda, \Psi) + \log p(z^{(i)}) - \log Q_i(z^{(i)})] dz^{(i)} \quad (5)$$

$$= \sum_{i=1}^n \mathbb{E}_{z^{(i)} \sim Q_i} [\log p(x^{(i)}|z^{(i)}; \mu, \Lambda, \Psi) + \log p(z^{(i)}) - \log Q_i(z^{(i)})] \quad (6)$$

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:

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

Let's maximize this with respect to  $\Lambda$ . Only the last term above depends on  $\Lambda$ . Taking derivatives, and using the facts that  $\text{tr } a = a$  (for  $a \in \mathbb{R}$ ),  $\text{tr } AB = \text{tr } BA$ , and  $\nabla_A \text{tr } ABA^T C = CAB + C^T AB$ , we get:

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

Setting this to zero and simplifying, we get:

$$\sum_{i=1}^n \Lambda E_{z^{(i)} \sim Q_i} [z^{(i)} z^{(i)T}] = \sum_{i=1}^n (x^{(i)} - \mu) E_{z^{(i)} \sim Q_i} [z^{(i)T}].$$

Hence, solving for  $\Lambda$ , we obtain

$$\Lambda = \left( \sum_{i=1}^n (x^{(i)} - \mu) E_{z^{(i)} \sim Q_i} [z^{(i)T}] \right) \left( \sum_{i=1}^n E_{z^{(i)} \sim Q_i} [z^{(i)} z^{(i)T}] \right)^{-1}. \quad (7)$$

It is interesting to note the close relationship between this equation and the normal equation that we'd derived for least squares regression,

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

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  $\Lambda$  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

$$\begin{aligned} E_{z^{(i)} \sim Q_i} [z^{(i)T}] &= \mu_{z^{(i)}|x^{(i)}}^T \\ E_{z^{(i)} \sim Q_i} [z^{(i)} z^{(i)T}] &= \mu_{z^{(i)}|x^{(i)}} \mu_{z^{(i)}|x^{(i)}}^T + \Sigma_{z^{(i)}|x^{(i)}}. \end{aligned}$$

The latter comes from the fact that, for a random variable  $Y$ ,  $\text{Cov}(Y) = E[YY^T] - E[Y]E[Y]^T$ , and hence  $E[YY^T] = E[Y]E[Y]^T + \text{Cov}(Y)$ . Substituting this back into Equation (7), we get the M-step update for  $\Lambda$ :

$$\Lambda = \left( \sum_{i=1}^n (x^{(i)} - \mu) \mu_{z^{(i)}|x^{(i)}}^T \right) \left( \sum_{i=1}^n \mu_{z^{(i)}|x^{(i)}} \mu_{z^{(i)}|x^{(i)}}^T + \Sigma_{z^{(i)}|x^{(i)}} \right)^{-1}. \quad (8)$$

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  $z^{(i)}$  given  $x^{(i)}$ , 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  $\mu$  and  $\Psi$ . It is not hard to show that the first is given by

$$\mu = \frac{1}{n} \sum_{i=1}^n x^{(i)}.$$

Since this doesn't change as the parameters are varied (i.e., unlike the update for  $\Lambda$ , 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  $\Psi$  can be found by calculating

$$\Phi = \frac{1}{n} \sum_{i=1}^n x^{(i)} x^{(i)T} - x^{(i)} \mu_{z^{(i)}|x^{(i)}}^T \Lambda^T - \Lambda \mu_{z^{(i)}|x^{(i)}} x^{(i)T} + \Lambda (\mu_{z^{(i)}|x^{(i)}} \mu_{z^{(i)}|x^{(i)}}^T + \Sigma_{z^{(i)}|x^{(i)}}) \Lambda^T,$$

and setting  $\Psi_{ii} = \Phi_{ii}$  (i.e., letting  $\Psi$  be the diagonal matrix containing only the diagonal entries of  $\Phi$ ).

# CS229 Lecture notes

Andrew Ng

## Part XI

# Principal components analysis

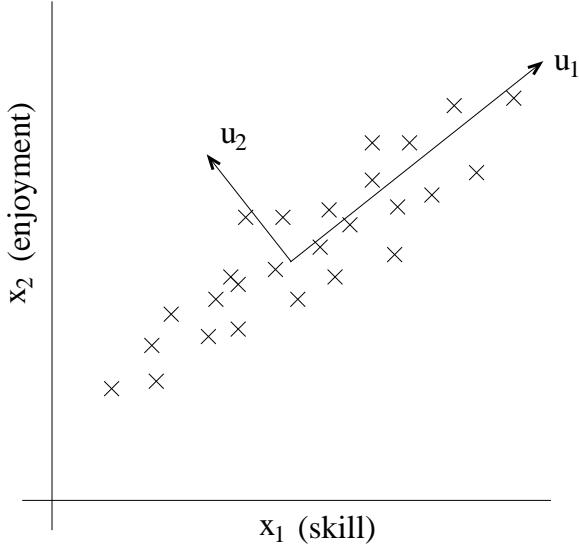
In our discussion of factor analysis, we gave a way to model data  $x \in \mathbb{R}^d$  as “approximately” lying in some  $k$ -dimension subspace, where  $k \ll d$ . Specifically, we imagined that each point  $x^{(i)}$  was created by first generating some  $z^{(i)}$  lying in the  $k$ -dimension affine space  $\{\Lambda z + \mu; z \in \mathbb{R}^k\}$ , and then adding  $\Psi$ -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  $\{x^{(i)}; i = 1, \dots, n\}$  of attributes of  $n$  different types of automobiles, such as their maximum speed, turn radius, and so on. Let  $x^{(i)} \in \mathbb{R}^d$  for each  $i$  ( $d \ll n$ ). 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 preprocess the data by normalizing each feature to have mean 0 and variance 1. We do this by subtracting the mean and dividing by the empirical standard deviation:

$$x_j^{(i)} \leftarrow \frac{x_j^{(i)} - \mu_j}{\sigma_j}$$

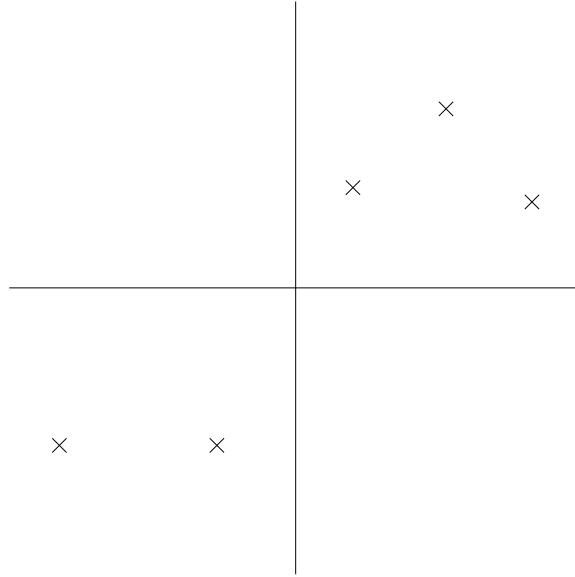
where  $\mu_j = \frac{1}{n} \sum_{i=1}^n x_j^{(i)}$  and  $\sigma_j^2 = \frac{1}{n} \sum_{i=1}^n (x_j^{(i)} - \mu_j)^2$  are the mean variance of feature  $j$ , respectively.

Subtracting  $\mu_j$  zeros out the mean and may be omitted for data known to have zero mean (for instance, time series corresponding to speech or other acoustic signals). Dividing by the standard deviation  $\sigma_j$  rescales 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. This rescaling may be omitted if we had a priori knowledge that the different attributes are all on the same scale. One example of this is if each data point represented a

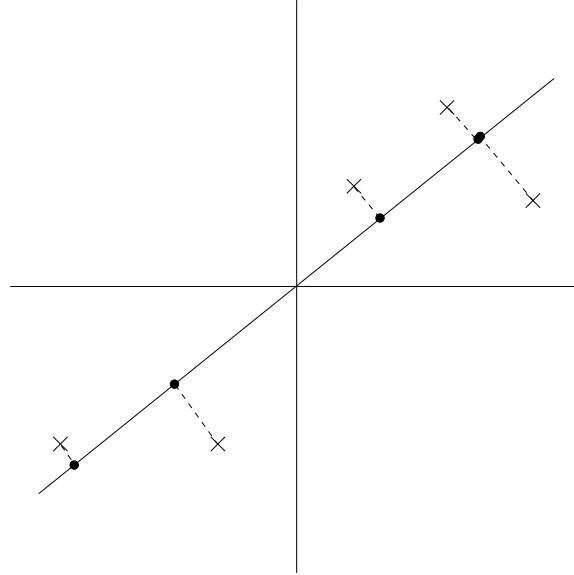
grayscale image, and each  $x_j^{(i)}$  took a value in  $\{0, 1, \dots, 255\}$  corresponding to the intensity value of pixel  $j$  in image  $i$ .

Now, having normalized our data, how do we compute the “major axis of variation”  $u$ —that is, the direction on which the data approximately lies? One way is to pose this problem 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.

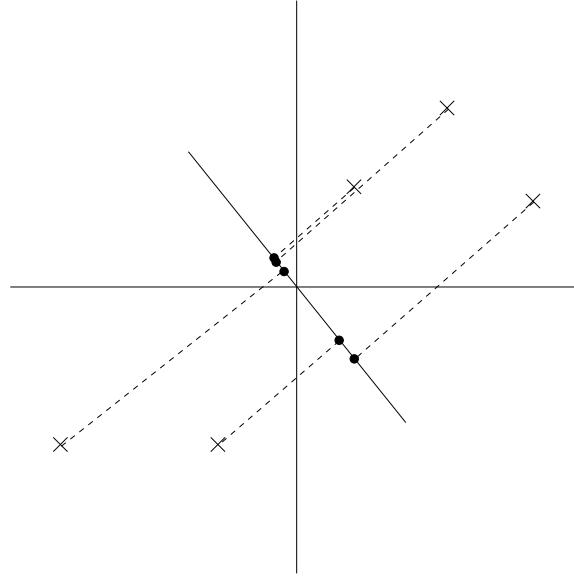
Consider the following dataset, on which we have already carried out the normalization steps:



Now, suppose we pick  $u$  to correspond to 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:

$$\begin{aligned} \frac{1}{n} \sum_{i=1}^n (x^{(i)T} u)^2 &= \frac{1}{n} \sum_{i=1}^n u^T x^{(i)} x^{(i)T} u \\ &= u^T \left( \frac{1}{n} \sum_{i=1}^n x^{(i)} x^{(i)T} \right) u. \end{aligned}$$

We easily recognize that the maximizing this subject to  $\|u\|_2 = 1$  gives the principal eigenvector of  $\Sigma = \frac{1}{n} \sum_{i=1}^n x^{(i)} x^{(i)T}$ , which is just the empirical covariance matrix of the data (assuming it has zero mean).<sup>1</sup>

To summarize, we have found that if we wish to find a 1-dimensional subspace with which to approximate the data, we should choose  $u$  to be the principal eigenvector of  $\Sigma$ . More generally, if we wish to project our data into a  $k$ -dimensional subspace ( $k < d$ ), we should choose  $u_1, \dots, u_k$  to be the top  $k$  eigenvectors of  $\Sigma$ . The  $u_i$ 's now form a new, orthogonal basis for the data.<sup>2</sup>

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  $x^{(i)} \in \mathbb{R}^d$ , 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  $u_1, \dots, u_k$  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

---

<sup>1</sup>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  $\lambda$ , which implies  $u$  is an eigenvector of  $\Sigma$ , with eigenvalue  $\lambda$ .

<sup>2</sup>Because  $\Sigma$  is symmetric, the  $u_i$ 's will (or always can be chosen to be) orthogonal to each other.

of all possible orthogonal bases  $u_1, \dots, u_k$ , 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  $x^{(i)} \in \mathbb{R}^{100 \times 100}$  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,  $d$  speakers are speaking simultaneously at a party, and any microphone placed in the room records only an overlapping combination of the  $d$  speakers’ voices. But let’s say we have  $d$  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  $d$  speakers’ speech signals?

To formalize this problem, we imagine that there is some data  $s \in \mathbb{R}^d$  that is generated via  $d$  independent sources. What we observe is

$$x = As,$$

where  $A$  is an unknown square matrix called the **mixing matrix**. Repeated observations give us a dataset  $\{x^{(i)}; i = 1, \dots, n\}$ , 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  $d$ -dimensional vector, and  $s_j^{(i)}$  is the sound that speaker  $j$  was uttering at time  $i$ . Also,  $x^{(i)}$  is an  $d$ -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

$$W = \begin{bmatrix} -w_1^T - \\ \vdots \\ -w_d^T - \end{bmatrix}.$$

Thus,  $w_i \in \mathbb{R}^d$ , and the  $j$ -th source can be recovered as  $s_j^{(i)} = w_j^T x^{(i)}$ .

## 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 easy 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  $d$ -by- $d$  permutation matrix. This means that each row and each column of  $P$  has exactly one “1.” Here are 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 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  $\alpha$ , and the corresponding source were scaled by a factor of  $1/\alpha$ , then there is again no way to determine that this had happened given only the  $x^{(i)}$ 's. 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  $\alpha$  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  $2 \times 2$  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. Then, the distribution of  $x$  will be Gaussian,  $x \sim \mathcal{N}(0, AA^T)$ , since

$$\begin{aligned}\mathbb{E}_{s \sim \mathcal{N}(0, I)}[x] &= \mathbb{E}[As] = A\mathbb{E}[s] = 0 \\ \text{Cov}[x] &= \mathbb{E}_{s \sim \mathcal{N}(0, I)}[xx^T] = \mathbb{E}[Ass^TA^T] = A\mathbb{E}[ss^T]A^T = A \cdot \text{Cov}[s] \cdot A^T = AA^T\end{aligned}$$

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,  $x' \sim \mathcal{N}(0, AA^T)$ , since  $\mathbb{E}_{s \sim \mathcal{N}(0, I)}[x'(x')^T] = \mathbb{E}[A'ss^T(A')^T] = \mathbb{E}[ARss^T(AR)^T] = ARR^TA^T = AA^T$ . 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  $A'$ . 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  $d$  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 a random variable  $s$  is drawn according to some density  $p_s(s)$ . For simplicity, assume 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  $s \sim \text{Uniform}[0, 1]$ , so  $p_s(s) = 1\{0 \leq s \leq 1\}$ . Now, let  $A = 2$ , so  $x = 2s$ . Clearly,  $x$  is distributed uniformly in the interval  $[0, 2]$ . Thus, its density is given by  $p_x(x) = (0.5)1\{0 \leq x \leq 2\}$ . This does not equal  $p_s(Wx)$ , where  $W = 0.5 = A^{-1}$ . Instead, the correct formula is  $p_x(x) = p_s(Wx)|W|$ .

More generally, if  $s$  is a vector-valued distribution with density  $p_s$ , and  $x = As$  for a square, invertible matrix  $A$ , then the density of  $x$  is given by

$$p_x(x) = p_s(Wx) \cdot |W|,$$

where  $W = A^{-1}$ .

**Remark.** If you’re seen the result that  $A$  maps  $[0, 1]^d$  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  $A \in \mathbb{R}^{d \times d}$  be given, and let  $W = A^{-1}$  as usual. Also let  $C_1 = [0, 1]^d$  be the  $d$ -dimensional hypercube, and define  $C_2 = \{As : s \in C_1\} \subseteq \mathbb{R}^d$  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  $[0, 1]^d$ , 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. We describe an algorithm by Bell and Sejnowski, and we give an interpretation of their algorithm as a method for maximum likelihood estimation. (This is different from their original interpretation involving a complicated idea called the infomax principal which is no longer necessary given the modern understanding of ICA.)

We suppose that the distribution of each source  $s_j$  is given by a density  $p_s$ , and that the joint distribution of the sources  $s$  is given by

$$p(s) = \prod_{j=1}^d p_s(s_j).$$

Note that by modeling the joint distribution as a product of marginals, we capture the assumption that the sources are independent. Using our formulas from the previous section, this implies the following density on  $x = As = W^{-1}s$ :

$$p(x) = \prod_{j=1}^d p_s(w_j^T x) \cdot |W|.$$

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  $F(z_0) = P(z \leq z_0) = \int_{-\infty}^{z_0} p_z(z) dz$  and the density is the derivative of the cdf:  $p_z(z) = F'(z)$ .

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 Gaussian cdf, as ICA doesn't work on Gaussian data. What we'll choose instead as a reasonable "default" cdf that slowly increases from 0 to 1, is the sigmoid function  $g(s) = 1/(1 + e^{-s})$ . Hence,  $p_s(s) = g'(s)$ .<sup>1</sup>

The square matrix  $W$  is the parameter in our model. Given a training set  $\{x^{(i)}; i = 1, \dots, n\}$ , the log likelihood is given by

$$\ell(W) = \sum_{i=1}^n \left( \sum_{j=1}^d \log g'(w_j^T x^{(i)}) + \log |W| \right).$$

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:

$$W := W + \alpha \left( \begin{bmatrix} 1 - 2g(w_1^T x^{(i)}) \\ 1 - 2g(w_2^T x^{(i)}) \\ \vdots \\ 1 - 2g(w_d^T x^{(i)}) \end{bmatrix} x^{(i)T} + (W^T)^{-1} \right),$$

---

<sup>1</sup>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  $\mathbb{E}[s] = 0$  (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  $\mathbb{E}[x] = \mathbb{E}[As] = 0$ .

where  $\alpha$  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 implicitly 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. However, for problems where successive training examples are correlated, when implementing stochastic gradient ascent, it 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:

$$s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} s_3 \xrightarrow{a_3} \dots$$

Upon visiting the sequence of states  $s_0, s_1, \dots$  with actions  $a_0, a_1, \dots$ , our total payoff is given by

$$R(s_0, a_0) + \gamma R(s_1, a_1) + \gamma^2 R(s_2, a_2) + \dots$$

Or, when we are writing rewards as a function of the states only, this becomes

$$R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \dots$$

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:

$$\mathbb{E} [R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \dots]$$

Note that the reward at timestep  $t$  is **discounted** by a factor of  $\gamma^t$ . 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,  $\gamma$  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  $\pi$  if, whenever we are in state  $s$ , we take action  $a = \pi(s)$ . We also define the **value function** for a policy  $\pi$  according to

$$V^\pi(s) = \mathbb{E} [R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \dots \mid s_0 = s, \pi].$$

$V^\pi(s)$  is simply the expected sum of discounted rewards upon starting in state  $s$ , and taking actions according to  $\pi$ .<sup>1</sup>

Given a fixed policy  $\pi$ , its value function  $V^\pi$  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 right away 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  $\mathbb{E}_{s' \sim P_{s\pi(s)}}[V^\pi(s')]$ . 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  $V^\pi$ . 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.

---

<sup>1</sup>This notation in which we condition on  $\pi$  isn't technically correct because  $\pi$  isn't a random variable, but this is quite standard in the literature.

We also define the **optimal value function** according to

$$V^*(s) = \max_{\pi} V^\pi(s). \quad (1)$$

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:

$$V^*(s) = R(s) + \max_{a \in A} \gamma \sum_{s' \in S} P_{sa}(s') V^*(s'). \quad (2)$$

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:

$$\pi^*(s) = \arg \max_{a \in A} \sum_{s' \in S} P_{sa}(s') V^*(s'). \quad (3)$$

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  $\pi$ , we have

$$V^*(s) = V^{\pi^*}(s) \geq V^\pi(s).$$

The first equality says that the  $V^{\pi^*}$ , the value function for  $\pi^*$ , is equal to the optimal value function  $V^*$  for every state  $s$ . Further, the inequality above says that  $\pi^*$ 's value is at least as large as the value of any other other policy. In other words,  $\pi^*$  as defined in Equation (3) is the optimal policy.

Note that  $\pi^*$  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  $s'$ . The same policy  $\pi^*$  attains the maximum in Equation (1) for *all* states  $s$ . This means that we can use the same policy  $\pi^*$  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 ( $|S| <$

$\infty$ ,  $|A| < \infty$ ). In this section, we will also assume that we know the state transition probabilities  $\{P_{sa}\}$  and the reward function  $R$ .

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  $\pi$  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  $\pi$  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  $\pi$  will converge to  $\pi^*$ .

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  $V^\pi$  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  $\gamma$  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:

$$\begin{aligned} s_0^{(1)} &\xrightarrow{a_0^{(1)}} s_1^{(1)} \xrightarrow{a_1^{(1)}} s_2^{(1)} \xrightarrow{a_2^{(1)}} s_3^{(1)} \xrightarrow{a_3^{(1)}} \dots \\ s_0^{(2)} &\xrightarrow{a_0^{(2)}} s_1^{(2)} \xrightarrow{a_1^{(2)}} s_2^{(2)} \xrightarrow{a_2^{(2)}} s_3^{(2)} \xrightarrow{a_3^{(2)}} \dots \\ &\dots \end{aligned}$$

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:

$$P_{sa}(s') = \frac{\#\text{times took we action } a \text{ in state } s \text{ and got to } s'}{\#\text{times we took action } a \text{ in state } s} \quad (4)$$

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  $\pi$  randomly.
2. Repeat {
  - (a) Execute  $\pi$  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  $\pi$  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

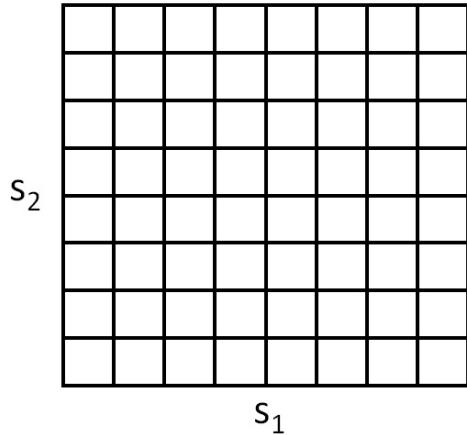
states. For example, for a car, we might represent the state as  $(x, y, \theta, \dot{x}, \dot{y}, \dot{\theta})$ , comprising its position  $(x, y)$ ; orientation  $\theta$ ; velocity in the  $x$  and  $y$  directions  $\dot{x}$  and  $\dot{y}$ ; and angular velocity  $\dot{\theta}$ . Hence,  $S = \mathbb{R}^6$  is an infinite set of states, because there is an infinite number of possible positions and orientations for the car.<sup>2</sup> Similarly, the inverted pendulum you saw in PS4 has states  $(x, \theta, \dot{x}, \dot{\theta})$ , where  $\theta$  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  $\phi$ , pitch  $\theta$ , and yaw  $\psi$  angles specify the 3d orientation of the helicopter.

In this section, we will consider settings where the state space is  $S = \mathbb{R}^d$ , 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

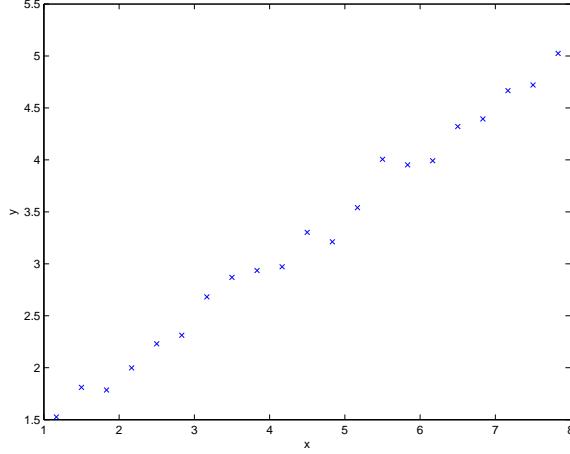
---

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

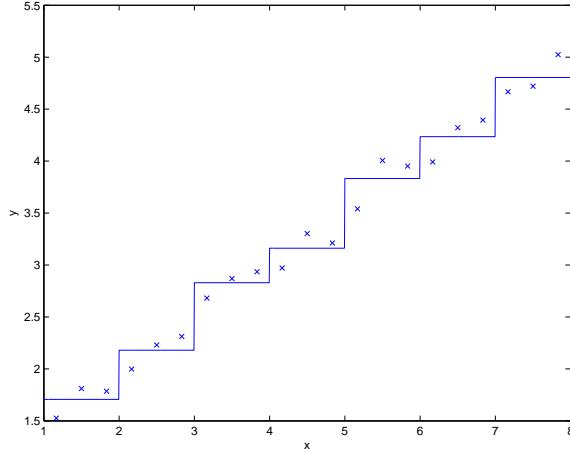
state  $s \in S$  and we need to pick an action to execute, we compute the corresponding discretized state  $\bar{s}$ , and execute action  $\pi^*(\bar{s})$ .

This discretization approach can work well for many problems. However, there are two downsides. First, it uses a fairly naive representation for  $V^*$  (and  $\pi^*$ ). 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  $S = \mathbb{R}^d$ , and we discretize each of the  $d$  dimensions of the state into  $k$  values. Then the total number of discrete states we have is  $k^d$ . This grows exponentially quickly in the dimension of the state space  $d$ , 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.

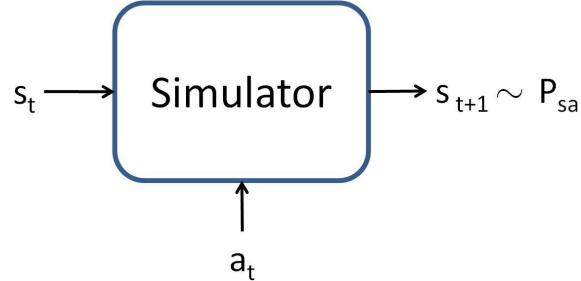
As a rule of thumb, discretization usually works extremely well for 1d and 2d problems (and has the advantage of being simple and quick to implement). Perhaps with a little bit of cleverness and some care in choosing the discretization method, it often works well for problems with up to 4d states. If 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, called 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 are 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.<sup>3</sup>

An alternative way to get a model is to learn one from data collected in the MDP. For example, suppose we execute  $n$  **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  $n$  state sequences like the following:

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

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

$$s_{t+1} = As_t + Ba_t, \quad (5)$$

---

<sup>3</sup>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.

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  $n$  trials, by picking

$$\arg \min_{A,B} \sum_{i=1}^n \sum_{t=0}^{T-1} \left\| s_{t+1}^{(i)} - \left( As_t^{(i)} + Ba_t^{(i)} \right) \right\|_2^2.$$

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

We could also potentially use other loss functions for learning the model. For example, it has been found in recent work [?] that using  $\|\cdot\|_2$  norm (without the square) may be helpful in certain cases.

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. 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 modeling it as

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

where here  $\epsilon_t$  is a noise term, usually modeled as  $\epsilon_t \sim \mathcal{N}(0, \Sigma)$ . (The covariance matrix  $\Sigma$  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  $\phi_s$  and  $\phi_a$  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  $S = \mathbb{R}^d$ , but that the action space  $A$  is small and discrete.<sup>4</sup>

---

<sup>4</sup>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 this set of actions is usually less of a problem than discretizing the state space would have been.

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

$$V(s) := R(s) + \gamma \max_a \int_{s'} P_{sa}(s') V(s') ds' \quad (6)$$

$$= R(s) + \gamma \max_a \mathbb{E}_{s' \sim P_{sa}} [V(s')] \quad (7)$$

(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  $s^{(1)}, \dots, s^{(n)}$ . Specifically, we will use a supervised learning algorithm—linear regression in our description below—to approximate the value function as a linear or non-linear function of the states:

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

Here,  $\phi$  is some appropriate feature mapping of the states.

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

In detail, the algorithm is as follows:

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

2. Initialize  $\theta := 0$ .

3. Repeat {

For  $i = 1, \dots, n$  {

For each action  $a \in A$  {

Sample  $s'_1, \dots, s'_k \sim P_{s^{(i)}a}$  (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  $R(s^{(i)}) + \gamma \mathbb{E}_{s' \sim P_{s^{(i)}a}} [V(s')]$ .

}

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

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

```

    }
    // 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  $\theta := \arg \min_{\theta} \frac{1}{2} \sum_{i=1}^n (\theta^T \phi(s^{(i)}) - y^{(i)})^2$ 
}

```

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  $(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \dots, (x^{(n)}, y^{(n)})$ , 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 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

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

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

In practice, there are 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 deterministic function of the states (such as  $f(s_t, a_t) = As_t + Ba_t$ ), and  $\epsilon$  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  $k = 1$ . Equivalent, this can be derived from Equation (8) using the approximation

$$E_{s'}[V(s')] \approx V(E_{s'}[s']) \quad (9)$$

$$= V(f(s, a)), \quad (10)$$

where here the expectation is over the random  $s' \sim P_{sa}$ . So long as the noise terms  $\epsilon_t$  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

Dan Boneh & Andrew Ng

## Part XIV LQR, DDP and LQG

Linear Quadratic Regulation, Differential Dynamic Programming and Linear Quadratic Gaussian

### 1 Finite-horizon MDPs

In the previous set of notes about Reinforcement Learning, we defined Markov Decision Processes (MDPs) and covered Value Iteration / Policy Iteration in a simplified setting. More specifically we introduced the **optimal Bellman equation** that defines the optimal value function  $V^{\pi^*}$  of the optimal policy  $\pi^*$ .

$$V^{\pi^*}(s) = R(s) + \max_{a \in \mathcal{A}} \gamma \sum_{s' \in S} P_{sa}(s') V^{\pi^*}(s')$$

Recall that from the optimal value function, we were able to recover the optimal policy  $\pi^*$  with

$$\pi^*(s) = \operatorname{argmax}_{a \in \mathcal{A}} \sum_{s' \in S} P_{sa}(s') V^*(s')$$

In this set of lecture notes we'll place ourselves in a more general setting:

1. We want to write equations that make sense for both the discrete and the continuous case. We'll therefore write

---

scribe: Guillaume Genthal

$$\begin{aligned} \mathbb{E}_{s' \sim P_{sa}} [V^{\pi^*}(s')] &\quad \text{instead of} \\ \sum_{s' \in S} P_{sa}(s') V^{\pi^*}(s') \end{aligned}$$

meaning that we take the expectation of the value function at the next state. In the finite case, we can rewrite the expectation as a sum over states. In the continuous case, we can rewrite the expectation as an integral. The notation  $s' \sim P_{sa}$  means that the state  $s'$  is sampled from the distribution  $P_{sa}$ .

2. We'll assume that the rewards depend on **both states and actions**. In other words,  $R : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$ . This implies that the previous mechanism for computing the optimal action is changed into

$$\pi^*(s) = \operatorname{argmax}_{a \in \mathcal{A}} R(s, a) + \gamma \mathbb{E}_{s' \sim P_{sa}} [V^{\pi^*}(s')]$$

3. Instead of considering an infinite horizon MDP, we'll assume that we have a **finite horizon MDP** that will be defined as a tuple

$$(\mathcal{S}, \mathcal{A}, P_{sa}, T, R)$$

with  $T > 0$  the **time horizon** (for instance  $T = 100$ ). In this setting, our definition of payoff is going to be (slightly) different:

$$R(s_0, a_0) + R(s_1, a_1) + \dots + R(s_T, a_T)$$

instead of (infinite horizon case)

$$\begin{aligned} R(s_0, a_0) + \gamma R(s_1, a_1) + \gamma^2 R(s_2, a_2) + \dots \\ \sum_{t=0}^{\infty} R(s_t, a_t) \gamma^t \end{aligned}$$

*What happened to the discount factor  $\gamma$ ?* Remember that the introduction of  $\gamma$  was (partly) justified by the necessity of making sure that

the infinite sum would be finite and well-defined. If the rewards are bounded by a constant  $\bar{R}$ , the payoff is indeed bounded by

$$\left| \sum_{t=0}^{\infty} R(s_t) \gamma^t \right| \leq \bar{R} \sum_{t=0}^{\infty} \gamma^t$$

and we recognize a geometric sum! Here, as the payoff is a finite sum, the discount factor  $\gamma$  is not necessary anymore.

In this new setting, things behave quite differently. First, the optimal policy  $\pi^*$  might be non-stationary, meaning that **it changes over time**. In other words, now we have

$$\pi^{(t)} : \mathcal{S} \rightarrow \mathcal{A}$$

where the superscript  $(t)$  denotes the policy at time step  $t$ . The dynamics of the finite horizon MDP following policy  $\pi^{(t)}$  proceeds as follows: we start in some state  $s_0$ , take some action  $a_0 := \pi^{(0)}(s_0)$  according to our policy at time step 0. The MDP transitions to a successor  $s_1$ , drawn according to  $P_{s_0 a_0}$ . Then, we get to pick another action  $a_1 := \pi^{(1)}(s_1)$  following our new policy at time step 1 and so on...

*Why does the optimal policy happen to be non-stationary in the finite-horizon setting?* Intuitively, as we have a finite numbers of actions to take, we might want to adopt different strategies depending on where we are in the environment and how much time we have left. Imagine a grid with 2 goals with rewards +1 and +10. At the beginning, we might want to take actions to aim for the +10 goal. But if after some steps, dynamics somehow pushed us closer to the +1 goal and we don't have enough steps left to be able to reach the +10 goal, then a better strategy would be to aim for the +1 goal...

4. This observation allows us to use **time dependent dynamics**

$$s_{t+1} \sim P_{s_t, a_t}^{(t)}$$

meaning that the transition's distribution  $P_{s_t, a_t}^{(t)}$  changes over time. The same thing can be said about  $R^{(t)}$ . Note that this setting is a better

model for real life. In a car, the gas tank empties, traffic changes, etc. Combining the previous remarks, we'll use the following general formulation for our finite horizon MDP

$$(\mathcal{S}, \mathcal{A}, P_{sa}^{(t)}, T, R^{(t)})$$

**Remark:** notice that the above formulation would be equivalent to adding the time into the state.

The value function at time  $t$  for a policy  $\pi$  is then defined in the same way as before, as an expectation over trajectories generated following policy  $\pi$  starting in state  $s$ .

$$V_t(s) = \mathbb{E} [R^{(t)}(s_t, a_t) + \dots + R^{(T)}(s_T, a_T) | s_t = s, \pi]$$

Now, the question is

*In this finite-horizon setting, how do we find the optimal value function*

$$V_t^*(s) = \max_{\pi} V_t^{\pi}(s)$$

It turns out that Bellman's equation for Value Iteration is made for **Dynamic Programming**. This may come as no surprise as Bellman is one of the fathers of dynamic programming and the Bellman equation is strongly related to the field. To understand how we can simplify the problem by adopting an iteration-based approach, we make the following observations:

1. Notice that at the end of the game (for time step  $T$ ), the optimal value is obvious

$$\forall s \in \mathcal{S} : V_T^*(s) := \max_{a \in \mathcal{A}} R^{(T)}(s, a) \quad (1)$$

2. For another time step  $0 \leq t < T$ , if we suppose that we know the optimal value function for the next time step  $V_{t+1}^*$ , then we have

$$\forall t < T, s \in \mathcal{S} : V_t^*(s) := \max_{a \in \mathcal{A}} \left[ R^{(t)}(s, a) + \mathbb{E}_{s' \sim P_{sa}^{(t)}} [V_{t+1}^*(s')] \right] \quad (2)$$

With these observations in mind, we can come up with a clever algorithm to solve for the optimal value function:

1. compute  $V_T^*$  using equation (1).

2. for  $t = T - 1, \dots, 0$ :

compute  $V_t^*$  using  $V_{t+1}^*$  using equation (2)

**Side note** We can interpret standard value iteration as a special case of this general case, but without keeping track of time. It turns out that in the standard setting, if we run value iteration for  $T$  steps, we get a  $\gamma^T$  approximation of the optimal value iteration (geometric convergence). See problem set 4 for a proof of the following result:

Theorem Let  $B$  denote the Bellman update and  $\|f(x)\|_\infty := \sup_x |f(x)|$ . If  $V_t$  denotes the value function at the  $t$ -th step, then

$$\begin{aligned} \|V_{t+1} - V^*\|_\infty &= \|B(V_t) - V^*\|_\infty \\ &\leq \gamma \|V_t - V^*\|_\infty \\ &\leq \gamma^t \|V_1 - V^*\|_\infty \end{aligned}$$

In other words, the Bellman operator  $B$  is a  $\gamma$ -contracting operator.

## 2 Linear Quadratic Regulation (LQR)

In this section, we'll cover a special case of the finite-horizon setting described in Section 1, for which the **exact solution** is (easily) tractable. This model is widely used in robotics, and a common technique in many problems is to reduce the formulation to this framework.

First, let's describe the model's assumptions. We place ourselves in the continuous setting, with

$$\mathcal{S} = \mathbb{R}^n, \quad \mathcal{A} = \mathbb{R}^d$$

and we'll assume **linear transitions** (with noise)

$$s_{t+1} = A_t s_t + B_t a_t + w_t$$

where  $A_t \in \mathbb{R}^{n \times n}, B_t \in \mathbb{R}^{n \times d}$  are matrices and  $w_t \sim \mathcal{N}(0, \Sigma_t)$  is some gaussian noise (with **zero** mean). As we'll show in the following paragraphs,

it turns out that the noise, as long as it has zero mean, does not impact the optimal policy!

We'll also assume **quadratic rewards**

$$R^{(t)}(s_t, a_t) = -s_t^\top U_t s_t - a_t^\top W_t a_t$$

where  $U_t \in R^{n \times n}$ ,  $W_t \in R^{d \times d}$  are positive definite matrices (meaning that the reward is always **negative**).

**Remark** Note that the quadratic formulation of the reward is equivalent to saying that we want our state to be close to the origin (where the reward is higher). For example, if  $U_t = I_n$  (the identity matrix) and  $W_t = I_d$ , then  $R_t = -\|s_t\|^2 - \|a_t\|^2$ , meaning that we want to take smooth actions (small norm of  $a_t$ ) to go back to the origin (small norm of  $s_t$ ). This could model a car trying to stay in the middle of lane without making impulsive moves...

Now that we have defined the assumptions of our LQR model, let's cover the 2 steps of the LQR algorithm

**step 1** suppose that we don't know the matrices  $A, B, \Sigma$ . To estimate them, we can follow the ideas outlined in the Value Approximation section of the RL notes. First, collect transitions from an arbitrary policy. Then, use linear regression to find  $\text{argmin}_{A,B} \sum_{i=1}^m \sum_{t=0}^{T-1} \left\| s_{t+1}^{(i)} - \left( As_t^{(i)} + Ba_t^{(i)} \right) \right\|^2$ . Finally, use a technique seen in Gaussian Discriminant Analysis to learn  $\Sigma$ .

**step 2** assuming that the parameters of our model are known (given or estimated with step 1), we can derive the optimal policy using dynamic programming.

In other words, given

$$\begin{cases} s_{t+1} &= A_t s_t + B_t a_t + w_t & A_t, B_t, U_t, W_t, \Sigma_t \text{ known} \\ R^{(t)}(s_t, a_t) &= -s_t^\top U_t s_t - a_t^\top W_t a_t \end{cases}$$

we want to compute  $V_t^*$ . If we go back to section 1, we can apply dynamic programming, which yields

## 1. Initialization step

For the last time step  $T$ ,

$$\begin{aligned} V_T^*(s_T) &= \max_{a_T \in \mathcal{A}} R_T(s_T, a_T) \\ &= \max_{a_T \in \mathcal{A}} -s_T^\top U_T s_T - a_T^\top W_t a_T \\ &= -s_T^\top U_t s_T \quad (\text{maximized for } a_T = 0) \end{aligned}$$

## 2. Recurrence step

Let  $t < T$ . Suppose we know  $V_{t+1}^*$ .

Fact 1: It can be shown that if  $V_{t+1}^*$  is a quadratic function in  $s_t$ , then  $V_t^*$  is also a quadratic function. In other words, there exists some matrix  $\Phi$  and some scalar  $\Psi$  such that

$$\begin{aligned} \text{if } V_{t+1}^*(s_{t+1}) &= s_{t+1}^\top \Phi_{t+1} s_{t+1} + \Psi_{t+1} \\ \text{then } V_t^*(s_t) &= s_t^\top \Phi_t s_t + \Psi_t \end{aligned}$$

For time step  $t = T$ , we had  $\Phi_t = -U_T$  and  $\Psi_T = 0$ .

Fact 2: We can show that the optimal policy is just a linear function of the state.

Knowing  $V_{t+1}^*$  is equivalent to knowing  $\Phi_{t+1}$  and  $\Psi_{t+1}$ , so we just need to explain how we compute  $\Phi_t$  and  $\Psi_t$  from  $\Phi_{t+1}$  and  $\Psi_{t+1}$  and the other parameters of the problem.

$$\begin{aligned} V_t^*(s_t) &= s_t^\top \Phi_t s_t + \Psi_t \\ &= \max_{a_t} \left[ R^{(t)}(s_t, a_t) + \mathbb{E}_{s_{t+1} \sim P_{s_t, a_t}^{(t)}} [V_{t+1}^*(s_{t+1})] \right] \\ &= \max_{a_t} \left[ -s_t^\top U_t s_t - a_t^\top V_t a_t + \mathbb{E}_{s_{t+1} \sim \mathcal{N}(A_t s_t + B_t a_t, \Sigma_t)} [s_{t+1}^\top \Phi_{t+1} s_{t+1} + \Psi_{t+1}] \right] \end{aligned}$$

where the second line is just the definition of the optimal value function and the third line is obtained by plugging in the dynamics of our model along with the quadratic assumption. Notice that the last expression is a quadratic function in  $a_t$  and can thus be (easily) optimized<sup>1</sup>. We get the optimal action  $a_t^*$

---

<sup>1</sup>Use the identity  $\mathbb{E} [w_t^\top \Phi_{t+1} w_t] = \text{Tr}(\Sigma_t \Phi_{t+1})$  with  $w_t \sim \mathcal{N}(0, \Sigma_t)$

$$\begin{aligned} a_t^* &= [(B_t^\top \Phi_{t+1} B_t - V_t)^{-1} B_t \Phi_{t+1} A_t] \cdot s_t \\ &= L_t \cdot s_t \end{aligned}$$

where

$$L_t := [(B_t^\top \Phi_{t+1} B_t - W_t)^{-1} B_t \Phi_{t+1} A_t]$$

which is an impressive result: our optimal policy is **linear in  $s_t$** . Given  $a_t^*$  we can solve for  $\Phi_t$  and  $\Psi_t$ . We finally get the **Discrete Riccati equations**

$$\begin{aligned} \Phi_t &= A_t^\top \left( \Phi_{t+1} - \Phi_{t+1} B_t \left( B_t^\top \Phi_{t+1} B_t - W_t \right)^{-1} B_t \Phi_{t+1} \right) A_t - U_t \\ \Psi_t &= -\text{tr}(\Sigma_t \Phi_{t+1}) + \Psi_{t+1} \end{aligned}$$

Fact 3: we notice that  $\Phi_t$  depends on neither  $\Psi$  nor the noise  $\Sigma_t$ ! As  $L_t$  is a function of  $A_t, B_t$  and  $\Phi_{t+1}$ , it implies that the optimal policy also **does not depend on the noise!** (But  $\Psi_t$  does depend on  $\Sigma_t$ , which implies that  $V_t^*$  depends on  $\Sigma_t$ .)

Then, to summarize, the LQR algorithm works as follows

1. (if necessary) estimate parameters  $A_t, B_t, \Sigma_t$
2. initialize  $\Phi_T := -U_T$  and  $\Psi_T := 0$ .
3. iterate from  $t = T - 1 \dots 0$  to update  $\Phi_t$  and  $\Psi_t$  using  $\Phi_{t+1}$  and  $\Psi_{t+1}$  using the discrete Riccati equations. If there exists a policy that drives the state towards zero, then convergence is guaranteed!

Using Fact 3, we can be even more clever and make our algorithm run (slightly) faster! As the optimal policy does not depend on  $\Psi_t$ , and the update of  $\Phi_t$  only depends on  $\Phi_t$ , it is sufficient to update **only  $\Phi_t$ !**

### 3 From non-linear dynamics to LQR

It turns out that a lot of problems can be reduced to LQR, even if dynamics are non-linear. While LQR is a nice formulation because we are able to come up with a nice exact solution, it is far from being general. Let's take for instance the case of the inverted pendulum. The transitions between states look like

$$\begin{pmatrix} x_{t+1} \\ \dot{x}_{t+1} \\ \theta_{t+1} \\ \dot{\theta}_{t+1} \end{pmatrix} = F \left( \begin{pmatrix} x_t \\ \dot{x}_t \\ \theta_t \\ \dot{\theta}_t \end{pmatrix}, a_t \right)$$

where the function  $F$  depends on the cos of the angle etc. Now, the question we may ask is

*Can we linearize this system?*

#### 3.1 Linearization of dynamics

Let's suppose that at time  $t$ , the system spends most of its time in some state  $\bar{s}_t$  and the actions we perform are around  $\bar{a}_t$ . For the inverted pendulum, if we reached some kind of optimal, this is true: our actions are small and we don't deviate much from the vertical.

We are going to use Taylor expansion to linearize the dynamics. In the simple case where the state is one-dimensional and the transition function  $F$  does not depend on the action, we would write something like

$$s_{t+1} = F(s_t) \approx F(\bar{s}_t) + F'(\bar{s}_t) \cdot (s_t - \bar{s}_t)$$

In the more general setting, the formula looks the same, with gradients instead of simple derivatives

$$s_{t+1} \approx F(\bar{s}_t, \bar{a}_t) + \nabla_s F(\bar{s}_t, \bar{a}_t) \cdot (s_t - \bar{s}_t) + \nabla_a F(\bar{s}_t, \bar{a}_t) \cdot (a_t - \bar{a}_t) \quad (3)$$

and now,  $s_{t+1}$  is linear in  $s_t$  and  $a_t$ , because we can rewrite equation (3) as

$$s_{t+1} \approx As_t + Bs_t + \kappa$$

where  $\kappa$  is some constant and  $A, B$  are matrices. Now, this writing looks awfully similar to the assumptions made for LQR. We just have to get rid

of the constant term  $\kappa$ ! It turns out that the constant term can be absorbed into  $s_t$  by artificially increasing the dimension by one. This is the same trick that we used at the beginning of the class for linear regression...

### 3.2 Differential Dynamic Programming (DDP)

The previous method works well for cases where the goal is to stay around some state  $s^*$  (think about the inverted pendulum, or a car having to stay in the middle of a lane). However, in some cases, the goal can be more complicated.

We'll cover a method that applies when our system has to follow some trajectory (think about a rocket). This method is going to discretize the trajectory into discrete time steps, and create intermediary goals around which we will be able to use the previous technique! This method is called **Differential Dynamic Programming**. The main steps are

**step 1** come up with a nominal trajectory using a naive controller, that approximate the trajectory we want to follow. In other words, our controller is able to approximate the gold trajectory with

$$s_0^*, a_0^* \rightarrow s_1^*, a_1^* \rightarrow \dots$$

**step 2** linearize the dynamics around each trajectory point  $s_t^*$ , in other words

$$s_{t+1} \approx F(s_t^*, a_t^*) + \nabla_s F(s_t^*, a_t^*)(s_t - s_t^*) + \nabla_a F(s_t^*, a_t^*)(a_t - a_t^*)$$

where  $s_t, a_t$  would be our current state and action. Now that we have a linear approximation around each of these points, we can use the previous section and rewrite

$$s_{t+1} = A_t \cdot s_t + B_t \cdot a_t$$

(notice that in that case, we use the non-stationary dynamics setting that we mentioned at the beginning of these lecture notes)

**Note** We can apply a similar derivation for the reward  $R^{(t)}$ , with a second-order Taylor expansion.

$$\begin{aligned}
R(s_t, a_t) &\approx R(s_t^*, a_t^*) + \nabla_s R(s_t^*, a_t^*)(s_t - s_t^*) + \nabla_a R(s_t^*, a_t^*)(a_t - a_t^*) \\
&+ \frac{1}{2}(s_t - s_t^*)^\top H_{ss}(s_t - s_t^*) + (s_t - s_t^*)^\top H_{sa}(a_t - a_t^*) \\
&+ \frac{1}{2}(a_t - a_t^*)^\top H_{aa}(a_t - a_t^*)
\end{aligned}$$

where  $H_{xy}$  refers to the entry of the Hessian of  $R$  with respect to  $x$  and  $y$  evaluated in  $(s_t^*, a_t^*)$  (omitted for readability). This expression can be re-written as

$$R_t(s_t, a_t) = -s_t^\top U_t s_t - a_t^\top W_t a_t$$

for some matrices  $U_t, W_t$ , with the same trick of adding an extra dimension of ones. To convince yourself, notice that

$$(1 \ x) \cdot \begin{pmatrix} a & b \\ b & c \end{pmatrix} \cdot \begin{pmatrix} 1 \\ x \end{pmatrix} = a + 2bx + cx^2$$

**step 3** Now, you can convince yourself that our problem is **strictly** re-written in the LQR framework. Let's just use LQR to find the optimal policy  $\pi_t$ . As a result, our new controller will (hopefully) be better!

**Note:** Some problems might arise if the LQR trajectory deviates too much from the linearized approximation of the trajectory, but that can be fixed with reward-shaping...

**step 4** Now that we get a new controller (our new policy  $\pi_t$ ), we use it to produce a new trajectory

$$s_0^*, \pi_0(s_0^*) \rightarrow s_1^*, \pi_1(s_1^*) \rightarrow \dots \rightarrow s_T^*$$

note that when we generate this new trajectory, we use the real  $F$  and not its linear approximation to compute transitions, meaning that

$$s_{t+1}^* = F(s_t^*, a_t^*)$$

then, go back to step 2 and repeat until some stopping criterion.

## 4 Linear Quadratic Gaussian (LQG)

Often, in the real word, we don't get to observe the full state  $s_t$ . For example, an autonomous car could receive an image from a camera, which is merely an **observation**, and not the full state of the world. So far, we assumed that the state was available. As this might not hold true for most of the real-world problems, we need a new tool to model this situation: **Partially Observable MDPs**.

A POMDP is an MDP with an extra observation layer. In other words, we introduce a new variable  $o_t$ , that follows some conditional distribution given the current state  $s_t$

$$o_t | s_t \sim O(o|s)$$

Formally, a finite-horizon POMDP is given by a tuple

$$(\mathcal{S}, \mathcal{O}, \mathcal{A}, P_{sa}, T, R)$$

Within this framework, the general strategy is to maintain a **belief state** (distribution over states) based on the observation  $o_1, \dots, o_t$ . Then, a policy in a POMDP maps belief states to actions.

In this section, we'll present a extension of LQR to this new setting. Assume that we observe  $y_t \in \mathbb{R}^m$  with  $m < n$  such that

$$\begin{cases} y_t &= C \cdot s_t + v_t \\ s_{t+1} &= A \cdot s_t + B \cdot a_t + w_t \end{cases}$$

where  $C \in \mathbb{R}^{m \times n}$  is a compression matrix and  $v_t$  is the sensor noise (also gaussian, like  $w_t$ ). Note that the reward function  $R^{(t)}$  is left unchanged, as a function of the state (not the observation) and action. Also, as distributions are gaussian, the belief state is also going to be gaussian. In this new framework, let's give an overview of the strategy we are going to adopt to find the optimal policy:

**step 1** first, compute the distribution on the possible states (the belief state), based on the observations we have. In other words, we want to compute the mean  $s_{t|t}$  and the covariance  $\Sigma_{t|t}$  of

$$s_t | y_1, \dots, y_t \sim \mathcal{N}(s_{t|t}, \Sigma_{t|t})$$

to perform the computation efficiently over time, we'll use the **Kalman Filter** algorithm (used on-board Apollo Lunar Module!).

**step 2** now that we have the distribution, we'll use the mean  $s_{t|t}$  as the best approximation for  $s_t$

**step 3** then set the action  $a_t := L_t s_{t|t}$  where  $L_t$  comes from the regular LQR algorithm.

Intuitively, to understand why this works, notice that  $s_{t|t}$  is a noisy approximation of  $s_t$  (equivalent to adding more noise to LQR) but we proved that LQR is independent of the noise!

Step 1 needs to be explicated. We'll cover a simple case where there is no action dependence in our dynamics (but the general case follows the same idea). Suppose that

$$\begin{cases} s_{t+1} &= A \cdot s_t + w_t, \quad w_t \sim N(0, \Sigma_s) \\ y_t &= C \cdot s_t + v_t, \quad v_t \sim N(0, \Sigma_y) \end{cases}$$

As noises are Gaussians, we can easily prove that the joint distribution is also Gaussian

$$\begin{pmatrix} s_1 \\ \vdots \\ s_t \\ y_1 \\ \vdots \\ y_t \end{pmatrix} \sim \mathcal{N}(\mu, \Sigma) \quad \text{for some } \mu, \Sigma$$

then, using the marginal formulas of gaussians (see Factor Analysis notes), we would get

$$s_t | y_1, \dots, y_t \sim \mathcal{N}(s_{t|t}, \Sigma_{t|t})$$

However, computing the marginal distribution parameters using these formulas would be computationally expensive! It would require manipulating matrices of shape  $t \times t$ . Recall that inverting a matrix can be done in  $O(t^3)$ , and it would then have to be repeated over the time steps, yielding a cost in  $O(t^4)$ !

The **Kalman filter** algorithm provides a much better way of computing the mean and variance, by updating them over time in **constant time in**

$t$ ! The kalman filter is based on two basics steps. Assume that we know the distribution of  $s_t|y_1, \dots, y_t$ :

**predict step** compute  $s_{t+1}|y_1, \dots, y_t$

**update step** compute  $s_{t+1}|y_1, \dots, y_{t+1}$

and iterate over time steps! The combination of the predict and update steps updates our belief states. In other words, the process looks like

$$(s_t|y_1, \dots, y_t) \xrightarrow{\text{predict}} (s_{t+1}|y_1, \dots, y_t) \xrightarrow{\text{update}} (s_{t+1}|y_1, \dots, y_{t+1}) \xrightarrow{\text{predict}} \dots$$

**predict step** Suppose that we know the distribution of

$$s_t|y_1, \dots, y_t \sim \mathcal{N}(s_{t|t}, \Sigma_{t|t})$$

then, the distribution over the next state is also a gaussian distribution

$$s_{t+1}|y_1, \dots, y_t \sim \mathcal{N}(s_{t+1|t}, \Sigma_{t+1|t})$$

where

$$\begin{cases} s_{t+1|t} = A \cdot s_{t|t} \\ \Sigma_{t+1|t} = A \cdot \Sigma_{t|t} \cdot A^\top + \Sigma_s \end{cases}$$

**update step** given  $s_{t+1|t}$  and  $\Sigma_{t+1|t}$  such that

$$s_{t+1}|y_1, \dots, y_t \sim \mathcal{N}(s_{t+1|t}, \Sigma_{t+1|t})$$

we can prove that

$$s_{t+1}|y_1, \dots, y_{t+1} \sim \mathcal{N}(s_{t+1|t+1}, \Sigma_{t+1|t+1})$$

where

$$\begin{cases} s_{t+1|t+1} = s_{t+1|t} + K_t(y_{t+1} - C s_{t+1|t}) \\ \Sigma_{t+1|t+1} = \Sigma_{t+1|t} - K_t \cdot C \cdot \Sigma_{t+1|t} \end{cases}$$

with

$$K_t := \Sigma_{t+1|t} C^\top (C \Sigma_{t+1|t} C^\top + \Sigma_y)^{-1}$$

The matrix  $K_t$  is called the **Kalman gain**.

Now, if we have a closer look at the formulas, we notice that we don't need the observations prior to time step  $t$ ! The update steps only depends on the previous distribution. Putting it all together, the algorithm first runs a forward pass to compute the  $K_t$ ,  $\Sigma_{t|t}$  and  $s_{t|t}$  (sometimes referred to as  $\hat{s}$  in the literature). Then, it runs a backward pass (the LQR updates) to compute the quantities  $\Psi_t$ ,  $\Psi_t$  and  $L_t$ . Finally, we recover the optimal policy with  $a_t^* = L_t s_{t|t}$ .

# Linear Algebra Review and Reference

Zico Kolter (updated by Chuong Do and Tengyu Ma)

June 20, 2020

## Contents

<b>1 Basic Concepts and Notation</b>	<b>2</b>
1.1 Basic Notation . . . . .	2
<b>2 Matrix Multiplication</b>	<b>3</b>
2.1 Vector-Vector Products . . . . .	3
2.2 Matrix-Vector Products . . . . .	4
2.3 Matrix-Matrix Products . . . . .	5
<b>3 Operations and Properties</b>	<b>7</b>
3.1 The Identity Matrix and Diagonal Matrices . . . . .	8
3.2 The Transpose . . . . .	8
3.3 Symmetric Matrices . . . . .	8
3.4 The Trace . . . . .	9
3.5 Norms . . . . .	10
3.6 Linear Independence and Rank . . . . .	11
3.7 The Inverse of a Square Matrix . . . . .	11
3.8 Orthogonal Matrices . . . . .	12
3.9 Range and Nullspace of a Matrix . . . . .	13
3.10 The Determinant . . . . .	14
3.11 Quadratic Forms and Positive Semidefinite Matrices . . . . .	17
3.12 Eigenvalues and Eigenvectors . . . . .	18
3.13 Eigenvalues and Eigenvectors of Symmetric Matrices . . . . .	19
<b>4 Matrix Calculus</b>	<b>23</b>
4.1 The Gradient . . . . .	23
4.2 The Hessian . . . . .	24
4.3 Gradients and Hessians of Quadratic and Linear Functions . . . . .	26
4.4 Least Squares . . . . .	27
4.5 Gradients of the Determinant . . . . .	28
4.6 Eigenvalues as Optimization . . . . .	28

# 1 Basic Concepts and Notation

Linear algebra provides a way of compactly representing and operating on sets of linear equations. For example, consider the following system of equations:

$$\begin{aligned} 4x_1 - 5x_2 &= -13 \\ -2x_1 + 3x_2 &= 9. \end{aligned}$$

This is two equations and two variables, so as you know from high school algebra, you can find a unique solution for  $x_1$  and  $x_2$  (unless the equations are somehow degenerate, for example if the second equation is simply a multiple of the first, but in the case above there is in fact a unique solution). In matrix notation, we can write the system more compactly as

$$Ax = b$$

with

$$A = \begin{bmatrix} 4 & -5 \\ -2 & 3 \end{bmatrix}, \quad b = \begin{bmatrix} -13 \\ 9 \end{bmatrix}.$$

As we will see shortly, there are many advantages (including the obvious space savings) to analyzing linear equations in this form.

## 1.1 Basic Notation

We use the following notation:

- By  $A \in \mathbb{R}^{m \times n}$  we denote a matrix with  $m$  rows and  $n$  columns, where the entries of  $A$  are real numbers.
- By  $x \in \mathbb{R}^n$ , we denote a vector with  $n$  entries. By convention, an  $n$ -dimensional vector is often thought of as a matrix with  $n$  rows and 1 column, known as a **column vector**. If we want to explicitly represent a **row vector** — a matrix with 1 row and  $n$  columns — we typically write  $x^T$  (here  $x^T$  denotes the transpose of  $x$ , which we will define shortly).
- The  $i$ th element of a vector  $x$  is denoted  $x_i$ :

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}.$$

- We use the notation  $a_{ij}$  (or  $A_{ij}$ ,  $A_{i,j}$ , etc) to denote the entry of  $A$  in the  $i$ th row and  $j$ th column:

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}.$$

- We denote the  $j$ th column of  $A$  by  $a^j$  or  $A_{:,j}$ :

$$A = \begin{bmatrix} | & | & & | \\ a^1 & a^2 & \cdots & a^n \\ | & | & & | \end{bmatrix}.$$

- We denote the  $i$ th row of  $A$  by  $a^T$  or  $A_{i,:}$ :

$$A = \begin{bmatrix} — & a_1^T & — \\ — & a_2^T & — \\ \vdots & & \\ — & a_m^T & — \end{bmatrix}.$$

- Viewing a matrix as a collection of column or row vectors is very important and convenient in many cases. In general, it would be mathematically (and conceptually) cleaner to operate on the level of vectors instead of scalars. There is no universal convention for denoting the columns or rows of a matrix, and thus you can feel free to change the notations as long as it's explicitly defined.

## 2 Matrix Multiplication

The product of two matrices  $A \in \mathbb{R}^{m \times n}$  and  $B \in \mathbb{R}^{n \times p}$  is the matrix

$$C = AB \in \mathbb{R}^{m \times p},$$

where

$$C_{ij} = \sum_{k=1}^n A_{ik}B_{kj}.$$

Note that in order for the matrix product to exist, the number of columns in  $A$  must equal the number of rows in  $B$ . There are many other ways of looking at matrix multiplication that may be more convenient and insightful than the standard definition above, and we'll start by examining a few special cases.

### 2.1 Vector-Vector Products

Given two vectors  $x, y \in \mathbb{R}^n$ , the quantity  $x^T y$ , sometimes called the *inner product* or *dot product* of the vectors, is a real number given by

$$x^T y \in \mathbb{R} = \begin{bmatrix} x_1 & x_2 & \cdots & x_n \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \sum_{i=1}^n x_i y_i.$$

Observe that inner products are really just special case of matrix multiplication. Note that it is always the case that  $x^T y = y^T x$ .

Given vectors  $x \in \mathbb{R}^m$ ,  $y \in \mathbb{R}^n$  (not necessarily of the same size),  $xy^T \in \mathbb{R}^{m \times n}$  is called the **outer product** of the vectors. It is a matrix whose entries are given by  $(xy^T)_{ij} = x_i y_j$ , i.e.,

$$xy^T \in \mathbb{R}^{m \times n} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix} \begin{bmatrix} y_1 & y_2 & \cdots & y_n \end{bmatrix} = \begin{bmatrix} x_1 y_1 & x_1 y_2 & \cdots & x_1 y_n \\ x_2 y_1 & x_2 y_2 & \cdots & x_2 y_n \\ \vdots & \vdots & \ddots & \vdots \\ x_m y_1 & x_m y_2 & \cdots & x_m y_n \end{bmatrix}.$$

As an example of how the outer product can be useful, let  $\mathbf{1} \in \mathbb{R}^n$  denote an  $n$ -dimensional vector whose entries are all equal to 1. Furthermore, consider the matrix  $A \in \mathbb{R}^{m \times n}$  whose columns are all equal to some vector  $x \in \mathbb{R}^m$ . Using outer products, we can represent  $A$  compactly as,

$$A = \begin{bmatrix} | & | & \cdots & | \\ x & x & \cdots & x \\ | & | & \cdots & | \end{bmatrix} = \begin{bmatrix} x_1 & x_1 & \cdots & x_1 \\ x_2 & x_2 & \cdots & x_2 \\ \vdots & \vdots & \ddots & \vdots \\ x_m & x_m & \cdots & x_m \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix} \begin{bmatrix} 1 & 1 & \cdots & 1 \end{bmatrix} = x\mathbf{1}^T.$$

## 2.2 Matrix-Vector Products

Given a matrix  $A \in \mathbb{R}^{m \times n}$  and a vector  $x \in \mathbb{R}^n$ , their product is a vector  $y = Ax \in \mathbb{R}^m$ . There are a couple ways of looking at matrix-vector multiplication, and we will look at each of them in turn.

If we write  $A$  by rows, then we can express  $Ax$  as,

$$y = Ax = \begin{bmatrix} \text{---} & a_1^T & \text{---} \\ \text{---} & a_2^T & \text{---} \\ \vdots & & \vdots \\ \text{---} & a_m^T & \text{---} \end{bmatrix} x = \begin{bmatrix} a_1^T x \\ a_2^T x \\ \vdots \\ a_m^T x \end{bmatrix}.$$

In other words, the  $i$ th entry of  $y$  is equal to the inner product of the  $i$ th row of  $A$  and  $x$ ,  $y_i = a_i^T x$ .

Alternatively, let's write  $A$  in column form. In this case we see that,

$$y = Ax = \begin{bmatrix} | & | & \cdots & | \\ a^1 & a^2 & \cdots & a^n \\ | & | & \cdots & | \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} a^1 \\ a^2 \\ \vdots \\ a^n \end{bmatrix} x_1 + \begin{bmatrix} a^2 \\ a^2 \\ \vdots \\ a^n \end{bmatrix} x_2 + \dots + \begin{bmatrix} a^n \\ a^n \\ \vdots \\ a^n \end{bmatrix} x_n . \quad (1)$$

In other words,  $y$  is a **linear combination** of the *columns* of  $A$ , where the coefficients of the linear combination are given by the entries of  $x$ .

So far we have been multiplying on the right by a column vector, but it is also possible to multiply on the left by a row vector. This is written,  $y^T = x^T A$  for  $A \in \mathbb{R}^{m \times n}$ ,  $x \in \mathbb{R}^m$ , and  $y \in \mathbb{R}^n$ . As before, we can express  $y^T$  in two obvious ways, depending on whether we express  $A$  in terms of its rows or columns. In the first case we express  $A$  in terms of its columns, which gives

$$y^T = x^T A = x^T \begin{bmatrix} | & | & & | \\ a^1 & a^2 & \cdots & a^n \\ | & | & & | \end{bmatrix} = [x^T a^1 \ x^T a^2 \ \cdots \ x^T a^n]$$

which demonstrates that the  $i$ th entry of  $y^T$  is equal to the inner product of  $x$  and the  $i$ th column of  $A$ .

Finally, expressing  $A$  in terms of rows we get the final representation of the vector-matrix product,

$$\begin{aligned} y^T &= x^T A \\ &= [x_1 \ x_2 \ \cdots \ x_n] \begin{bmatrix} \_ & a_1^T & \_ \\ \_ & a_2^T & \_ \\ \vdots & & \\ \_ & a_m^T & \_ \end{bmatrix} \\ &= x_1 [\_ \ a_1^T \ \_] + x_2 [\_ \ a_2^T \ \_] + \dots + x_n [\_ \ a_m^T \ \_] \end{aligned}$$

so we see that  $y^T$  is a linear combination of the *rows* of  $A$ , where the coefficients for the linear combination are given by the entries of  $x$ .

## 2.3 Matrix-Matrix Products

Armed with this knowledge, we can now look at four different (but, of course, equivalent) ways of viewing the matrix-matrix multiplication  $C = AB$  as defined at the beginning of this section.

First, we can view matrix-matrix multiplication as a set of vector-vector products. The most obvious viewpoint, which follows immediately from the definition, is that the  $(i, j)$ th entry of  $C$  is equal to the inner product of the  $i$ th row of  $A$  and the  $j$ th column of  $B$ . Symbolically, this looks like the following,

$$C = AB = \begin{bmatrix} \_ & a_1^T & \_ \\ \_ & a_2^T & \_ \\ \vdots & & \\ \_ & a_m^T & \_ \end{bmatrix} \begin{bmatrix} | & | & & | \\ b^1 & b^2 & \cdots & b^p \\ | & | & & | \end{bmatrix} = \begin{bmatrix} a_1^T b^1 & a_1^T b^2 & \cdots & a_1^T b^p \\ a_2^T b^1 & a_2^T b^2 & \cdots & a_2^T b^p \\ \vdots & \vdots & \ddots & \vdots \\ a_m^T b^1 & a_m^T b^2 & \cdots & a_m^T b^p \end{bmatrix}.$$

Remember that since  $A \in \mathbb{R}^{m \times n}$  and  $B \in \mathbb{R}^{n \times p}$ ,  $a_i \in \mathbb{R}^n$  and  $b^j \in \mathbb{R}^n$ , so these inner products all make sense. This is the most “natural” representation when we represent  $A$

by rows and  $B$  by columns. Alternatively, we can represent  $A$  by columns, and  $B$  by rows. This representation leads to a much trickier interpretation of  $AB$  as a sum of outer products. Symbolically,

$$C = AB = \left[ \begin{array}{cccc} | & | & & | \\ a^1 & a^2 & \cdots & a^n \\ | & | & & | \end{array} \right] \left[ \begin{array}{ccc} | & b_1^T & | \\ | & b_2^T & | \\ \vdots & & \\ | & b_n^T & | \end{array} \right] = \sum_{i=1}^n a^i b_i^T .$$

Put another way,  $AB$  is equal to the sum, over all  $i$ , of the outer product of the  $i$ th column of  $A$  and the  $i$ th row of  $B$ . Since, in this case,  $a^i \in \mathbb{R}^m$  and  $b_i \in \mathbb{R}^p$ , the dimension of the outer product  $a^i b_i^T$  is  $m \times p$ , which coincides with the dimension of  $C$ . Chances are, the last equality above may appear confusing to you. If so, take the time to check it for yourself!

Second, we can also view matrix-matrix multiplication as a set of matrix-vector products. Specifically, if we represent  $B$  by columns, we can view the columns of  $C$  as matrix-vector products between  $A$  and the columns of  $B$ . Symbolically,

$$C = AB = A \left[ \begin{array}{cccc} | & | & & | \\ b^1 & b^2 & \cdots & b^p \\ | & | & & | \end{array} \right] = \left[ \begin{array}{cccc} | & Ab^1 & | & | \\ | & Ab^2 & \cdots & | \\ | & & & | \end{array} \right]. \quad (2)$$

Here the  $i$ th column of  $C$  is given by the matrix-vector product with the vector on the right,  $c_i = Ab_i$ . These matrix-vector products can in turn be interpreted using both viewpoints given in the previous subsection. Finally, we have the analogous viewpoint, where we represent  $A$  by rows, and view the rows of  $C$  as the matrix-vector product between the rows of  $A$  and  $C$ . Symbolically,

$$C = AB = \left[ \begin{array}{ccc} | & a_1^T & | \\ | & a_2^T & | \\ \vdots & & \\ | & a_m^T & | \end{array} \right] B = \left[ \begin{array}{ccc} | & a_1^T B & | \\ | & a_2^T B & | \\ \vdots & & \\ | & a_m^T B & | \end{array} \right].$$

Here the  $i$ th row of  $C$  is given by the matrix-vector product with the vector on the left,  $c_i^T = a_i^T B$ .

It may seem like overkill to dissect matrix multiplication to such a large degree, especially when all these viewpoints follow immediately from the initial definition we gave (in about a line of math) at the beginning of this section. **The direct advantage of these various viewpoints is that they allow you to operate on the level/unit of vectors instead of scalars.** To fully understand linear algebra without getting lost in the complicated manipulation of indices, the key is to operate with as large concepts as possible.<sup>1</sup>

---

<sup>1</sup>E.g., if you could write all your math derivations with matrices or vectors, it would be better than doing them with scalar elements.

Virtually all of linear algebra deals with matrix multiplications of some kind, and it is worthwhile to spend some time trying to develop an intuitive understanding of the viewpoints presented here.

In addition to this, it is useful to know a few basic properties of matrix multiplication at a higher level:

- Matrix multiplication is associative:  $(AB)C = A(BC)$ .
- Matrix multiplication is distributive:  $A(B + C) = AB + AC$ .
- Matrix multiplication is, in general, *not* commutative; that is, it can be the case that  $AB \neq BA$ . (For example, if  $A \in \mathbb{R}^{m \times n}$  and  $B \in \mathbb{R}^{n \times q}$ , the matrix product  $BA$  does not even exist if  $m$  and  $q$  are not equal!)

If you are not familiar with these properties, take the time to verify them for yourself. For example, to check the associativity of matrix multiplication, suppose that  $A \in \mathbb{R}^{m \times n}$ ,  $B \in \mathbb{R}^{n \times p}$ , and  $C \in \mathbb{R}^{p \times q}$ . Note that  $AB \in \mathbb{R}^{m \times p}$ , so  $(AB)C \in \mathbb{R}^{m \times q}$ . Similarly,  $BC \in \mathbb{R}^{n \times q}$ , so  $A(BC) \in \mathbb{R}^{m \times q}$ . Thus, the dimensions of the resulting matrices agree. To show that matrix multiplication is associative, it suffices to check that the  $(i, j)$ th entry of  $(AB)C$  is equal to the  $(i, j)$ th entry of  $A(BC)$ . We can verify this directly using the definition of matrix multiplication:

$$\begin{aligned} ((AB)C)_{ij} &= \sum_{k=1}^p (AB)_{ik} C_{kj} = \sum_{k=1}^p \left( \sum_{l=1}^n A_{il} B_{lk} \right) C_{kj} \\ &= \sum_{k=1}^p \left( \sum_{l=1}^n A_{il} B_{lk} C_{kj} \right) = \sum_{l=1}^n \left( \sum_{k=1}^p A_{il} B_{lk} C_{kj} \right) \\ &= \sum_{l=1}^n A_{il} \left( \sum_{k=1}^p B_{lk} C_{kj} \right) = \sum_{l=1}^n A_{il} (BC)_{lj} = (A(BC))_{ij}. \end{aligned}$$

Here, the first and last two equalities simply use the definition of matrix multiplication, the third and fifth equalities use the distributive property for *scalar multiplication over addition*, and the fourth equality uses the *commutative and associativity of scalar addition*. This technique for proving matrix properties by reduction to simple scalar properties will come up often, so make sure you're familiar with it.

### 3 Operations and Properties

In this section we present several operations and properties of matrices and vectors. Hopefully a great deal of this will be review for you, so the notes can just serve as a reference for these topics.

### 3.1 The Identity Matrix and Diagonal Matrices

The ***identity matrix***, denoted  $I \in \mathbb{R}^{n \times n}$ , is a square matrix with ones on the diagonal and zeros everywhere else. That is,

$$I_{ij} = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases}$$

It has the property that for all  $A \in \mathbb{R}^{m \times n}$ ,

$$AI = A = IA.$$

Note that in some sense, the notation for the identity matrix is ambiguous, since it does not specify the dimension of  $I$ . Generally, the dimensions of  $I$  are inferred from context so as to make matrix multiplication possible. For example, in the equation above, the  $I$  in  $AI = A$  is an  $n \times n$  matrix, whereas the  $I$  in  $A = IA$  is an  $m \times m$  matrix.

A ***diagonal matrix*** is a matrix where all non-diagonal elements are 0. This is typically denoted  $D = \text{diag}(d_1, d_2, \dots, d_n)$ , with

$$D_{ij} = \begin{cases} d_i & i = j \\ 0 & i \neq j \end{cases}$$

Clearly,  $I = \text{diag}(1, 1, \dots, 1)$ .

### 3.2 The Transpose

The ***transpose*** of a matrix results from “flipping” the rows and columns. Given a matrix  $A \in \mathbb{R}^{m \times n}$ , its transpose, written  $A^T \in \mathbb{R}^{n \times m}$ , is the  $n \times m$  matrix whose entries are given by

$$(A^T)_{ij} = A_{ji}.$$

We have in fact already been using the transpose when describing row vectors, since the transpose of a column vector is naturally a row vector.

The following properties of transposes are easily verified:

- $(A^T)^T = A$
- $(AB)^T = B^T A^T$
- $(A + B)^T = A^T + B^T$

### 3.3 Symmetric Matrices

A square matrix  $A \in \mathbb{R}^{n \times n}$  is ***symmetric*** if  $A = A^T$ . It is ***anti-symmetric*** if  $A = -A^T$ . It is easy to show that for any matrix  $A \in \mathbb{R}^{n \times n}$ , the matrix  $A + A^T$  is symmetric and the

matrix  $A - A^T$  is anti-symmetric. From this it follows that any square matrix  $A \in \mathbb{R}^{n \times n}$  can be represented as a sum of a symmetric matrix and an anti-symmetric matrix, since

$$A = \frac{1}{2}(A + A^T) + \frac{1}{2}(A - A^T)$$

and the first matrix on the right is symmetric, while the second is anti-symmetric. It turns out that symmetric matrices occur a great deal in practice, and they have many nice properties which we will look at shortly. It is common to denote the set of all symmetric matrices of size  $n$  as  $\mathbb{S}^n$ , so that  $A \in \mathbb{S}^n$  means that  $A$  is a symmetric  $n \times n$  matrix;

### 3.4 The Trace

The **trace** of a square matrix  $A \in \mathbb{R}^{n \times n}$ , denoted  $\text{tr}(A)$  (or just  $\text{tr}A$  if the parentheses are obviously implied), is the sum of diagonal elements in the matrix:

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

As described in the CS229 lecture notes, the trace has the following properties (included here for the sake of completeness):

- For  $A \in \mathbb{R}^{n \times n}$ ,  $\text{tr}A = \text{tr}A^T$ .
- For  $A, B \in \mathbb{R}^{n \times n}$ ,  $\text{tr}(A + B) = \text{tr}A + \text{tr}B$ .
- For  $A \in \mathbb{R}^{n \times n}$ ,  $t \in \mathbb{R}$ ,  $\text{tr}(tA) = t \text{tr}A$ .
- For  $A, B$  such that  $AB$  is square,  $\text{tr}AB = \text{tr}BA$ .
- For  $A, B, C$  such that  $ABC$  is square,  $\text{tr}ABC = \text{tr}BCA = \text{tr}CAB$ , and so on for the product of more matrices.

As an example of how these properties can be proven, we'll consider the fourth property given above. Suppose that  $A \in \mathbb{R}^{m \times n}$  and  $B \in \mathbb{R}^{n \times m}$  (so that  $AB \in \mathbb{R}^{m \times m}$  is a square matrix). Observe that  $BA \in \mathbb{R}^{n \times n}$  is also a square matrix, so it makes sense to apply the trace operator to it. To verify that  $\text{tr}AB = \text{tr}BA$ , note that

$$\begin{aligned} \text{tr}AB &= \sum_{i=1}^m (AB)_{ii} = \sum_{i=1}^m \left( \sum_{j=1}^n A_{ij} B_{ji} \right) \\ &= \sum_{i=1}^m \sum_{j=1}^n A_{ij} B_{ji} = \sum_{j=1}^n \sum_{i=1}^m B_{ji} A_{ij} \\ &= \sum_{j=1}^n \left( \sum_{i=1}^m B_{ji} A_{ij} \right) = \sum_{j=1}^n (BA)_{jj} = \text{tr}BA. \end{aligned}$$

Here, the first and last two equalities use the definition of the trace operator and matrix multiplication. The fourth equality, where the main work occurs, uses the commutativity of scalar multiplication in order to reverse the order of the terms in each product, and the commutativity and associativity of scalar addition in order to rearrange the order of the summation.

### 3.5 Norms

A **norm** of a vector  $\|x\|$  is informally a measure of the “length” of the vector. For example, we have the commonly-used Euclidean or  $\ell_2$  norm,

$$\|x\|_2 = \sqrt{\sum_{i=1}^n x_i^2}.$$

Note that  $\|x\|_2^2 = x^T x$ .

More formally, a norm is any function  $f : \mathbb{R}^n \rightarrow \mathbb{R}$  that satisfies 4 properties:

1. For all  $x \in \mathbb{R}^n$ ,  $f(x) \geq 0$  (non-negativity).
2.  $f(x) = 0$  if and only if  $x = 0$  (definiteness).
3. For all  $x \in \mathbb{R}^n$ ,  $t \in \mathbb{R}$ ,  $f(tx) = |t|f(x)$  (homogeneity).
4. For all  $x, y \in \mathbb{R}^n$ ,  $f(x + y) \leq f(x) + f(y)$  (triangle inequality).

Other examples of norms are the  $\ell_1$  norm,

$$\|x\|_1 = \sum_{i=1}^n |x_i|$$

and the  $\ell_\infty$  norm,

$$\|x\|_\infty = \max_i |x_i|.$$

In fact, all three norms presented so far are examples of the family of  $\ell_p$  norms, which are parameterized by a real number  $p \geq 1$ , and defined as

$$\|x\|_p = \left( \sum_{i=1}^n |x_i|^p \right)^{1/p}.$$

Norms can also be defined for matrices, such as the Frobenius norm,

$$\|A\|_F = \sqrt{\sum_{i=1}^m \sum_{j=1}^n A_{ij}^2} = \sqrt{\text{tr}(A^T A)}.$$

Many other norms exist, but they are beyond the scope of this review.

### 3.6 Linear Independence and Rank

A set of vectors  $\{x_1, x_2, \dots, x_n\} \subset \mathbb{R}^m$  is said to be **(linearly) independent** if no vector can be represented as a linear combination of the remaining vectors. Conversely, if one vector belonging to the set *can* be represented as a linear combination of the remaining vectors, then the vectors are said to be **(linearly) dependent**. That is, if

$$x_n = \sum_{i=1}^{n-1} \alpha_i x_i$$

for some scalar values  $\alpha_1, \dots, \alpha_{n-1} \in \mathbb{R}$ , then we say that the vectors  $x_1, \dots, x_n$  are linearly dependent; otherwise, the vectors are linearly independent. For example, the vectors

$$x_1 = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \quad x_2 = \begin{bmatrix} 4 \\ 1 \\ 5 \end{bmatrix} \quad x_3 = \begin{bmatrix} 2 \\ -3 \\ -1 \end{bmatrix}$$

are linearly dependent because  $x_3 = -2x_1 + x_2$ .

The **column rank** of a matrix  $A \in \mathbb{R}^{m \times n}$  is the size of the largest subset of columns of  $A$  that constitute a linearly independent set. With some abuse of terminology, this is often referred to simply as the number of linearly independent columns of  $A$ . In the same way, the **row rank** is the largest number of rows of  $A$  that constitute a linearly independent set.

For any matrix  $A \in \mathbb{R}^{m \times n}$ , it turns out that the column rank of  $A$  is equal to the row rank of  $A$  (though we will not prove this), and so both quantities are referred to collectively as the **rank** of  $A$ , denoted as  $\text{rank}(A)$ . The following are some basic properties of the rank:

- For  $A \in \mathbb{R}^{m \times n}$ ,  $\text{rank}(A) \leq \min(m, n)$ . If  $\text{rank}(A) = \min(m, n)$ , then  $A$  is said to be **full rank**.
- For  $A \in \mathbb{R}^{m \times n}$ ,  $\text{rank}(A) = \text{rank}(A^T)$ .
- For  $A \in \mathbb{R}^{m \times n}$ ,  $B \in \mathbb{R}^{n \times p}$ ,  $\text{rank}(AB) \leq \min(\text{rank}(A), \text{rank}(B))$ .
- For  $A, B \in \mathbb{R}^{m \times n}$ ,  $\text{rank}(A + B) \leq \text{rank}(A) + \text{rank}(B)$ .

### 3.7 The Inverse of a Square Matrix

The **inverse** of a square matrix  $A \in \mathbb{R}^{n \times n}$  is denoted  $A^{-1}$ , and is the unique matrix such that

$$A^{-1}A = I = AA^{-1}.$$

Note that not all matrices have inverses. Non-square matrices, for example, do not have inverses by definition. However, for some square matrices  $A$ , it may still be the case that

$A^{-1}$  may not exist. In particular, we say that  $A$  is **invertible** or **non-singular** if  $A^{-1}$  exists and **non-invertible** or **singular** otherwise.<sup>2</sup>

In order for a square matrix  $A$  to have an inverse  $A^{-1}$ , then  $A$  must be full rank. We will soon see that there are many alternative sufficient and necessary conditions, in addition to full rank, for invertibility.

The following are properties of the inverse; all assume that  $A, B \in \mathbb{R}^{n \times n}$  are non-singular:

- $(A^{-1})^{-1} = A$
- $(AB)^{-1} = B^{-1}A^{-1}$
- $(A^{-1})^T = (A^T)^{-1}$ . For this reason this matrix is often denoted  $A^{-T}$ .

As an example of how the inverse is used, consider the linear system of equations,  $Ax = b$  where  $A \in \mathbb{R}^{n \times n}$ , and  $x, b \in \mathbb{R}^n$ . If  $A$  is nonsingular (i.e., invertible), then  $x = A^{-1}b$ .

(What if  $A \in \mathbb{R}^{m \times n}$  is not a square matrix? Does this work?)

### 3.8 Orthogonal Matrices

Two vectors  $x, y \in \mathbb{R}^n$  are **orthogonal** if  $x^T y = 0$ . A vector  $x \in \mathbb{R}^n$  is **normalized** if  $\|x\|_2 = 1$ . A square matrix  $U \in \mathbb{R}^{n \times n}$  is **orthogonal** (note the different meanings when talking about vectors versus matrices) if all its columns are orthogonal to each other and are normalized (the columns are then referred to as being **orthonormal**).

It follows immediately from the definition of orthogonality and normality that

$$U^T U = I = U U^T.$$

In other words, the inverse of an orthogonal matrix is its transpose. Note that if  $U$  is not square — i.e.,  $U \in \mathbb{R}^{m \times n}$ ,  $n < m$  — but its columns are still orthonormal, then  $U^T U = I$ , but  $U U^T \neq I$ . We generally only use the term orthogonal to describe the previous case, where  $U$  is square.

Another nice property of orthogonal matrices is that operating on a vector with an orthogonal matrix will not change its Euclidean norm, i.e.,

$$\|Ux\|_2 = \|x\|_2 \tag{3}$$

for any  $x \in \mathbb{R}^n$ ,  $U \in \mathbb{R}^{n \times n}$  orthogonal.

---

<sup>2</sup>It's easy to get confused and think that non-singular means non-invertible. But in fact, it means the opposite! Watch out!

### 3.9 Range and Nullspace of a Matrix

The **span** of a set of vectors  $\{x_1, x_2, \dots, x_n\}$  is the set of all vectors that can be expressed as a linear combination of  $\{x_1, \dots, x_n\}$ . That is,

$$\text{span}(\{x_1, \dots, x_n\}) = \left\{ v : v = \sum_{i=1}^n \alpha_i x_i, \quad \alpha_i \in \mathbb{R} \right\}.$$

It can be shown that if  $\{x_1, \dots, x_n\}$  is a set of  $n$  linearly independent vectors, where each  $x_i \in \mathbb{R}^n$ , then  $\text{span}(\{x_1, \dots, x_n\}) = \mathbb{R}^n$ . In other words, *any* vector  $v \in \mathbb{R}^n$  can be written as a linear combination of  $x_1$  through  $x_n$ . The **projection** of a vector  $y \in \mathbb{R}^m$  onto the span of  $\{x_1, \dots, x_n\}$  (here we assume  $x_i \in \mathbb{R}^m$ ) is the vector  $v \in \text{span}(\{x_1, \dots, x_n\})$ , such that  $v$  is as close as possible to  $y$ , as measured by the Euclidean norm  $\|v - y\|_2$ . We denote the projection as  $\text{Proj}(y; \{x_1, \dots, x_n\})$  and can define it formally as,

$$\text{Proj}(y; \{x_1, \dots, x_n\}) = \underset{v \in \text{span}(\{x_1, \dots, x_n\})}{\text{argmin}} \|y - v\|_2.$$

The **range** (sometimes also called the columnspace) of a matrix  $A \in \mathbb{R}^{m \times n}$ , denoted  $\mathcal{R}(A)$ , is the span of the columns of  $A$ . In other words,

$$\mathcal{R}(A) = \{v \in \mathbb{R}^m : v = Ax, x \in \mathbb{R}^n\}.$$

Making a few technical assumptions (namely that  $A$  is full rank and that  $n < m$ ), the projection of a vector  $y \in \mathbb{R}^m$  onto the range of  $A$  is given by,

$$\text{Proj}(y; A) = \underset{v \in \mathcal{R}(A)}{\text{argmin}} \|v - y\|_2 = A(A^T A)^{-1} A^T y .$$

This last equation should look extremely familiar, since it is almost the same formula we derived in class (and which we will soon derive again) for the least squares estimation of parameters. Looking at the definition for the projection, it should not be too hard to convince yourself that this is in fact the same objective that we minimized in our least squares problem (except for a squaring of the norm, which doesn't affect the optimal point) and so these problems are naturally very connected. When  $A$  contains only a single column,  $a \in \mathbb{R}^m$ , this gives the special case for a projection of a vector on to a line:

$$\text{Proj}(y; a) = \frac{aa^T}{a^T a} y .$$

The **nullspace** of a matrix  $A \in \mathbb{R}^{m \times n}$ , denoted  $\mathcal{N}(A)$  is the set of all vectors that equal 0 when multiplied by  $A$ , i.e.,

$$\mathcal{N}(A) = \{x \in \mathbb{R}^n : Ax = 0\}.$$

Note that vectors in  $\mathcal{R}(A)$  are of size  $m$ , while vectors in the  $\mathcal{N}(A)$  are of size  $n$ , so vectors in  $\mathcal{R}(A^T)$  and  $\mathcal{N}(A)$  are both in  $\mathbb{R}^n$ . In fact, we can say much more. It turns out that

$$\{w : w = u + v, u \in \mathcal{R}(A^T), v \in \mathcal{N}(A)\} = \mathbb{R}^n \text{ and } \mathcal{R}(A^T) \cap \mathcal{N}(A) = \{\mathbf{0}\} .$$

In other words,  $\mathcal{R}(A^T)$  and  $\mathcal{N}(A)$  are disjoint subsets that together span the entire space of  $\mathbb{R}^n$ . Sets of this type are called **orthogonal complements**, and we denote this  $\mathcal{R}(A^T) = \mathcal{N}(A)^\perp$ .

## 3.10 The Determinant

The **determinant** of a square matrix  $A \in \mathbb{R}^{n \times n}$ , is a function  $\det : \mathbb{R}^{n \times n} \rightarrow \mathbb{R}$ , and is denoted  $|A|$  or  $\det A$  (like the trace operator, we usually omit parentheses). Algebraically, one could write down an explicit formula for the determinant of  $A$ , but this unfortunately gives little intuition about its meaning. Instead, we'll start out by providing a geometric interpretation of the determinant and then visit some of its specific algebraic properties afterwards.

Given a matrix

$$\begin{bmatrix} - & a_1^T & - \\ - & a_2^T & - \\ \vdots & & \\ - & a_n^T & - \end{bmatrix},$$

consider the set of points  $S \subset \mathbb{R}^n$  formed by taking all possible linear combinations of the row vectors  $a_1, \dots, a_n \in \mathbb{R}^n$  of  $A$ , where the coefficients of the linear combination are all between 0 and 1; that is, the set  $S$  is the restriction of  $\text{span}(\{a_1, \dots, a_n\})$  to only those linear combinations whose coefficients  $\alpha_1, \dots, \alpha_n$  satisfy  $0 \leq \alpha_i \leq 1$ ,  $i = 1, \dots, n$ . Formally,

$$S = \{v \in \mathbb{R}^n : v = \sum_{i=1}^n \alpha_i a_i \text{ where } 0 \leq \alpha_i \leq 1, i = 1, \dots, n\}.$$

The absolute value of the determinant of  $A$ , it turns out, is a measure of the “volume” of the set  $S$ .<sup>3</sup>

For example, consider the  $2 \times 2$  matrix,

$$A = \begin{bmatrix} 1 & 3 \\ 3 & 2 \end{bmatrix}. \quad (4)$$

Here, the rows of the matrix are

$$a_1 = \begin{bmatrix} 1 \\ 3 \end{bmatrix} \quad a_2 = \begin{bmatrix} 3 \\ 2 \end{bmatrix}.$$

The set  $S$  corresponding to these rows is shown in Figure 3.10. For two-dimensional matrices,  $S$  generally has the shape of a *parallelogram*. In our example, the value of the determinant is  $|A| = -7$  (as can be computed using the formulas shown later in this section), so the area of the parallelogram is 7. (Verify this for yourself!)

In three dimensions, the set  $S$  corresponds to an object known as a *parallelepiped* (a three-dimensional box with skewed sides, such that every face has the shape of a parallelogram). The absolute value of the determinant of the  $3 \times 3$  matrix whose rows define  $S$  give the three-dimensional volume of the parallelepiped. In even higher dimensions, the set  $S$  is an object known as an  $n$ -dimensional *parallelotope*.

---

<sup>3</sup>Admittedly, we have not actually defined what we mean by “volume” here, but hopefully the intuition should be clear enough. When  $n = 2$ , our notion of “volume” corresponds to the area of  $S$  in the Cartesian plane. When  $n = 3$ , “volume” corresponds with our usual notion of volume for a three-dimensional object.

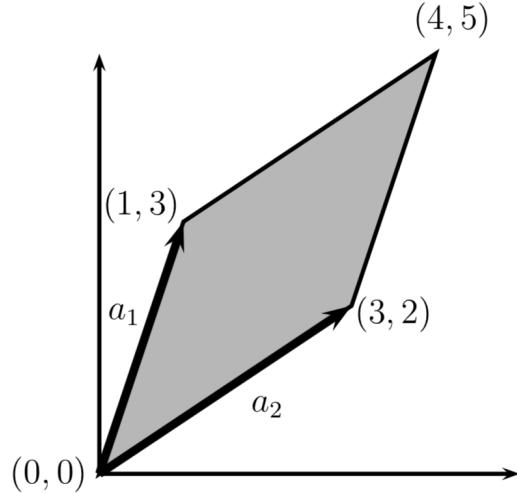


Figure 1: Illustration of the determinant for the  $2 \times 2$  matrix  $A$  given in (4). Here,  $a_1$  and  $a_2$  are vectors corresponding to the rows of  $A$ , and the set  $S$  corresponds to the shaded region (i.e., the parallelogram). The absolute value of the determinant,  $|\det A| = 7$ , is the area of the parallelogram.

Algebraically, the determinant satisfies the following three properties (from which all other properties follow, including the general formula):

1. The determinant of the identity is 1,  $|I| = 1$ . (Geometrically, the volume of a unit hypercube is 1).
2. Given a matrix  $A \in \mathbb{R}^{n \times n}$ , if we multiply a single row in  $A$  by a scalar  $t \in \mathbb{R}$ , then the determinant of the new matrix is  $t|A|$ ,

$$\left| \begin{bmatrix} - & t a_1^T & - \\ - & a_2^T & - \\ \vdots & & \\ - & a_m^T & - \end{bmatrix} \right| = t|A|.$$

(Geometrically, multiplying one of the sides of the set  $S$  by a factor  $t$  causes the volume to increase by a factor  $t$ .)

3. If we exchange any two rows  $a_i^T$  and  $a_j^T$  of  $A$ , then the determinant of the new matrix is  $-|A|$ , for example

$$\left| \begin{bmatrix} - & a_2^T & - \\ - & a_1^T & - \\ \vdots & & \\ - & a_m^T & - \end{bmatrix} \right| = -|A|.$$

In case you are wondering, it is not immediately obvious that a function satisfying the above three properties exists. In fact, though, such a function does exist, and is unique (which we will not prove here).

Several properties that follow from the three properties above include:

- For  $A \in \mathbb{R}^{n \times n}$ ,  $|A| = |A^T|$ .
- For  $A, B \in \mathbb{R}^{n \times n}$ ,  $|AB| = |A||B|$ .
- For  $A \in \mathbb{R}^{n \times n}$ ,  $|A| = 0$  if and only if  $A$  is singular (i.e., non-invertible). (If  $A$  is singular then it does not have full rank, and hence its columns are linearly dependent. In this case, the set  $S$  corresponds to a “flat sheet” within the  $n$ -dimensional space and hence has zero volume.)
- For  $A \in \mathbb{R}^{n \times n}$  and  $A$  non-singular,  $|A^{-1}| = 1/|A|$ .

Before giving the general definition for the determinant, we define, for  $A \in \mathbb{R}^{n \times n}$ ,  $A_{\setminus i, \setminus j} \in \mathbb{R}^{(n-1) \times (n-1)}$  to be the *matrix* that results from deleting the  $i$ th row and  $j$ th column from  $A$ . The general (recursive) formula for the determinant is

$$\begin{aligned} |A| &= \sum_{i=1}^n (-1)^{i+j} a_{ij} |A_{\setminus i, \setminus j}| \quad (\text{for any } j \in 1, \dots, n) \\ &= \sum_{j=1}^n (-1)^{i+j} a_{ij} |A_{\setminus i, \setminus j}| \quad (\text{for any } i \in 1, \dots, n) \end{aligned}$$

with the initial case that  $|A| = a_{11}$  for  $A \in \mathbb{R}^{1 \times 1}$ . If we were to expand this formula completely for  $A \in \mathbb{R}^{n \times n}$ , there would be a total of  $n!$  ( $n$  factorial) different terms. For this reason, we hardly ever explicitly write the complete equation of the determinant for matrices bigger than  $3 \times 3$ . However, the equations for determinants of matrices up to size  $3 \times 3$  are fairly common, and it is good to know them:

$$\begin{aligned} |[a_{11}]| &= a_{11} \\ \left| \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \right| &= a_{11}a_{22} - a_{12}a_{21} \\ \left| \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \right| &= a_{11}a_{22}a_{33} + a_{12}a_{23}a_{31} + a_{13}a_{21}a_{32} \\ &\quad - a_{11}a_{23}a_{32} - a_{12}a_{21}a_{33} - a_{13}a_{22}a_{31} \end{aligned}$$

The ***classical adjoint*** (often just called the adjoint) of a matrix  $A \in \mathbb{R}^{n \times n}$ , is denoted  $\text{adj}(A)$ , and defined as

$$\text{adj}(A) \in \mathbb{R}^{n \times n}, \quad (\text{adj}(A))_{ij} = (-1)^{i+j} |A_{\setminus j, \setminus i}|$$

(note the switch in the indices  $A_{\setminus j \setminus i}$ ). It can be shown that for any nonsingular  $A \in \mathbb{R}^{n \times n}$ ,

$$A^{-1} = \frac{1}{|A|} \text{adj}(A) .$$

While this is a nice “explicit” formula for the inverse of matrix, we should note that, numerically, there are in fact much more efficient ways of computing the inverse.

### 3.11 Quadratic Forms and Positive Semidefinite Matrices

Given a square matrix  $A \in \mathbb{R}^{n \times n}$  and a vector  $x \in \mathbb{R}^n$ , the scalar value  $x^T A x$  is called a **quadratic form**. Written explicitly, we see that

$$x^T A x = \sum_{i=1}^n x_i (Ax)_i = \sum_{i=1}^n x_i \left( \sum_{j=1}^n A_{ij} x_j \right) = \sum_{i=1}^n \sum_{j=1}^n A_{ij} x_i x_j .$$

Note that,

$$x^T A x = (x^T A x)^T = x^T A^T x = x^T \left( \frac{1}{2} A + \frac{1}{2} A^T \right) x,$$

where the first equality follows from the fact that the transpose of a scalar is equal to itself, and the second equality follows from the fact that we are averaging two quantities which are themselves equal. From this, we can conclude that only the symmetric part of  $A$  contributes to the quadratic form. For this reason, we often implicitly assume that the matrices appearing in a quadratic form are symmetric.

We give the following definitions:

- A symmetric matrix  $A \in \mathbb{S}^n$  is **positive definite** (PD) if for all non-zero vectors  $x \in \mathbb{R}^n$ ,  $x^T A x > 0$ . This is usually denoted  $A \succ 0$  (or just  $A > 0$ ), and often times the set of all positive definite matrices is denoted  $\mathbb{S}_{++}^n$ .
- A symmetric matrix  $A \in \mathbb{S}^n$  is **positive semidefinite** (PSD) if for all vectors  $x^T A x \geq 0$ . This is written  $A \succeq 0$  (or just  $A \geq 0$ ), and the set of all positive semidefinite matrices is often denoted  $\mathbb{S}_+^n$ .
- Likewise, a symmetric matrix  $A \in \mathbb{S}^n$  is **negative definite** (ND), denoted  $A \prec 0$  (or just  $A < 0$ ) if for all non-zero  $x \in \mathbb{R}^n$ ,  $x^T A x < 0$ .
- Similarly, a symmetric matrix  $A \in \mathbb{S}^n$  is **negative semidefinite** (NSD), denoted  $A \preceq 0$  (or just  $A \leq 0$ ) if for all  $x \in \mathbb{R}^n$ ,  $x^T A x \leq 0$ .
- Finally, a symmetric matrix  $A \in \mathbb{S}^n$  is **indefinite**, if it is neither positive semidefinite nor negative semidefinite — i.e., if there exists  $x_1, x_2 \in \mathbb{R}^n$  such that  $x_1^T A x_1 > 0$  and  $x_2^T A x_2 < 0$ .

It should be obvious that if  $A$  is positive definite, then  $-A$  is negative definite and vice versa. Likewise, if  $A$  is positive semidefinite then  $-A$  is negative semidefinite and vice versa. If  $A$  is indefinite, then so is  $-A$ .

One important property of positive definite and negative definite matrices is that they are always full rank, and hence, invertible. To see why this is the case, suppose that some matrix  $A \in \mathbb{R}^{n \times n}$  is not full rank. Then, suppose that the  $j$ th column of  $A$  is expressible as a linear combination of other  $n - 1$  columns:

$$a_j = \sum_{i \neq j} x_i a_i,$$

for some  $x_1, \dots, x_{j-1}, x_{j+1}, \dots, x_n \in \mathbb{R}$ . Setting  $x_j = -1$ , we have

$$Ax = \sum_{i=1}^n x_i a_i = 0.$$

But this implies  $x^T Ax = 0$  for some non-zero vector  $x$ , so  $A$  must be neither positive definite nor negative definite. Therefore, if  $A$  is either positive definite or negative definite, it must be full rank.

Finally, there is one type of positive definite matrix that comes up frequently, and so deserves some special mention. Given any matrix  $A \in \mathbb{R}^{m \times n}$  (not necessarily symmetric or even square), the matrix  $G = A^T A$  (sometimes called a **Gram matrix**) is always positive semidefinite. Further, if  $m \geq n$  (and we assume for convenience that  $A$  is full rank), then  $G = A^T A$  is positive definite.

### 3.12 Eigenvalues and Eigenvectors

Given a square matrix  $A \in \mathbb{R}^{n \times n}$ , we say that  $\lambda \in \mathbb{C}$  is an **eigenvalue** of  $A$  and  $x \in \mathbb{C}^n$  is the corresponding **eigenvector**<sup>4</sup> if

$$Ax = \lambda x, \quad x \neq 0.$$

Intuitively, this definition means that multiplying  $A$  by the vector  $x$  results in a new vector that points in the same direction as  $x$ , but scaled by a factor  $\lambda$ . Also note that for any eigenvector  $x \in \mathbb{C}^n$ , and scalar  $t \in \mathbb{C}$ ,  $A(cx) = cAx = c\lambda x = \lambda(cx)$ , so  $cx$  is also an eigenvector. For this reason when we talk about “the” eigenvector associated with  $\lambda$ , we usually assume that the eigenvector is normalized to have length 1 (this still creates some ambiguity, since  $x$  and  $-x$  will both be eigenvectors, but we will have to live with this).

We can rewrite the equation above to state that  $(\lambda, x)$  is an eigenvalue-eigenvector pair of  $A$  if,

$$(\lambda I - A)x = 0, \quad x \neq 0.$$

---

<sup>4</sup>Note that  $\lambda$  and the entries of  $x$  are actually in  $\mathbb{C}$ , the set of complex numbers, not just the reals; we will see shortly why this is necessary. Don’t worry about this technicality for now, you can think of complex vectors in the same way as real vectors.

But  $(\lambda I - A)x = 0$  has a non-zero solution to  $x$  if and only if  $(\lambda I - A)$  has a non-empty nullspace, which is only the case if  $(\lambda I - A)$  is singular, i.e.,

$$|(\lambda I - A)| = 0.$$

We can now use the previous definition of the determinant to expand this expression  $|(\lambda I - A)|$  into a (very large) polynomial in  $\lambda$ , where  $\lambda$  will have degree  $n$ . It's often called the characteristic polynomial of the matrix  $A$ .

We then find the  $n$  (possibly complex) roots of this characteristic polynomial and denote them by  $\lambda_1, \dots, \lambda_n$ . These are all the eigenvalues of the matrix  $A$ , but we note that they may not be distinct. To find the eigenvector corresponding to the eigenvalue  $\lambda_i$ , we simply solve the linear equation  $(\lambda_i I - A)x = 0$ , which is guaranteed to have a non-zero solution because  $\lambda_i I - A$  is singular (but there could also be multiple or infinite solutions.)

It should be noted that this is not the method which is actually used in practice to numerically compute the eigenvalues and eigenvectors (remember that the complete expansion of the determinant has  $n!$  terms); it is rather a mathematical argument.

The following are properties of eigenvalues and eigenvectors (in all cases assume  $A \in \mathbb{R}^{n \times n}$  has eigenvalues  $\lambda_1, \dots, \lambda_n$ ):

- The trace of  $A$  is equal to the sum of its eigenvalues,

$$\text{tr}A = \sum_{i=1}^n \lambda_i.$$

- The determinant of  $A$  is equal to the product of its eigenvalues,

$$|A| = \prod_{i=1}^n \lambda_i.$$

- The rank of  $A$  is equal to the number of non-zero eigenvalues of  $A$ .
- Suppose  $A$  is non-singular with eigenvalue  $\lambda$  and an associated eigenvector  $x$ . Then  $1/\lambda$  is an eigenvalue of  $A^{-1}$  with an associated eigenvector  $x$ , i.e.,  $A^{-1}x = (1/\lambda)x$ . (To prove this, take the eigenvector equation,  $Ax = \lambda x$  and left-multiply each side by  $A^{-1}$ .)
- The eigenvalues of a diagonal matrix  $D = \text{diag}(d_1, \dots, d_n)$  are just the diagonal entries  $d_1, \dots, d_n$ .

### 3.13 Eigenvalues and Eigenvectors of Symmetric Matrices

In general, the structures of the eigenvalues and eigenvectors of a general square matrix can be subtle to characterize. Fortunately, in most of the cases in machine learning, it suffices to deal with symmetric real matrices, whose eigenvalues and eigenvectors have remarkable properties.

Throughout this section, let's assume that  $A$  is a symmetric real matrix. We have the following properties:

1. All eigenvalues of  $A$  are real numbers. We denote them by  $\lambda_1, \dots, \lambda_n$ .
2. There exists a set of eigenvectors  $u_1, \dots, u_n$  such that a) for all  $i$ ,  $u_i$  is an eigenvector with eigenvalue  $\lambda_i$  and b)  $u_1, \dots, u_n$  are unit vectors and orthogonal to each other.<sup>5</sup>

Let  $U$  be the orthonormal matrix that contains  $u_i$ 's as columns:<sup>6</sup>

$$U = \begin{bmatrix} | & | & & | \\ u_1 & u_2 & \cdots & u_n \\ | & | & & | \end{bmatrix} \quad (5)$$

Let  $\Lambda = \text{diag}(\lambda_1, \dots, \lambda_n)$  be the diagonal matrix that contains  $\lambda_1, \dots, \lambda_n$  as entries on the diagonal. Using the view of matrix-matrix vector multiplication in equation (2) of Section 2.3, we can verify that

$$AU = \begin{bmatrix} | & | & & | \\ Au_1 & Au_2 & \cdots & Au_n \\ | & | & & | \end{bmatrix} = \begin{bmatrix} | & | & & | \\ \lambda_1 u_1 & \lambda_2 u_2 & \cdots & \lambda_n u_n \\ | & | & & | \end{bmatrix} = U \text{diag}(\lambda_1, \dots, \lambda_n) = U\Lambda$$

Recalling that orthonormal matrix  $U$  satisfies that  $UU^T = I$  and using the equation above, we have

$$A = AUU^T = U\Lambda U^T \quad (6)$$

This new presentation of  $A$  as  $U\Lambda U^T$  is often called the diagonalization of the matrix  $A$ . The term diagonalization comes from the fact that with such representation, we can often effectively treat a symmetric matrix  $A$  as a diagonal matrix — which is much easier to understand — w.r.t the basis defined by the eigenvectors  $U$ . We will elaborate this below by several examples.

**Background: representing vector w.r.t. another basis.** Any orthonormal matrix  $U = \begin{bmatrix} | & | & & | \\ u_1 & u_2 & \cdots & u_n \\ | & | & & | \end{bmatrix}$  defines a new basis (coordinate system) of  $\mathbb{R}^n$  in the following sense. For any vector  $x \in \mathbb{R}^n$  can be represented as a linear combination of  $u_1, \dots, u_n$  with coefficient  $\hat{x}_1, \dots, \hat{x}_n$ :

$$x = \hat{x}_1 u_1 + \cdots + \hat{x}_n u_n = U\hat{x}$$

---

<sup>5</sup>Mathematically, we have  $\forall i, Au_i = \lambda_i u_i$ ,  $\|u_i\|_2 = 1$ , and  $\forall j \neq i, u_i^T u_j = 0$ . Moreover, we remark that it's not true that all eigenvectors  $u_1, \dots, u_n$  satisfying a) of any matrix  $A$  are orthogonal to each other, because the eigenvalues can be repetitive and so can eigenvectors.

<sup>6</sup>Here for notational simplicity, we deviate from the notational convention for columns of matrices in the previous sections.

where in the second equality we use the view of equation (1). Indeed, such  $\hat{x}$  uniquely exists

$$x = U\hat{x} \Leftrightarrow U^T x = \hat{x}$$

In other words, the vector  $\hat{x} = U^T x$  can serve as another representation of the vector  $x$  w.r.t the basis defined by  $U$ .

**“Diagonalizing” matrix-vector multiplication.** With the setup above, we will see that left-multiplying matrix  $A$  can be viewed as left-multiplying a diagonal matrix w.r.t the basis of the eigenvectors. Suppose  $x$  is a vector and  $\hat{x}$  is its representation w.r.t to the basis of  $U$ . Let  $z = Ax$  be the matrix-vector product. Now let’s compute the representation  $z$  w.r.t the basis of  $U$ :

Then, again using the fact that  $UU^T = U^T U = I$  and equation (6), we have that

$$\hat{z} = U^T z = U^T Ax = U^T U \Lambda U^T x = \Lambda \hat{x} = \begin{bmatrix} \lambda_1 \hat{x}_1 \\ \lambda_2 \hat{x}_2 \\ \vdots \\ \lambda_n \hat{x}_n \end{bmatrix}$$

We see that left-multiplying matrix  $A$  in the original space is equivalent to left-multiplying the diagonal matrix  $\Lambda$  w.r.t the new basis, which is merely scaling each coordinate by the corresponding eigenvalue.

Under the new basis, multiplying a matrix multiple times becomes much simpler as well. For example, suppose  $q = AAAx$ . Deriving out the analytical form of  $q$  in terms of the entries of  $A$  may be a nightmare under the original basis, but can be much easier under the new on:

$$\hat{q} = U^T q = U^T AAAx = U^T U \Lambda U^T U \Lambda U^T U \Lambda U^T x = \Lambda^3 \hat{x} = \begin{bmatrix} \lambda_1^3 \hat{x}_1 \\ \lambda_2^3 \hat{x}_2 \\ \vdots \\ \lambda_n^3 \hat{x}_n \end{bmatrix} \quad (7)$$

**“Diagonalizing” quadratic form.** As a directly corollary, the quadratic form  $x^T Ax$  can also be simplified under the new basis

$$x^T Ax = x^T U \Lambda U^T x = \hat{x}^T \Lambda \hat{x} = \sum_{i=1}^n \lambda_i \hat{x}_i^2 \quad (8)$$

(Recall that with the old representation,  $x^T Ax = \sum_{i=1, j=1}^n x_i x_j A_{ij}$  involves a sum of  $n^2$  terms instead of  $n$  terms in the equation above.) With this viewpoint, we can also show that the definiteness of the matrix  $A$  depends entirely on the sign of its eigenvalues:

1. If all  $\lambda_i > 0$ , then the matrix  $A$  is positive definite because  $x^T A x = \sum_{i=1}^n \lambda_i \hat{x}_i^2 > 0$  for any  $\hat{x} \neq 0$ .<sup>7</sup>
2. If all  $\lambda_i \geq 0$ , it is positive semidefinite because  $x^T A x = \sum_{i=1}^n \lambda_i \hat{x}_i^2 \geq 0$  for all  $\hat{x}$ .
3. Likewise, if all  $\lambda_i < 0$  or  $\lambda_i \leq 0$ , then  $A$  is negative definite or negative semidefinite respectively.
4. Finally, if  $A$  has both positive and negative eigenvalues, say  $\lambda_i > 0$  and  $\lambda_j < 0$ , then it is indefinite. This is because if we let  $\hat{x}$  satisfy  $\hat{x}_i = 1$  and  $\hat{x}_k = 0, \forall k \neq i$ , then  $x^T A x = \sum_{i=1}^n \lambda_i \hat{x}_i^2 > 0$ . Similarly we can let  $\hat{x}$  satisfy  $\hat{x}_j = 1$  and  $\hat{x}_k = 0, \forall k \neq j$ , then  $x^T A x = \sum_{i=1}^n \lambda_i \hat{x}_i^2 < 0$ .<sup>8</sup>

An application where eigenvalues and eigenvectors come up frequently is in maximizing some function of a matrix. In particular, for a matrix  $A \in \mathbb{S}^n$ , consider the following maximization problem,

$$\max_{x \in \mathbb{R}^n} x^T A x = \sum_{i=1}^n \lambda_i \hat{x}_i^2 \quad \text{subject to } \|x\|_2^2 = 1 \quad (9)$$

i.e., we want to find the vector (of norm 1) which maximizes the quadratic form. Assuming the eigenvalues are ordered as  $\lambda_1 \geq \lambda_2 \geq \dots \geq \lambda_n$ , the optimal value of this optimization problem is  $\lambda_1$  and any eigenvector  $u_1$  corresponding to  $\lambda_1$  is one of the maximizers. (If  $\lambda_1 > \lambda_2$ , then there is a unique eigenvector corresponding to eigenvalue  $\lambda_1$ , which is the unique maximizer of the optimization problem (9).)

We can show this by using the diagonalization technique: Note that  $\|x\|_2 = \|\hat{x}\|_2$  by equation (3), and using equation (8), we can rewrite the optimization (9) as

$$\max_{\hat{x} \in \mathbb{R}^n} \hat{x}^T \Lambda \hat{x} = \sum_{i=1}^n \lambda_i \hat{x}_i^2 \quad \text{subject to } \|\hat{x}\|_2^2 = 1 \quad (10)$$

Then, we have that the objective is upper bounded by  $\lambda_1$ :

$$\hat{x}^T \Lambda \hat{x} = \sum_{i=1}^n \lambda_i \hat{x}_i^2 \leq \sum_{i=1}^n \lambda_1 \hat{x}_i^2 = \lambda_1 \quad (11)$$

Moreover, setting  $\hat{x} = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$  achieves the equality in the equation above, and this corresponds to setting  $x = u_1$ .

---

<sup>7</sup>Note that  $\hat{x} \neq 0 \Leftrightarrow x \neq 0$ .

<sup>8</sup>Note that  $x = U\hat{x}$  and therefore constructing  $\hat{x}$  gives an implicit construction of  $x$ .

## 4 Matrix Calculus

While the topics in the previous sections are typically covered in a standard course on linear algebra, one topic that does not seem to be covered very often (and which we will use extensively) is the extension of calculus to the vector setting. Despite the fact that all the actual calculus we use is relatively trivial, the notation can often make things look much more difficult than they are. In this section we present some basic definitions of matrix calculus and provide a few examples.

### 4.1 The Gradient

Suppose that  $f : \mathbb{R}^{m \times n} \rightarrow \mathbb{R}$  is a function that takes as input a matrix  $A$  of size  $m \times n$  and returns a real value. Then the **gradient** of  $f$  (with respect to  $A \in \mathbb{R}^{m \times n}$ ) is the matrix of partial derivatives, defined as:

$$\nabla_A f(A) \in \mathbb{R}^{m \times n} = \begin{bmatrix} \frac{\partial f(A)}{\partial A_{11}} & \frac{\partial f(A)}{\partial A_{12}} & \dots & \frac{\partial f(A)}{\partial A_{1n}} \\ \frac{\partial f(A)}{\partial A_{21}} & \frac{\partial f(A)}{\partial A_{22}} & \dots & \frac{\partial f(A)}{\partial A_{2n}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f(A)}{\partial A_{m1}} & \frac{\partial f(A)}{\partial A_{m2}} & \dots & \frac{\partial f(A)}{\partial A_{mn}} \end{bmatrix}$$

i.e., an  $m \times n$  matrix with

$$(\nabla_A f(A))_{ij} = \frac{\partial f(A)}{\partial A_{ij}}.$$

Note that the size of  $\nabla_A f(A)$  is always the same as the size of  $A$ . So if, in particular,  $A$  is just a vector  $x \in \mathbb{R}^n$ ,

$$\nabla_x f(x) = \begin{bmatrix} \frac{\partial f(x)}{\partial x_1} \\ \frac{\partial f(x)}{\partial x_2} \\ \vdots \\ \frac{\partial f(x)}{\partial x_n} \end{bmatrix}.$$

It is very important to remember that the gradient of a function is *only* defined if the function is real-valued, that is, if it returns a scalar value. We can not, for example, take the gradient of  $Ax$ ,  $A \in \mathbb{R}^{n \times n}$  with respect to  $x$ , since this quantity is vector-valued.

It follows directly from the equivalent properties of partial derivatives that:

- $\nabla_x(f(x) + g(x)) = \nabla_x f(x) + \nabla_x g(x).$
- For  $t \in \mathbb{R}$ ,  $\nabla_x(t f(x)) = t \nabla_x f(x).$

In principle, gradients are a natural extension of partial derivatives to functions of multiple variables. In practice, however, working with gradients can sometimes be tricky for notational reasons. For example, suppose that  $A \in \mathbb{R}^{m \times n}$  is a matrix of fixed coefficients

and suppose that  $b \in \mathbb{R}^m$  is a vector of fixed coefficients. Let  $f : \mathbb{R}^m \rightarrow \mathbb{R}$  be the function defined by  $f(z) = z^T z$ , such that  $\nabla_z f(z) = 2z$ . But now, consider the expression,

$$\nabla f(Ax).$$

How should this expression be interpreted? There are at least two possibilities:

1. In the first interpretation, recall that  $\nabla_z f(z) = 2z$ . Here, we interpret  $\nabla f(Ax)$  as evaluating the gradient at the point  $Ax$ , hence,

$$\nabla f(Ax) = 2(Ax) = 2Ax \in \mathbb{R}^m.$$

2. In the second interpretation, we consider the quantity  $f(Ax)$  as a function of the input variables  $x$ . More formally, let  $g(x) = f(Ax)$ . Then in this interpretation,

$$\nabla f(Ax) = \nabla_x g(x) \in \mathbb{R}^n.$$

Here, we can see that these two interpretations are indeed different. One interpretation yields an  $m$ -dimensional vector as a result, while the other interpretation yields an  $n$ -dimensional vector as a result! How can we resolve this?

Here, the key is to make explicit the variables which we are differentiating with respect to. In the first case, we are differentiating the function  $f$  with respect to its arguments  $z$  and then substituting the argument  $Ax$ . In the second case, we are differentiating the composite function  $g(x) = f(Ax)$  with respect to  $x$  directly. We denote the first case as  $\nabla_z f(Ax)$  and the second case as  $\nabla_x f(Ax)$ .<sup>9</sup> Keeping the notation clear is extremely important (as you'll find out in your homework, in fact!).

## 4.2 The Hessian

Suppose that  $f : \mathbb{R}^n \rightarrow \mathbb{R}$  is a function that takes a vector in  $\mathbb{R}^n$  and returns a real number. Then the **Hessian** matrix with respect to  $x$ , written  $\nabla_x^2 f(x)$  or simply as  $H$  is the  $n \times n$  matrix of partial derivatives,

$$\nabla_x^2 f(x) \in \mathbb{R}^{n \times n} = \begin{bmatrix} \frac{\partial^2 f(x)}{\partial x_1^2} & \frac{\partial^2 f(x)}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f(x)}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f(x)}{\partial x_2 \partial x_1} & \frac{\partial^2 f(x)}{\partial x_2^2} & \cdots & \frac{\partial^2 f(x)}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f(x)}{\partial x_n \partial x_1} & \frac{\partial^2 f(x)}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f(x)}{\partial x_n^2} \end{bmatrix}.$$

---

<sup>9</sup>A drawback to this notation that we will have to live with is the fact that in the first case,  $\nabla_z f(Ax)$  it appears that we are differentiating with respect to a variable that does not even appear in the expression being differentiated! For this reason, the first case is often written as  $\nabla f(Ax)$ , and the fact that we are differentiating with respect to the arguments of  $f$  is understood. However, the second case is *always* written as  $\nabla_x f(Ax)$ .

In other words,  $\nabla_x^2 f(x) \in \mathbb{R}^{n \times n}$ , with

$$(\nabla_x^2 f(x))_{ij} = \frac{\partial^2 f(x)}{\partial x_i \partial x_j}.$$

Note that the Hessian is always symmetric, since

$$\frac{\partial^2 f(x)}{\partial x_i \partial x_j} = \frac{\partial^2 f(x)}{\partial x_j \partial x_i}.$$

Similar to the gradient, the Hessian is defined only when  $f(x)$  is real-valued.

It is natural to think of the gradient as the analogue of the first derivative for functions of vectors, and the Hessian as the analogue of the second derivative (and the symbols we use also suggest this relation). This intuition is generally correct, but there are a few caveats to keep in mind.

First, for real-valued functions of one variable  $f : \mathbb{R} \rightarrow \mathbb{R}$ , it is a basic definition that the second derivative is the derivative of the first derivative, i.e.,

$$\frac{\partial^2 f(x)}{\partial x^2} = \frac{\partial}{\partial x} \frac{\partial}{\partial x} f(x).$$

However, for functions of a vector, the gradient of the function is a vector, and we cannot take the gradient of a vector — i.e.,

$$\nabla_x \nabla_x f(x) = \nabla_x \begin{bmatrix} \frac{\partial f(x)}{\partial x_1} \\ \frac{\partial f(x)}{\partial x_2} \\ \vdots \\ \frac{\partial f(x)}{\partial x_n} \end{bmatrix}$$

and this expression is not defined. Therefore, it is *not* the case that the Hessian is the gradient of the gradient. However, this is *almost* true, in the following sense: If we look at the  $i$ th entry of the gradient  $(\nabla_x f(x))_i = \partial f(x)/\partial x_i$ , and take the gradient with respect to  $x$  we get

$$\nabla_x \frac{\partial f(x)}{\partial x_i} = \begin{bmatrix} \frac{\partial^2 f(x)}{\partial x_i \partial x_1} \\ \frac{\partial^2 f(x)}{\partial x_i \partial x_2} \\ \vdots \\ \frac{\partial^2 f(x)}{\partial x_i \partial x_n} \end{bmatrix}$$

which is the  $i$ th column (or row) of the Hessian. Therefore,

$$\nabla_x^2 f(x) = \begin{bmatrix} \nabla_x(\nabla_x f(x))_1 & \nabla_x(\nabla_x f(x))_2 & \cdots & \nabla_x(\nabla_x f(x))_n \end{bmatrix}.$$

If we don't mind being a little bit sloppy we can say that (essentially)  $\nabla_x^2 f(x) = \nabla_x(\nabla_x f(x))^T$ , so long as we understand that this really means taking the gradient of each entry of  $(\nabla_x f(x))^T$ , not the gradient of the whole vector.

Finally, note that while we can take the gradient with respect to a matrix  $A \in \mathbb{R}^{n \times n}$ , for the purposes of this class we will only consider taking the Hessian with respect to a vector  $x \in \mathbb{R}^n$ . This is simply a matter of convenience (and the fact that none of the calculations we do require us to find the Hessian with respect to a matrix), since the Hessian with respect to a matrix would have to represent all the partial derivatives  $\partial^2 f(A)/(\partial A_{ij} \partial A_{kl})$ , and it is rather cumbersome to represent this as a matrix.

### 4.3 Gradients and Hessians of Quadratic and Linear Functions

Now let's try to determine the gradient and Hessian matrices for a few simple functions. It should be noted that all the gradients given here are special cases of the gradients given in the CS229 lecture notes.

For  $x \in \mathbb{R}^n$ , let  $f(x) = b^T x$  for some known vector  $b \in \mathbb{R}^n$ . Then

$$f(x) = \sum_{i=1}^n b_i x_i$$

so

$$\frac{\partial f(x)}{\partial x_k} = \frac{\partial}{\partial x_k} \sum_{i=1}^n b_i x_i = b_k.$$

From this we can easily see that  $\nabla_x b^T x = b$ . This should be compared to the analogous situation in single variable calculus, where  $\partial/(\partial x) ax = a$ .

Now consider the quadratic function  $f(x) = x^T A x$  for  $A \in \mathbb{S}^n$ . Remember that

$$f(x) = \sum_{i=1}^n \sum_{j=1}^n A_{ij} x_i x_j.$$

To take the partial derivative, we'll consider the terms including  $x_k$  and  $x_k^2$  factors separately:

$$\begin{aligned} \frac{\partial f(x)}{\partial x_k} &= \frac{\partial}{\partial x_k} \sum_{i=1}^n \sum_{j=1}^n A_{ij} x_i x_j \\ &= \frac{\partial}{\partial x_k} \left[ \sum_{i \neq k} \sum_{j \neq k} A_{ij} x_i x_j + \sum_{i \neq k} A_{ik} x_i x_k + \sum_{j \neq k} A_{kj} x_k x_j + A_{kk} x_k^2 \right] \\ &= \sum_{i \neq k} A_{ik} x_i + \sum_{j \neq k} A_{kj} x_j + 2A_{kk} x_k \\ &= \sum_{i=1}^n A_{ik} x_i + \sum_{j=1}^n A_{kj} x_j = 2 \sum_{i=1}^n A_{ki} x_i, \end{aligned}$$

where the last equality follows since  $A$  is symmetric (which we can safely assume, since it is appearing in a quadratic form). Note that the  $k$ th entry of  $\nabla_x f(x)$  is just the inner product

of the  $k$ th row of  $A$  and  $x$ . Therefore,  $\nabla_x x^T A x = 2Ax$ . Again, this should remind you of the analogous fact in single-variable calculus, that  $\partial/(\partial x) ax^2 = 2ax$ .

Finally, let's look at the Hessian of the quadratic function  $f(x) = x^T Ax$  (it should be obvious that the Hessian of a linear function  $b^T x$  is zero). In this case,

$$\frac{\partial^2 f(x)}{\partial x_k \partial x_\ell} = \frac{\partial}{\partial x_k} \left[ \frac{\partial f(x)}{\partial x_\ell} \right] = \frac{\partial}{\partial x_k} \left[ 2 \sum_{i=1}^n A_{\ell i} x_i \right] = 2A_{\ell k} = 2A_{k\ell}.$$

Therefore, it should be clear that  $\nabla_x^2 x^T Ax = 2A$ , which should be entirely expected (and again analogous to the single-variable fact that  $\partial^2/(\partial x^2) ax^2 = 2a$ ).

To recap,

- $\nabla_x b^T x = b$
- $\nabla_x x^T Ax = 2Ax$  (if  $A$  symmetric)
- $\nabla_x^2 x^T Ax = 2A$  (if  $A$  symmetric)

## 4.4 Least Squares

Let's apply the equations we obtained in the last section to derive the least squares equations. Suppose we are given matrices  $A \in \mathbb{R}^{m \times n}$  (for simplicity we assume  $A$  is full rank) and a vector  $b \in \mathbb{R}^m$  such that  $b \notin \mathcal{R}(A)$ . In this situation we will not be able to find a vector  $x \in \mathbb{R}^n$ , such that  $Ax = b$ , so instead we want to find a vector  $x$  such that  $Ax$  is as close as possible to  $b$ , as measured by the square of the Euclidean norm  $\|Ax - b\|_2^2$ .

Using the fact that  $\|x\|_2^2 = x^T x$ , we have

$$\begin{aligned} \|Ax - b\|_2^2 &= (Ax - b)^T (Ax - b) \\ &= x^T A^T Ax - 2b^T Ax + b^T b \end{aligned}$$

Taking the gradient with respect to  $x$  we have, and using the properties we derived in the previous section

$$\begin{aligned} \nabla_x (x^T A^T Ax - 2b^T Ax + b^T b) &= \nabla_x x^T A^T Ax - \nabla_x 2b^T Ax + \nabla_x b^T b \\ &= 2A^T Ax - 2A^T b \end{aligned}$$

Setting this last expression equal to zero and solving for  $x$  gives the normal equations

$$x = (A^T A)^{-1} A^T b$$

which is the same as what we derived in class.

## 4.5 Gradients of the Determinant

Now let's consider a situation where we find the gradient of a function with respect to a matrix, namely for  $A \in \mathbb{R}^{n \times n}$ , we want to find  $\nabla_A |A|$ . Recall from our discussion of determinants that

$$|A| = \sum_{i=1}^n (-1)^{i+j} A_{ij} |A_{\setminus i, \setminus j}| \quad (\text{for any } j \in 1, \dots, n)$$

so

$$\frac{\partial}{\partial A_{k\ell}} |A| = \frac{\partial}{\partial A_{k\ell}} \sum_{i=1}^n (-1)^{i+j} A_{ij} |A_{\setminus i, \setminus j}| = (-1)^{k+\ell} |A_{\setminus k, \setminus \ell}| = (\text{adj}(A))_{\ell k}.$$

From this it immediately follows from the properties of the adjoint that

$$\nabla_A |A| = (\text{adj}(A))^T = |A| A^{-T}.$$

Now let's consider the function  $f : \mathbb{S}_{++}^n \rightarrow \mathbb{R}$ ,  $f(A) = \log |A|$ . Note that we have to restrict the domain of  $f$  to be the positive definite matrices, since this ensures that  $|A| > 0$ , so that the log of  $|A|$  is a real number. In this case we can use the chain rule (nothing fancy, just the ordinary chain rule from single-variable calculus) to see that

$$\frac{\partial \log |A|}{\partial A_{ij}} = \frac{\partial \log |A|}{\partial |A|} \frac{\partial |A|}{\partial A_{ij}} = \frac{1}{|A|} \frac{\partial |A|}{\partial A_{ij}}.$$

From this it should be obvious that

$$\nabla_A \log |A| = \frac{1}{|A|} \nabla_A |A| = A^{-1},$$

where we can drop the transpose in the last expression because  $A$  is symmetric. Note the similarity to the single-valued case, where  $\partial/(\partial x) \log x = 1/x$ .

## 4.6 Eigenvalues as Optimization

Finally, we use matrix calculus to solve an optimization problem in a way that leads directly to eigenvalue/eigenvector analysis. Consider the following, equality constrained optimization problem:

$$\max_{x \in \mathbb{R}^n} x^T A x \quad \text{subject to } \|x\|_2^2 = 1$$

for a symmetric matrix  $A \in \mathbb{S}^n$ . A standard way of solving optimization problems with equality constraints is by forming the **Lagrangian**, an objective function that includes the equality constraints.<sup>10</sup> The Lagrangian in this case can be given by

$$\mathcal{L}(x, \lambda) = x^T A x - \lambda(x^T x - 1)$$

---

<sup>10</sup>Don't worry if you haven't seen Lagrangians before, as we will cover them in greater detail later in CS229.

where  $\lambda$  is called the Lagrange multiplier associated with the equality constraint. It can be established that for  $x^*$  to be a optimal point to the problem, the gradient of the Lagrangian has to be zero at  $x^*$  (this is not the only condition, but it is required). That is,

$$\nabla_x \mathcal{L}(x, \lambda) = \nabla_x(x^T A x - \lambda x^T x) = 2A^T x - 2\lambda x = 0.$$

Notice that this is just the linear equation  $Ax = \lambda x$ . This shows that the only points which can possibly maximize (or minimize)  $x^T A x$  assuming  $x^T x = 1$  are the eigenvectors of  $A$ .

---

# Review of Probability Theory

---

Arian Maleki and Tom Do  
Stanford University

Probability theory is the study of uncertainty. Through this class, we will be relying on concepts from probability theory for deriving machine learning algorithms. These notes attempt to cover the basics of probability theory at a level appropriate for CS 229. The mathematical theory of probability is very sophisticated, and delves into a branch of analysis known as **measure theory**. In these notes, we provide a basic treatment of probability that does not address these finer details.

## 1 Elements of probability

In order to define a probability on a set we need a few basic elements,

- **Sample space**  $\Omega$ : The set of all the outcomes of a random experiment. Here, each outcome  $\omega \in \Omega$  can be thought of as a complete description of the state of the real world at the end of the experiment.
- **Set of events (or event space)**  $\mathcal{F}$ : A set whose elements  $A \in \mathcal{F}$  (called **events**) are subsets of  $\Omega$  (i.e.,  $A \subseteq \Omega$  is a collection of possible outcomes of an experiment).<sup>1</sup>.
- **Probability measure**: A function  $P : \mathcal{F} \rightarrow \mathbb{R}$  that satisfies the following properties,
  - $P(A) \geq 0$ , for all  $A \in \mathcal{F}$
  - $P(\Omega) = 1$
  - If  $A_1, A_2, \dots$  are disjoint events (i.e.,  $A_i \cap A_j = \emptyset$  whenever  $i \neq j$ ), then

$$P(\cup_i A_i) = \sum_i P(A_i)$$

These three properties are called the **Axioms of Probability**.

**Example:** Consider the event of tossing a six-sided die. The sample space is  $\Omega = \{1, 2, 3, 4, 5, 6\}$ . We can define different event spaces on this sample space. For example, the simplest event space is the trivial event space  $\mathcal{F} = \{\emptyset, \Omega\}$ . Another event space is the set of all subsets of  $\Omega$ . For the first event space, the unique probability measure satisfying the requirements above is given by  $P(\emptyset) = 0, P(\Omega) = 1$ . For the second event space, one valid probability measure is to assign the probability of each set in the event space to be  $\frac{i}{6}$  where  $i$  is the number of elements of that set; for example,  $P(\{1, 2, 3, 4\}) = \frac{4}{6}$  and  $P(\{1, 2, 3\}) = \frac{3}{6}$ .

### Properties:

- If  $A \subseteq B \implies P(A) \leq P(B)$ .
- $P(A \cap B) \leq \min(P(A), P(B))$ .
- (Union Bound)  $P(A \cup B) \leq P(A) + P(B)$ .
- $P(\Omega \setminus A) = 1 - P(A)$ .
- (Law of Total Probability) If  $A_1, \dots, A_k$  are a set of disjoint events such that  $\cup_{i=1}^k A_i = \Omega$ , then  $\sum_{i=1}^k P(A_i) = 1$ .

---

<sup>1</sup>  $\mathcal{F}$  should satisfy three properties: (1)  $\emptyset \in \mathcal{F}$ ; (2)  $A \in \mathcal{F} \implies \Omega \setminus A \in \mathcal{F}$ ; and (3)  $A_1, A_2, \dots \in \mathcal{F} \implies \cup_i A_i \in \mathcal{F}$ .

## 1.1 Conditional probability and independence

Let  $B$  be an event with non-zero probability. The conditional probability of any event  $A$  given  $B$  is defined as,

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

In other words,  $P(A|B)$  is the probability measure of the event  $A$  after observing the occurrence of event  $B$ . Two events are called independent if and only if  $P(A \cap B) = P(A)P(B)$  (or equivalently,  $P(A|B) = P(A)$ ). Therefore, independence is equivalent to saying that observing  $B$  does not have any effect on the probability of  $A$ .

## 2 Random variables

Consider an experiment in which we flip 10 coins, and we want to know the number of coins that come up heads. Here, the elements of the sample space  $\Omega$  are 10-length sequences of heads and tails. For example, we might have  $w_0 = \langle H, H, T, H, T, H, H, T, T, T \rangle \in \Omega$ . However, in practice, we usually do not care about the probability of obtaining any particular sequence of heads and tails. Instead we usually care about real-valued functions of outcomes, such as the number of heads that appear among our 10 tosses, or the length of the longest run of tails. These functions, under some technical conditions, are known as **random variables**.

More formally, a random variable  $X$  is a function  $X : \Omega \rightarrow \mathbb{R}$ .<sup>2</sup> Typically, we will denote random variables using upper case letters  $X(\omega)$  or more simply  $X$  (where the dependence on the random outcome  $\omega$  is implied). We will denote the value that a random variable may take on using lower case letters  $x$ .

**Example:** In our experiment above, suppose that  $X(\omega)$  is the number of heads which occur in the sequence of tosses  $\omega$ . Given that only 10 coins are tossed,  $X(\omega)$  can take only a finite number of values, so it is known as a **discrete random variable**. Here, the probability of the set associated with a random variable  $X$  taking on some specific value  $k$  is

$$P(X = k) := P(\{\omega : X(\omega) = k\}).$$

**Example:** Suppose that  $X(\omega)$  is a random variable indicating the amount of time it takes for a radioactive particle to decay. In this case,  $X(\omega)$  takes on a infinite number of possible values, so it is called a **continuous random variable**. We denote the probability that  $X$  takes on a value between two real constants  $a$  and  $b$  (where  $a < b$ ) as

$$P(a \leq X \leq b) := P(\{\omega : a \leq X(\omega) \leq b\}).$$

### 2.1 Cumulative distribution functions

In order to specify the probability measures used when dealing with random variables, it is often convenient to specify alternative functions (CDFs, PDFs, and PMFs) from which the probability measure governing an experiment immediately follows. In this section and the next two sections, we describe each of these types of functions in turn.

A **cumulative distribution function (CDF)** is a function  $F_X : \mathbb{R} \rightarrow [0, 1]$  which specifies a probability measure as,

$$F_X(x) \triangleq P(X \leq x). \quad (1)$$

By using this function one can calculate the probability of any event in  $\mathcal{F}$ .<sup>3</sup> Figure ?? shows a sample CDF function.

#### Properties:

---

<sup>2</sup>Technically speaking, not every function is not acceptable as a random variable. From a measure-theoretic perspective, random variables must be Borel-measurable functions. Intuitively, this restriction ensures that given a random variable and its underlying outcome space, one can implicitly define the each of the events of the event space as being sets of outcomes  $\omega \in \Omega$  for which  $X(\omega)$  satisfies some property (e.g., the event  $\{\omega : X(\omega) \geq 3\}$ ).

<sup>3</sup>This is a remarkable fact and is actually a theorem that is proved in more advanced courses.

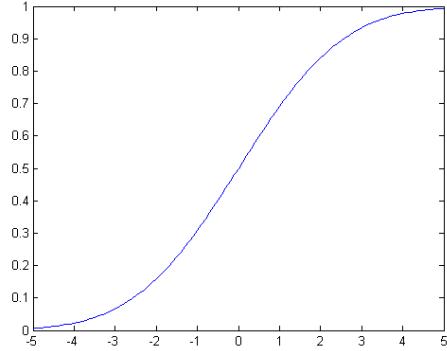


Figure 1: A cumulative distribution function (CDF).

- $0 \leq F_X(x) \leq 1$ .
- $\lim_{x \rightarrow -\infty} F_X(x) = 0$ .
- $\lim_{x \rightarrow \infty} F_X(x) = 1$ .
- $x \leq y \implies F_X(x) \leq F_X(y)$ .

## 2.2 Probability mass functions

When a random variable  $X$  takes on a finite set of possible values (i.e.,  $X$  is a discrete random variable), a simpler way to represent the probability measure associated with a random variable is to directly specify the probability of each value that the random variable can assume. In particular, a *probability mass function (PMF)* is a function  $p_X : \Omega \rightarrow \mathbb{R}$  such that

$$p_X(x) \triangleq P(X = x).$$

In the case of discrete random variable, we use the notation  $Val(X)$  for the set of possible values that the random variable  $X$  may assume. For example, if  $X(\omega)$  is a random variable indicating the number of heads out of ten tosses of coin, then  $Val(X) = \{0, 1, 2, \dots, 10\}$ .

### Properties:

- $0 \leq p_X(x) \leq 1$ .
- $\sum_{x \in Val(X)} p_X(x) = 1$ .
- $\sum_{x \in A} p_X(x) = P(X \in A)$ .

## 2.3 Probability density functions

For some continuous random variables, the cumulative distribution function  $F_X(x)$  is differentiable everywhere. In these cases, we define the **Probability Density Function or PDF** as the derivative of the CDF, i.e.,

$$f_X(x) \triangleq \frac{dF_X(x)}{dx}. \quad (2)$$

Note here, that the PDF for a continuous random variable may not always exist (i.e., if  $F_X(x)$  is not differentiable everywhere).

According to the properties of differentiation, for very small  $\Delta x$ ,

$$P(x \leq X \leq x + \Delta x) \approx f_X(x)\Delta x. \quad (3)$$

Both CDFs and PDFs (when they exist!) can be used for calculating the probabilities of different events. But it should be emphasized that the value of PDF at any given point  $x$  is not the probability

of that event, i.e.,  $f_X(x) \neq P(X = x)$ . For example,  $f_X(x)$  can take on values larger than one (but the integral of  $f_X(x)$  over any subset of  $\mathbb{R}$  will be at most one).

#### Properties:

- $f_X(x) \geq 0$ .
- $\int_{-\infty}^{\infty} f_X(x) = 1$ .
- $\int_{x \in A} f_X(x) dx = P(X \in A)$ .

### 2.4 Expectation

Suppose that  $X$  is a discrete random variable with PMF  $p_X(x)$  and  $g : \mathbb{R} \rightarrow \mathbb{R}$  is an arbitrary function. In this case,  $g(X)$  can be considered a random variable, and we define the **expectation** or **expected value** of  $g(X)$  as

$$E[g(X)] \triangleq \sum_{x \in Val(X)} g(x)p_X(x).$$

If  $X$  is a continuous random variable with PDF  $f_X(x)$ , then the expected value of  $g(X)$  is defined as,

$$E[g(X)] \triangleq \int_{-\infty}^{\infty} g(x)f_X(x)dx.$$

Intuitively, the expectation of  $g(X)$  can be thought of as a “weighted average” of the values that  $g(x)$  can take on for different values of  $x$ , where the weights are given by  $p_X(x)$  or  $f_X(x)$ . As a special case of the above, note that the expectation,  $E[X]$  of a random variable itself is found by letting  $g(x) = x$ ; this is also known as the **mean** of the random variable  $X$ .

#### Properties:

- $E[a] = a$  for any constant  $a \in \mathbb{R}$ .
- $E[af(X)] = aE[f(X)]$  for any constant  $a \in \mathbb{R}$ .
- (Linearity of Expectation)  $E[f(X) + g(X)] = E[f(X)] + E[g(X)]$ .
- For a discrete random variable  $X$ ,  $E[1\{X = k\}] = P(X = k)$ .

### 2.5 Variance

The **variance** of a random variable  $X$  is a measure of how concentrated the distribution of a random variable  $X$  is around its mean. Formally, the variance of a random variable  $X$  is defined as

$$Var[X] \triangleq E[(X - E(X))^2]$$

Using the properties in the previous section, we can derive an alternate expression for the variance:

$$\begin{aligned} E[(X - E[X])^2] &= E[X^2 - 2E[X]X + E[X]^2] \\ &= E[X^2] - 2E[X]E[X] + E[X]^2 \\ &= E[X^2] - E[X]^2, \end{aligned}$$

where the second equality follows from linearity of expectations and the fact that  $E[X]$  is actually a constant with respect to the outer expectation.

#### Properties:

- $Var[a] = 0$  for any constant  $a \in \mathbb{R}$ .
- $Var[af(X)] = a^2Var[f(X)]$  for any constant  $a \in \mathbb{R}$ .

**Example** Calculate the mean and the variance of the uniform random variable  $X$  with PDF  $f_X(x) = 1$ ,  $\forall x \in [0, 1]$ , 0 elsewhere.

$$E[X] = \int_{-\infty}^{\infty} xf_X(x)dx = \int_0^1 xdx = \frac{1}{2}.$$

$$E[X^2] = \int_{-\infty}^{\infty} x^2 f_X(x) dx = \int_0^1 x^2 dx = \frac{1}{3}.$$

$$Var[X] = E[X^2] - E[X]^2 = \frac{1}{3} - \frac{1}{4} = \frac{1}{12}.$$

**Example:** Suppose that  $g(x) = 1\{x \in A\}$  for some subset  $A \subseteq \Omega$ . What is  $E[g(X)]$ ?

Discrete case:

$$E[g(X)] = \sum_{x \in Val(X)} 1\{x \in A\} P_X(x) dx = \sum_{x \in A} P_X(x) dx = P(x \in A).$$

Continuous case:

$$E[g(X)] = \int_{-\infty}^{\infty} 1\{x \in A\} f_X(x) dx = \int_{x \in A} f_X(x) dx = P(x \in A).$$

## 2.6 Some common random variables

### Discrete random variables

- $X \sim Bernoulli(p)$  (where  $0 \leq p \leq 1$ ): one if a coin with heads probability  $p$  comes up heads, zero otherwise.

$$p(x) = \begin{cases} p & \text{if } p = 1 \\ 1 - p & \text{if } p = 0 \end{cases}$$

- $X \sim Binomial(n, p)$  (where  $0 \leq p \leq 1$ ): the number of heads in  $n$  independent flips of a coin with heads probability  $p$ .

$$p(x) = \binom{n}{x} p^x (1-p)^{n-x}$$

- $X \sim Geometric(p)$  (where  $p > 0$ ): the number of flips of a coin with heads probability  $p$  until the first heads.

$$p(x) = p(1-p)^{x-1}$$

- $X \sim Poisson(\lambda)$  (where  $\lambda > 0$ ): a probability distribution over the nonnegative integers used for modeling the frequency of rare events.

$$p(x) = e^{-\lambda} \frac{\lambda^x}{x!}$$

### Continuous random variables

- $X \sim Uniform(a, b)$  (where  $a < b$ ): equal probability density to every value between  $a$  and  $b$  on the real line.

$$f(x) = \begin{cases} \frac{1}{b-a} & \text{if } a \leq x \leq b \\ 0 & \text{otherwise} \end{cases}$$

- $X \sim Exponential(\lambda)$  (where  $\lambda > 0$ ): decaying probability density over the nonnegative reals.

$$f(x) = \begin{cases} \lambda e^{-\lambda x} & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

- $X \sim Normal(\mu, \sigma^2)$ : also known as the Gaussian distribution

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

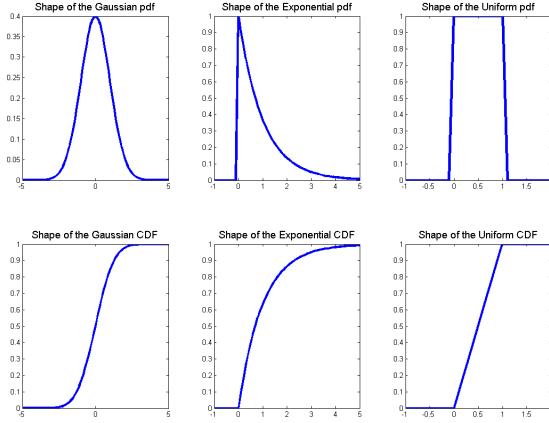


Figure 2: PDF and CDF of a couple of random variables.

The shape of the PDFs and CDFs of some of these random variables are shown in Figure ??.

The following table is the summary of some of the properties of these distributions.

Distribution	PDF or PMF	Mean	Variance
$Bernoulli(p)$	$\begin{cases} p, & \text{if } x = 1 \\ 1 - p, & \text{if } x = 0. \end{cases}$	$p$	$p(1 - p)$
$Binomial(n, p)$	$\binom{n}{k} p^k (1 - p)^{n-k} \text{ for } 0 \leq k \leq n$	$np$	$npq$
$Geometric(p)$	$p(1 - p)^{k-1} \text{ for } k = 1, 2, \dots$	$\frac{1}{p}$	$\frac{1-p}{p^2}$
$Poisson(\lambda)$	$e^{-\lambda} \lambda^x / x! \text{ for } k = 1, 2, \dots$	$\lambda$	$\lambda$
$Uniform(a, b)$	$\frac{1}{b-a} \forall x \in (a, b)$	$\frac{a+b}{2}$	$\frac{(b-a)^2}{12}$
$Gaussian(\mu, \sigma^2)$	$\frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$	$\mu$	$\sigma^2$
$Exponential(\lambda)$	$\lambda e^{-\lambda x} \quad x \geq 0, \lambda > 0$	$\frac{1}{\lambda}$	$\frac{1}{\lambda^2}$

### 3 Two random variables

Thus far, we have considered single random variables. In many situations, however, there may be more than one quantity that we are interested in knowing during a random experiment. For instance, in an experiment where we flip a coin ten times, we may care about both  $X(\omega) =$  the number of heads that come up as well as  $Y(\omega) =$  the length of the longest run of consecutive heads. In this section, we consider the setting of two random variables.

#### 3.1 Joint and marginal distributions

Suppose that we have two random variables  $X$  and  $Y$ . One way to work with these two random variables is to consider each of them separately. If we do that we will only need  $F_X(x)$  and  $F_Y(y)$ . But if we want to know about the values that  $X$  and  $Y$  assume simultaneously during outcomes of a random experiment, we require a more complicated structure known as the **joint cumulative distribution function** of  $X$  and  $Y$ , defined by

$$F_{XY}(x, y) = P(X \leq x, Y \leq y)$$

It can be shown that by knowing the joint cumulative distribution function, the probability of any event involving  $X$  and  $Y$  can be calculated.

The joint CDF  $F_{XY}(x, y)$  and the joint distribution functions  $F_X(x)$  and  $F_Y(y)$  of each variable separately are related by

$$\begin{aligned} F_X(x) &= \lim_{y \rightarrow \infty} F_{XY}(x, y) dy \\ F_Y(y) &= \lim_{x \rightarrow \infty} F_{XY}(x, y) dx. \end{aligned}$$

Here, we call  $F_X(x)$  and  $F_Y(y)$  the **marginal cumulative distribution functions** of  $F_{XY}(x, y)$ .

### Properties:

- $0 \leq F_{XY}(x, y) \leq 1$ .
- $\lim_{x, y \rightarrow \infty} F_{XY}(x, y) = 1$ .
- $\lim_{x, y \rightarrow -\infty} F_{XY}(x, y) = 0$ .
- $F_X(x) = \lim_{y \rightarrow \infty} F_{XY}(x, y)$ .

## 3.2 Joint and marginal probability mass functions

If  $X$  and  $Y$  are discrete random variables, then the **joint probability mass function**  $p_{XY} : \mathbb{R} \times \mathbb{R} \rightarrow [0, 1]$  is defined by

$$p_{XY}(x, y) = P(X = x, Y = y).$$

Here,  $0 \leq p_{XY}(x, y) \leq 1$  for all  $x, y$ , and  $\sum_{x \in Val(X)} \sum_{y \in Val(Y)} p_{XY}(x, y) = 1$ .

How does the joint PMF over two variables relate to the probability mass function for each variable separately? It turns out that

$$p_X(x) = \sum_y p_{XY}(x, y).$$

and similarly for  $p_Y(y)$ . In this case, we refer to  $p_X(x)$  as the **marginal probability mass function** of  $X$ . In statistics, the process of forming the marginal distribution with respect to one variable by summing out the other variable is often known as “marginalization.”

## 3.3 Joint and marginal probability density functions

Let  $X$  and  $Y$  be two continuous random variables with joint distribution function  $F_{XY}$ . In the case that  $F_{XY}(x, y)$  is everywhere differentiable in both  $x$  and  $y$ , then we can define the **joint probability density function**,

$$f_{XY}(x, y) = \frac{\partial^2 F_{XY}(x, y)}{\partial x \partial y}.$$

Like in the single-dimensional case,  $f_{XY}(x, y) \neq P(X = x, Y = y)$ , but rather

$$\iint_{x \in A} f_{XY}(x, y) dx dy = P((X, Y) \in A).$$

Note that the values of the probability density function  $f_{XY}(x, y)$  are always nonnegative, but they may be greater than 1. Nonetheless, it must be the case that  $\int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f_{XY}(x, y) dx dy = 1$ .

Analogous to the discrete case, we define

$$f_X(x) = \int_{-\infty}^{\infty} f_{XY}(x, y) dy,$$

as the **marginal probability density function** (or **marginal density**) of  $X$ , and similarly for  $f_Y(y)$ .

### 3.4 Conditional distributions

Conditional distributions seek to answer the question, what is the probability distribution over  $Y$ , when we know that  $X$  must take on a certain value  $x$ ? In the discrete case, the conditional probability mass function of  $X$  given  $Y$  is simply

$$p_{Y|X}(y|x) = \frac{p_{XY}(x,y)}{p_X(x)},$$

assuming that  $p_X(x) \neq 0$ .

In the continuous case, the situation is technically a little more complicated because the probability that a continuous random variable  $X$  takes on a specific value  $x$  is equal to zero<sup>4</sup>. Ignoring this technical point, we simply define, by analogy to the discrete case, the *conditional probability density* of  $Y$  given  $X = x$  to be

$$f_{Y|X}(y|x) = \frac{f_{XY}(x,y)}{f_X(x)},$$

provided  $f_X(x) \neq 0$ .

### 3.5 Bayes's rule

A useful formula that often arises when trying to derive expression for the conditional probability of one variable given another, is **Bayes's rule**.

In the case of discrete random variables  $X$  and  $Y$ ,

$$P_{Y|X}(y|x) = \frac{P_{XY}(x,y)}{P_X(x)} = \frac{P_{X|Y}(x|y)P_Y(y)}{\sum_{y' \in Val(Y)} P_{X|Y}(x|y')P_Y(y')}.$$

If the random variables  $X$  and  $Y$  are continuous,

$$f_{Y|X}(y|x) = \frac{f_{XY}(x,y)}{f_X(x)} = \frac{f_{X|Y}(x|y)f_Y(y)}{\int_{-\infty}^{\infty} f_{X|Y}(x|y')f_Y(y')dy'}.$$

### 3.6 Independence

Two random variables  $X$  and  $Y$  are **independent** if  $F_{XY}(x,y) = F_X(x)F_Y(y)$  for all values of  $x$  and  $y$ . Equivalently,

- For discrete random variables,  $p_{XY}(x,y) = p_X(x)p_Y(y)$  for all  $x \in Val(X)$ ,  $y \in Val(Y)$ .
- For discrete random variables,  $p_{Y|X}(y|x) = p_Y(y)$  whenever  $p_X(x) \neq 0$  for all  $y \in Val(Y)$ .
- For continuous random variables,  $f_{XY}(x,y) = f_X(x)f_Y(y)$  for all  $x, y \in \mathbb{R}$ .
- For continuous random variables,  $f_{Y|X}(y|x) = f_Y(y)$  whenever  $f_X(x) \neq 0$  for all  $y \in \mathbb{R}$ .

---

<sup>4</sup>To get around this, a more reasonable way to calculate the conditional CDF is,

$$F_{Y|X}(y,x) = \lim_{\Delta x \rightarrow 0} P(Y \leq y | x \leq X \leq x + \Delta x).$$

It can be easily seen that if  $F(x,y)$  is differentiable in both  $x, y$  then,

$$F_{Y|X}(y,x) = \int_{-\infty}^y \frac{f_{X,Y}(x,\alpha)}{f_X(x)} d\alpha$$

and therefore we define the conditional PDF of  $Y$  given  $X = x$  in the following way,

$$f_{Y|X}(y|x) = \frac{f_{X,Y}(x,y)}{f_X(x)}$$

Informally, two random variables  $X$  and  $Y$  are **independent** if “knowing” the value of one variable will never have any effect on the conditional probability distribution of the other variable, that is, you know all the information about the pair  $(X, Y)$  by just knowing  $f(x)$  and  $f(y)$ . The following lemma formalizes this observation:

**Lemma 3.1.** *If  $X$  and  $Y$  are independent then for any subsets  $A, B \subseteq \mathbb{R}$ , we have,*

$$P(X \in A, Y \in B) = P(X \in A)P(Y \in B)$$

By using the above lemma one can prove that if  $X$  is independent of  $Y$  then any function of  $X$  is independent of any function of  $Y$ .

### 3.7 Expectation and covariance

Suppose that we have two discrete random variables  $X, Y$  and  $g : \mathbf{R}^2 \rightarrow \mathbf{R}$  is a function of these two random variables. Then the expected value of  $g$  is defined in the following way,

$$E[g(X, Y)] \triangleq \sum_{x \in Val(X)} \sum_{y \in Val(Y)} g(x, y)p_{XY}(x, y).$$

For continuous random variables  $X, Y$ , the analogous expression is

$$E[g(X, Y)] = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} g(x, y)f_{XY}(x, y)dxdy.$$

We can use the concept of expectation to study the relationship of two random variables with each other. In particular, the **covariance** of two random variables  $X$  and  $Y$  is defined as

$$Cov[X, Y] \triangleq E[(X - E[X])(Y - E[Y])]$$

Using an argument similar to that for variance, we can rewrite this as,

$$\begin{aligned} Cov[X, Y] &= E[(X - E[X])(Y - E[Y])] \\ &= E[XY - XE[Y] - YE[X] + E[X]E[Y]] \\ &= E[XY] - E[X]E[Y] - E[Y]E[X] + E[X]E[Y] \\ &= E[XY] - E[X]E[Y]. \end{aligned}$$

Here, the key step in showing the equality of the two forms of covariance is in the third equality, where we use the fact that  $E[X]$  and  $E[Y]$  are actually constants which can be pulled out of the expectation. When  $Cov[X, Y] = 0$ , we say that  $X$  and  $Y$  are **uncorrelated**<sup>5</sup>.

**Properties:**

- (Linearity of expectation)  $E[f(X, Y) + g(X, Y)] = E[f(X, Y)] + E[g(X, Y)]$ .
- $Var[X + Y] = Var[X] + Var[Y] + 2Cov[X, Y]$ .
- If  $X$  and  $Y$  are independent, then  $Cov[X, Y] = 0$ .
- If  $X$  and  $Y$  are independent, then  $E[f(X)g(Y)] = E[f(X)]E[g(Y)]$ .

## 4 Multiple random variables

The notions and ideas introduced in the previous section can be generalized to more than two random variables. In particular, suppose that we have  $n$  continuous random variables,  $X_1(\omega), X_2(\omega), \dots, X_n(\omega)$ . In this section, for simplicity of presentation, we focus only on the continuous case, but the generalization to discrete random variables works similarly.

---

<sup>5</sup>However, this is not the same thing as stating that  $X$  and  $Y$  are independent! For example, if  $X \sim Uniform(-1, 1)$  and  $Y = X^2$ , then one can show that  $X$  and  $Y$  are uncorrelated, even though they are not independent.

## 4.1 Basic properties

We can define the **joint distribution function** of  $X_1, X_2, \dots, X_n$ , the **joint probability density function** of  $X_1, X_2, \dots, X_n$ , the **marginal probability density function** of  $X_1$ , and the **conditional probability density function** of  $X_1$  given  $X_2, \dots, X_n$ , as

$$\begin{aligned} F_{X_1, X_2, \dots, X_n}(x_1, x_2, \dots, x_n) &= P(X_1 \leq x_1, X_2 \leq x_2, \dots, X_n \leq x_n) \\ f_{X_1, X_2, \dots, X_n}(x_1, x_2, \dots, x_n) &= \frac{\partial^n F_{X_1, X_2, \dots, X_n}(x_1, x_2, \dots, x_n)}{\partial x_1 \dots \partial x_n} \\ f_{X_1}(X_1) &= \int_{-\infty}^{\infty} \dots \int_{-\infty}^{\infty} f_{X_1, X_2, \dots, X_n}(x_1, x_2, \dots, x_n) dx_2 \dots dx_n \\ f_{X_1|X_2, \dots, X_n}(x_1|x_2, \dots, x_n) &= \frac{f_{X_1, X_2, \dots, X_n}(x_1, x_2, \dots, x_n)}{f_{X_2, \dots, X_n}(x_1, x_2, \dots, x_n)} \end{aligned}$$

To calculate the probability of an event  $A \subseteq \mathbb{R}^n$  we have,

$$P((x_1, x_2, \dots, x_n) \in A) = \int_{(x_1, x_2, \dots, x_n) \in A} f_{X_1, X_2, \dots, X_n}(x_1, x_2, \dots, x_n) dx_1 dx_2 \dots dx_n \quad (4)$$

**Chain rule:** From the definition of conditional probabilities for multiple random variables, one can show that

$$\begin{aligned} f(x_1, x_2, \dots, x_n) &= f(x_n|x_1, x_2, \dots, x_{n-1}) f(x_1, x_2, \dots, x_{n-1}) \\ &= f(x_n|x_1, x_2, \dots, x_{n-1}) f(x_{n-1}|x_1, x_2, \dots, x_{n-2}) f(x_1, x_2, \dots, x_{n-2}) \\ &= \dots = f(x_1) \prod_{i=2}^n f(x_i|x_1, \dots, x_{i-1}). \end{aligned}$$

**Independence:** For multiple events,  $A_1, \dots, A_k$ , we say that  $A_1, \dots, A_k$  are **mutually independent** if for any subset  $S \subseteq \{1, 2, \dots, k\}$ , we have

$$P(\cap_{i \in S} A_i) = \prod_{i \in S} P(A_i).$$

Likewise, we say that random variables  $X_1, \dots, X_n$  are independent if

$$f(x_1, \dots, x_n) = f(x_1)f(x_2) \cdots f(x_n).$$

Here, the definition of mutual independence is simply the natural generalization of independence of two random variables to multiple random variables.

Independent random variables arise often in machine learning algorithms where we assume that the training examples belonging to the training set represent independent samples from some unknown probability distribution. To make the significance of independence clear, consider a “bad” training set in which we first sample a single training example  $(x^{(1)}, y^{(1)})$  from the some unknown distribution, and then add  $m - 1$  copies of the exact same training example to the training set. In this case, we have (with some abuse of notation)

$$P((x^{(1)}, y^{(1)}), \dots, (x^{(m)}, y^{(m)})) \neq \prod_{i=1}^m P(x^{(i)}, y^{(i)}).$$

Despite the fact that the training set has size  $m$ , the examples are not independent! While clearly the procedure described here is not a sensible method for building a training set for a machine learning algorithm, it turns out that in practice, non-independence of samples does come up often, and it has the effect of reducing the “effective size” of the training set.

## 4.2 Random vectors

Suppose that we have  $n$  random variables. When working with all these random variables together, we will often find it convenient to put them in a vector  $X = [X_1 \ X_2 \ \dots \ X_n]^T$ . We call the resulting vector a **random vector** (more formally, a random vector is a mapping from  $\Omega$  to  $\mathbb{R}^n$ ). It should be clear that random vectors are simply an alternative notation for dealing with  $n$  random variables, so the notions of joint PDF and CDF will apply to random vectors as well.

**Expectation:** Consider an arbitrary function from  $g : \mathbb{R}^n \rightarrow \mathbb{R}$ . The expected value of this function is defined as

$$E[g(X)] = \int_{\mathbb{R}^n} g(x_1, x_2, \dots, x_n) f_{X_1, X_2, \dots, X_n}(x_1, x_2, \dots, x_n) dx_1 dx_2 \dots dx_n, \quad (5)$$

where  $\int_{\mathbb{R}^n}$  is  $n$  consecutive integrations from  $-\infty$  to  $\infty$ . If  $g$  is a function from  $\mathbb{R}^n$  to  $\mathbb{R}^m$ , then the expected value of  $g$  is the element-wise expected values of the output vector, i.e., if  $g$  is

$$g(x) = \begin{bmatrix} g_1(x) \\ g_2(x) \\ \vdots \\ g_m(x) \end{bmatrix},$$

Then,

$$E[g(X)] = \begin{bmatrix} E[g_1(X)] \\ E[g_2(X)] \\ \vdots \\ E[g_m(X)] \end{bmatrix}.$$

**Covariance matrix:** For a given random vector  $X : \Omega \rightarrow \mathbb{R}^n$ , its covariance matrix  $\Sigma$  is the  $n \times n$  square matrix whose entries are given by  $\Sigma_{ij} = Cov[X_i, X_j]$ .

From the definition of covariance, we have

$$\begin{aligned} \Sigma &= \begin{bmatrix} Cov[X_1, X_1] & \cdots & Cov[X_1, X_n] \\ \vdots & \ddots & \vdots \\ Cov[X_n, X_1] & \cdots & Cov[X_n, X_n] \end{bmatrix} \\ &= \begin{bmatrix} E[X_1^2] - E[X_1]E[X_1] & \cdots & E[X_1X_n] - E[X_1]E[X_n] \\ \vdots & \ddots & \vdots \\ E[X_nX_1] - E[X_n]E[X_1] & \cdots & E[X_n^2] - E[X_n]E[X_n] \end{bmatrix} \\ &= \begin{bmatrix} E[X_1^2] & \cdots & E[X_1X_n] \\ \vdots & \ddots & \vdots \\ E[X_nX_1] & \cdots & E[X_n^2] \end{bmatrix} - \begin{bmatrix} E[X_1]E[X_1] & \cdots & E[X_1]E[X_n] \\ \vdots & \ddots & \vdots \\ E[X_n]E[X_1] & \cdots & E[X_n]E[X_n] \end{bmatrix} \\ &= E[XX^T] - E[X]E[X]^T = \dots = E[(X - E[X])(X - E[X])^T]. \end{aligned}$$

where the matrix expectation is defined in the obvious way.

The covariance matrix has a number of useful properties:

- $\Sigma \succeq 0$ ; that is,  $\Sigma$  is positive semidefinite.
- $\Sigma = \Sigma^T$ ; that is,  $\Sigma$  is symmetric.

## 4.3 The multivariate Gaussian distribution

One particularly important example of a probability distribution over random vectors  $X$  is called the **multivariate Gaussian** or **multivariate normal** distribution. A random vector  $X \in \mathbb{R}^n$  is said to have a multivariate normal (or Gaussian) distribution with mean  $\mu \in \mathbb{R}^n$  and covariance matrix  $\Sigma \in \mathbb{S}_{++}^n$  (where  $\mathbb{S}_{++}^n$  refers to the space of symmetric positive definite  $n \times n$  matrices)

$$f_{X_1, X_2, \dots, X_n}(x_1, x_2, \dots, x_n; \mu, \Sigma) = \frac{1}{(2\pi)^{n/2} |\Sigma|^{1/2}} \exp\left(-\frac{1}{2}(x - \mu)^T \Sigma^{-1} (x - \mu)\right).$$

We write this as  $X \sim \mathcal{N}(\mu, \Sigma)$ . Notice that in the case  $n = 1$ , this reduces the regular definition of a normal distribution with mean parameter  $\mu_1$  and variance  $\Sigma_{11}$ .

Generally speaking, Gaussian random variables are extremely useful in machine learning and statistics for two main reasons. First, they are extremely common when modeling “noise” in statistical algorithms. Quite often, noise can be considered to be the accumulation of a large number of small independent random perturbations affecting the measurement process; by the Central Limit Theorem, summations of independent random variables will tend to “look Gaussian.” Second, Gaussian random variables are convenient for many analytical manipulations, because many of the integrals involving Gaussian distributions that arise in practice have simple closed form solutions. We will encounter this later in the course.

## 5 Other resources

A good textbook on probability at the level needed for CS229 is the book, *A First Course on Probability* by Sheldon Ross.

# Gaussian processes

Chuong B. Do (updated by Honglak Lee)

July 17, 2019

Many of the classical machine learning algorithms that we talked about during the first half of this course fit the following pattern: given a training set of i.i.d. examples sampled from some unknown distribution,

1. solve a convex optimization problem in order to identify the single “best fit” model for the data, and
2. use this estimated model to make “best guess” predictions for future test input points.

In these notes, we will talk about a different flavor of learning algorithms, known as **Bayesian methods**. Unlike classical learning algorithm, Bayesian algorithms do not attempt to identify “best-fit” models of the data (or similarly, make “best guess” predictions for new test inputs). Instead, they compute a posterior distribution over models (or similarly, compute posterior predictive distributions for new test inputs). These distributions provide a useful way to quantify our uncertainty in model estimates, and to exploit our knowledge of this uncertainty in order to make more robust predictions on new test points.

We focus on **regression** problems, where the goal is to learn a mapping from some input space  $\mathcal{X} = \mathbf{R}^d$  of  $d$ -dimensional vectors to an output space  $\mathcal{Y} = \mathbf{R}$  of real-valued targets. In particular, we will talk about a kernel-based fully Bayesian regression algorithm, known as Gaussian process regression. The material covered in these notes draws heavily on many different topics that we discussed previously in class (namely, the probabilistic interpretation of linear regression<sup>1</sup>, Bayesian methods<sup>2</sup>, kernels<sup>3</sup>, and properties of multivariate Gaussians<sup>4</sup>).

The organization of these notes is as follows. In Section 1, we provide a brief review of multivariate Gaussian distributions and their properties. In Section 2, we briefly review Bayesian methods in the context of probabilistic linear regression. The central ideas underlying Gaussian processes are presented in Section 3, and we derive the full Gaussian process regression model in Section 4.

---

<sup>1</sup>See course lecture notes on “Supervised Learning, Discriminative Algorithms.”

<sup>2</sup>See course lecture notes on “Regularization and Model Selection.”

<sup>3</sup>See course lecture notes on “Support Vector Machines.”

<sup>4</sup>See course lecture notes on “Factor Analysis.”

# 1 Multivariate Gaussians

A vector-valued random variable  $x \in \mathbf{R}^d$  is said to have a **multivariate normal (or Gaussian) distribution** with mean  $\mu \in \mathbf{R}^d$  and covariance matrix  $\Sigma \in \mathbf{S}_{++}^d$  if

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

We write this as  $x \sim \mathcal{N}(\mu, \Sigma)$ . Here, recall from the section notes on linear algebra that  $\mathbf{S}_{++}^d$  refers to the space of symmetric positive definite  $n \times d$  matrices.<sup>5</sup>

Generally speaking, Gaussian random variables are extremely useful in machine learning and statistics for two main reasons. First, they are extremely common when modeling “noise” in statistical algorithms. Quite often, noise can be considered to be the accumulation of a large number of small independent random perturbations affecting the measurement process; by the Central Limit Theorem, summations of independent random variables will tend to “look Gaussian.” Second, Gaussian random variables are convenient for many analytical manipulations, because many of the integrals involving Gaussian distributions that arise in practice have simple closed form solutions. In the remainder of this section, we will review a number of useful properties of multivariate Gaussians.

Consider a random vector  $x \in \mathbf{R}^d$  with  $x \sim \mathcal{N}(\mu, \Sigma)$ . Suppose also that the variables in  $x$  have been partitioned into two sets  $x_A = [x_1 \cdots x_r]^T \in \mathbf{R}^r$  and  $x_B = [x_{r+1} \cdots x_d]^T \in \mathbf{R}^{d-r}$  (and similarly for  $\mu$  and  $\Sigma$ ), such that

$$x = \begin{bmatrix} x_A \\ x_B \end{bmatrix} \quad \mu = \begin{bmatrix} \mu_A \\ \mu_B \end{bmatrix} \quad \Sigma = \begin{bmatrix} \Sigma_{AA} & \Sigma_{AB} \\ \Sigma_{BA} & \Sigma_{BB} \end{bmatrix}.$$

Here,  $\Sigma_{AB} = \Sigma_{BA}^T$  since  $\Sigma = E[(x - \mu)(x - \mu)^T] = \Sigma^T$ . The following properties hold:

1. **Normalization.** The density function normalizes, i.e.,

$$\int_x p(x; \mu, \Sigma) dx = 1.$$

This property, though seemingly trivial at first glance, turns out to be immensely useful for evaluating all sorts of integrals, even ones which appear to have no relation to probability distributions at all (see Appendix A.1)!

2. **Marginalization.** The marginal densities,

$$\begin{aligned} p(x_A) &= \int_{x_B} p(x_A, x_B; \mu, \Sigma) dx_B \\ p(x_B) &= \int_{x_A} p(x_A, x_B; \mu, \Sigma) dx_A \end{aligned}$$

---

<sup>5</sup>There are actually cases in which we would want to deal with multivariate Gaussian distributions where  $\Sigma$  is positive semidefinite but not positive definite (i.e.,  $\Sigma$  is not full rank). In such cases,  $\Sigma^{-1}$  does not exist, so the definition of the Gaussian density given in (1) does not apply. For instance, see the course lecture notes on “Factor Analysis.”

are Gaussian:

$$\begin{aligned} x_A &\sim \mathcal{N}(\mu_A, \Sigma_{AA}) \\ x_B &\sim \mathcal{N}(\mu_B, \Sigma_{BB}). \end{aligned}$$

**3. Conditioning.** The conditional densities

$$\begin{aligned} p(x_A | x_B) &= \frac{p(x_A, x_B; \mu, \Sigma)}{\int_{x_A} p(x_A, x_B; \mu, \Sigma) dx_A} \\ p(x_B | x_A) &= \frac{p(x_A, x_B; \mu, \Sigma)}{\int_{x_B} p(x_A, x_B; \mu, \Sigma) dx_B} \end{aligned}$$

are also Gaussian:

$$\begin{aligned} x_A | x_B &\sim \mathcal{N}(\mu_A + \Sigma_{AB}\Sigma_{BB}^{-1}(x_B - \mu_B), \Sigma_{AA} - \Sigma_{AB}\Sigma_{BB}^{-1}\Sigma_{BA}) \\ x_B | x_A &\sim \mathcal{N}(\mu_B + \Sigma_{BA}\Sigma_{AA}^{-1}(x_A - \mu_A), \Sigma_{BB} - \Sigma_{BA}\Sigma_{AA}^{-1}\Sigma_{AB}). \end{aligned}$$

A proof of this property is given in Appendix A.2. (See also Appendix A.3 for an easier version of the derivation.)

**4. Summation.** The sum of independent Gaussian random variables (with the same dimensionality),  $y \sim \mathcal{N}(\mu, \Sigma)$  and  $z \sim \mathcal{N}(\mu', \Sigma')$ , is also Gaussian:

$$y + z \sim \mathcal{N}(\mu + \mu', \Sigma + \Sigma').$$

## 2 Bayesian linear regression

Let  $S = \{(x^{(i)}, y^{(i)})\}_{i=1}^n$  be a training set of i.i.d. examples from some unknown distribution. The standard probabilistic interpretation of linear regression states that

$$y^{(i)} = \theta^T x^{(i)} + \varepsilon^{(i)}, \quad i = 1, \dots, n$$

where the  $\varepsilon^{(i)}$  are i.i.d. “noise” variables with independent  $\mathcal{N}(0, \sigma^2)$  distributions. It follows that  $y^{(i)} - \theta^T x^{(i)} \sim \mathcal{N}(0, \sigma^2)$ , or equivalently,

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

For notational convenience, we define

$$X = \begin{bmatrix} \cdots & (x^{(1)})^T & \cdots \\ \cdots & (x^{(2)})^T & \cdots \\ \vdots & & \\ \cdots & (x^{(n)})^T & \cdots \end{bmatrix} \in \mathbf{R}^{n \times d} \quad \vec{y} = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(n)} \end{bmatrix} \in \mathbf{R}^n \quad \vec{\varepsilon} = \begin{bmatrix} \varepsilon^{(1)} \\ \varepsilon^{(2)} \\ \vdots \\ \varepsilon^{(n)} \end{bmatrix} \in \mathbf{R}^n.$$

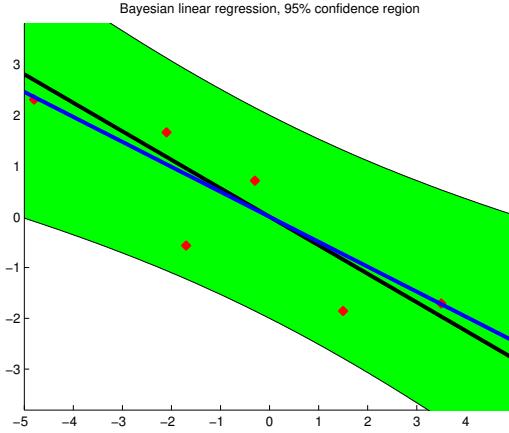


Figure 1: Bayesian linear regression for a one-dimensional linear regression problem,  $y^{(i)} = \theta x^{(i)} + \epsilon^{(i)}$ , with  $\epsilon^{(i)} \sim \mathcal{N}(0, 1)$  i.i.d. noise. The green region denotes the 95% confidence region for predictions of the model. Note that the (vertical) width of the green region is largest at the ends but narrowest in the middle. This region reflects the uncertainty in the estimates for the parameter  $\theta$ . In contrast, a classical linear regression model would display a confidence region of constant width, reflecting only the  $\mathcal{N}(0, \sigma^2)$  noise in the outputs.

In Bayesian linear regression, we assume that a **prior distribution** over parameters is also given; a typical choice, for instance, is  $\theta \sim \mathcal{N}(0, \tau^2 I)$ . Using Bayes's rule, we obtain the **parameter posterior**,

$$p(\theta | S) = \frac{p(\theta)p(S | \theta)}{\int_{\theta'} p(\theta')p(S | \theta')d\theta'} = \frac{p(\theta) \prod_{i=1}^n p(y^{(i)} | x^{(i)}, \theta)}{\int_{\theta'} p(\theta') \prod_{i=1}^n p(y^{(i)} | x^{(i)}, \theta')d\theta'}. \quad (2)$$

Assuming the same noise model on testing points as on our training points, the “output” of Bayesian linear regression on a new test point  $x_*$  is not just a single guess “ $y_*$ ”, but rather an entire probability distribution over possible outputs, known as the **posterior predictive distribution**:

$$p(y_* | x_*, S) = \int_{\theta} p(y_* | x_*, \theta)p(\theta | S)d\theta. \quad (3)$$

For many types of models, the integrals in (2) and (3) are difficult to compute, and hence, we often resort to approximations, such as MAP estimation (see course lecture notes on “Regularization and Model Selection”).

In the case of Bayesian linear regression, however, the integrals actually are tractable! In particular, for Bayesian linear regression, one can show (after much work!) that

$$\begin{aligned} \theta | S &\sim \mathcal{N}\left(\frac{1}{\sigma^2} A^{-1} X^T \vec{y}, A^{-1}\right) \\ y_* | x_*, S &\sim \mathcal{N}\left(\frac{1}{\sigma^2} x_*^T A^{-1} X^T \vec{y}, x_*^T A^{-1} x_* + \sigma^2\right) \end{aligned}$$

where  $A = \frac{1}{\sigma^2} X^T X + \frac{1}{\tau^2} I$ . The derivation of these formulas is somewhat involved.<sup>6</sup> Nonetheless, from these equations, we get at least a flavor of what Bayesian methods are all about: the posterior distribution over the test output  $y_*$  for a test input  $x_*$  is a Gaussian distribution—this distribution reflects the uncertainty in our predictions  $y_* = \theta^T x_* + \varepsilon_*$  arising from both the randomness in  $\varepsilon_*$  and the uncertainty in our choice of parameters  $\theta$ . In contrast, classical probabilistic linear regression models estimate parameters  $\theta$  directly from the training data but provide no estimate of how reliable these learned parameters may be (see Figure 1).

### 3 Gaussian processes

As described in Section 1, multivariate Gaussian distributions are useful for modeling finite collections of real-valued variables because of their nice analytical properties. **Gaussian processes** are the extension of multivariate Gaussians to infinite-sized collections of real-valued variables. In particular, this extension will allow us to think of Gaussian processes as distributions not just over random vectors but in fact distributions over **random functions**.<sup>7</sup>

#### 3.1 Probability distributions over functions with finite domains

To understand how one might parameterize probability distributions over functions, consider the following simple example. Let  $\mathcal{X} = \{x_1, \dots, x_n\}$  be any finite set of elements. Now, consider the set  $\mathcal{H}$  of all possible functions mapping from  $\mathcal{X}$  to  $\mathbf{R}$ . For instance, one example of a function  $f_0(\cdot) \in \mathcal{H}$  is given by

$$f_0(x_1) = 5, \quad f_0(x_2) = 2.3, \quad f_0(x_3) = -7, \quad \dots, \quad f_0(x_{n-1}) = -\pi, \quad f_0(x_n) = 8.$$

Since the domain of any  $f(\cdot) \in \mathcal{H}$  has only  $n$  elements, we can always represent  $f(\cdot)$  compactly as an  $n$ -dimensional vector,  $\vec{f} = [f(x_1) \ f(x_2) \ \dots \ f(x_n)]^T$ . In order to specify a probability distribution over functions  $f(\cdot) \in \mathcal{H}$ , we must associate some “probability density” with each function in  $\mathcal{H}$ . One natural way to do this is to exploit the one-to-one correspondence between functions  $f(\cdot) \in \mathcal{H}$  and their vector representations,  $\vec{f}$ . In particular, if we specify that  $\vec{f} \sim \mathcal{N}(\vec{\mu}, \sigma^2 I)$ , then this in turn implies a probability distribution over functions  $f(\cdot)$ , whose probability density function is given by

$$p(\vec{f}) = \prod_{i=1}^n \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{1}{2\sigma^2}(f(x_i) - \mu_i)^2\right).$$

---

<sup>6</sup>For the complete derivation, see, for instance, [1]. Alternatively, read the Appendices, which gives a number of arguments based on the “completion-of-squares” trick, and derive this formula yourself!

<sup>7</sup>Let  $\mathcal{H}$  be a class of functions mapping from  $\mathcal{X} \rightarrow \mathcal{Y}$ . A random function  $f(\cdot)$  from  $\mathcal{H}$  is a function which is randomly drawn from  $\mathcal{H}$ , according to some probability distribution over  $\mathcal{H}$ . One potential source of confusion is that you may be tempted to think of random functions as functions whose outputs are in some way stochastic; this is not the case. Instead, a random function  $f(\cdot)$ , once selected from  $\mathcal{H}$  probabilistically, implies a deterministic mapping from inputs in  $\mathcal{X}$  to outputs in  $\mathcal{Y}$ .

In the example above, we showed that probability distributions over functions with finite domains can be represented using a finite-dimensional multivariate Gaussian distribution over function outputs  $f(x_1), \dots, f(x_n)$  at a finite number of input points  $x_1, \dots, x_n$ . How can we specify probability distributions over functions when the domain size may be infinite? For this, we turn to a fancier type of probability distribution known as a Gaussian process.

## 3.2 Probability distributions over functions with infinite domains

A stochastic process is a collection of random variables,  $\{f(x) : x \in \mathcal{X}\}$ , indexed by elements from some set  $\mathcal{X}$ , known as the index set.<sup>8</sup> A **Gaussian process** is a stochastic process such that any finite subcollection of random variables has a multivariate Gaussian distribution.

In particular, a collection of random variables  $\{f(x) : x \in \mathcal{X}\}$  is said to be drawn from a Gaussian process with **mean function**  $m(\cdot)$  and **covariance function**  $k(\cdot, \cdot)$  if for any finite set of elements  $x_1, \dots, x_n \in \mathcal{X}$ , the associated finite set of random variables  $f(x_1), \dots, f(x_n)$  have distribution,

$$\begin{bmatrix} f(x_1) \\ \vdots \\ f(x_n) \end{bmatrix} \sim \mathcal{N} \left( \begin{bmatrix} m(x_1) \\ \vdots \\ m(x_n) \end{bmatrix}, \begin{bmatrix} k(x_1, x_1) & \cdots & k(x_1, x_n) \\ \vdots & \ddots & \vdots \\ k(x_n, x_1) & \cdots & k(x_n, x_n) \end{bmatrix} \right).$$

We denote this using the notation,

$$f(\cdot) \sim \mathcal{GP}(m(\cdot), k(\cdot, \cdot)).$$

Observe that the mean function and covariance function are aptly named since the above properties imply that

$$\begin{aligned} m(x) &= \mathbb{E}[f(x)] \\ k(x, x') &= \mathbb{E}[(f(x) - m(x))(f(x') - m(x'))]. \end{aligned}$$

for any  $x, x' \in \mathcal{X}$ .

Intuitively, one can think of a function  $f(\cdot)$  drawn from a Gaussian process prior as an extremely high-dimensional vector drawn from an extremely high-dimensional multivariate Gaussian. Here, each dimension of the Gaussian corresponds to an element  $x$  from the index set  $\mathcal{X}$ , and the corresponding component of the random vector represents the value of  $f(x)$ . Using the marginalization property for multivariate Gaussians, we can obtain the marginal multivariate Gaussian density corresponding to any finite subcollection of variables.

What sort of functions  $m(\cdot)$  and  $k(\cdot, \cdot)$  give rise to valid Gaussian processes? In general, any real-valued function  $m(\cdot)$  is acceptable, but for  $k(\cdot, \cdot)$ , it must be the case that for any

---

<sup>8</sup>Often, when  $\mathcal{X} = \mathbf{R}$ , one can interpret the indices  $x \in \mathcal{X}$  as representing times, and hence the variables  $f(x)$  represent the temporal evolution of some random quantity over time. In the models that are used for Gaussian process regression, however, the index set is taken to be the input space of our regression problem.

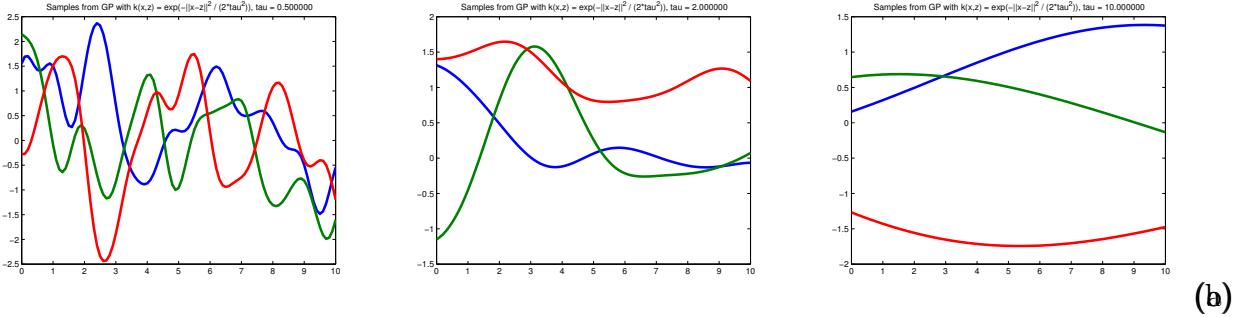


Figure 2: Samples from a zero-mean Gaussian process prior with  $k_{SE}(\cdot, \cdot)$  covariance function, using (a)  $\tau = 0.5$ , (b)  $\tau = 2$ , and (c)  $\tau = 10$ . Note that as the bandwidth parameter  $\tau$  increases, then points which are farther away will have higher correlations than before, and hence the sampled functions tend to be smoother overall.

set of elements  $x_1, \dots, x_n \in \mathcal{X}$ , the resulting matrix

$$K = \begin{bmatrix} k(x_1, x_1) & \cdots & k(x_1, x_n) \\ \vdots & \ddots & \vdots \\ k(x_n, x_1) & \cdots & k(x_n, x_n) \end{bmatrix}$$

is a valid covariance matrix corresponding to some multivariate Gaussian distribution. A standard result in probability theory states that this is true provided that  $K$  is positive semidefinite. Sound familiar?

The positive semidefiniteness requirement for covariance matrices computed based on arbitrary input points is, in fact, identical to Mercer’s condition for kernels! A function  $k(\cdot, \cdot)$  is a valid kernel provided the resulting kernel matrix  $K$  defined as above is always positive semidefinite for any set of input points  $x_1, \dots, x_n \in \mathcal{X}$ . Gaussian processes, therefore, are kernel-based probability distributions in the sense that any valid kernel function can be used as a covariance function!

### 3.3 The squared exponential kernel

In order to get an intuition for how Gaussian processes work, consider a simple zero-mean Gaussian process,

$$f(\cdot) \sim \mathcal{GP}(0, k(\cdot, \cdot)).$$

defined for functions  $h : \mathcal{X} \rightarrow \mathbf{R}$  where we take  $\mathcal{X} = \mathbf{R}$ . Here, we choose the kernel function  $k(\cdot, \cdot)$  to be the **squared exponential**<sup>9</sup> kernel function, defined as

$$k_{SE}(x, x') = \exp\left(-\frac{1}{2\tau^2} \|x - x'\|^2\right)$$

---

<sup>9</sup>In the context of SVMs, we called this the Gaussian kernel; to avoid confusion with “Gaussian” processes, we refer to this kernel here as the squared exponential kernel, even though the two are formally identical.

for some  $\tau > 0$ . What do random functions sampled from this Gaussian process look like?

In our example, since we use a zero-mean Gaussian process, we would expect that for the function values from our Gaussian process will tend to be distributed around zero. Furthermore, for any pair of elements  $x, x' \in \mathcal{X}$ .

- $f(x)$  and  $f(x')$  will tend to have high covariance when  $x$  and  $x'$  are “nearby” in the input space (i.e.,  $\|x - x'\| = |x - x'| \approx 0$ , so  $\exp(-\frac{1}{2\tau^2}\|x - x'\|^2) \approx 1$ ).
- $f(x)$  and  $f(x')$  will tend to have low covariance when  $x$  and  $x'$  are “far apart” (i.e.,  $\|x - x'\| \gg 0$ , so  $\exp(-\frac{1}{2\tau^2}\|x - x'\|^2) \approx 0$ ).

More simply stated, functions drawn from a zero-mean Gaussian process prior with the squared exponential kernel will tend to be “locally smooth” with high probability; i.e., nearby function values are highly correlated, and the correlation drops off as a function of distance in the input space (see Figure 2).

## 4 Gaussian process regression

As discussed in the last section, Gaussian processes provide a method for modelling probability distributions over functions. Here, we discuss how probability distributions over functions can be used in the framework of Bayesian regression.

### 4.1 The Gaussian process regression model

Let  $S = \{(x^{(i)}, y^{(i)})\}_{i=1}^n$  be a training set of i.i.d. examples from some unknown distribution. In the Gaussian process regression model,

$$y^{(i)} = f(x^{(i)}) + \varepsilon^{(i)}, \quad i = 1, \dots, n$$

where the  $\varepsilon^{(i)}$  are i.i.d. “noise” variables with independent  $\mathcal{N}(0, \sigma^2)$  distributions. Like in Bayesian linear regression, we also assume a **prior distribution** over functions  $f(\cdot)$ ; in particular, we assume a zero-mean Gaussian process prior,

$$f(\cdot) \sim \mathcal{GP}(0, k(\cdot, \cdot))$$

for some valid covariance function  $k(\cdot, \cdot)$ .

Now, let  $T = \{(x_*^{(i)}, y_*^{(i)})\}_{i=1}^{n_*}$  be a set of i.i.d. testing points drawn from the same unknown

distribution as  $S$ .<sup>10</sup> For notational convenience, we define

$$X = \begin{bmatrix} \cdots & (x^{(1)})^T & \cdots \\ \cdots & (x^{(2)})^T & \cdots \\ \vdots & & \\ \cdots & (x^{(n)})^T & \cdots \end{bmatrix} \in \mathbf{R}^{n \times d} \quad \vec{f} = \begin{bmatrix} f(x^{(1)}) \\ f(x^{(2)}) \\ \vdots \\ f(x^{(n)}) \end{bmatrix}, \quad \vec{\varepsilon} = \begin{bmatrix} \varepsilon^{(1)} \\ \varepsilon^{(2)} \\ \vdots \\ \varepsilon^{(n)} \end{bmatrix}, \quad \vec{y} = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(n)} \end{bmatrix} \in \mathbf{R}^n,$$

$$X_* = \begin{bmatrix} \cdots & (x_*^{(1)})^T & \cdots \\ \cdots & (x_*^{(2)})^T & \cdots \\ \vdots & & \\ \cdots & (x_*^{(n*)})^T & \cdots \end{bmatrix} \in \mathbf{R}^{n_* \times d} \quad \vec{f}_* = \begin{bmatrix} f(x_*^{(1)}) \\ f(x_*^{(2)}) \\ \vdots \\ f(x_*^{(n*)}) \end{bmatrix}, \quad \vec{\varepsilon}_* = \begin{bmatrix} \varepsilon_*^{(1)} \\ \varepsilon_*^{(2)} \\ \vdots \\ \varepsilon_*^{(n*)} \end{bmatrix}, \quad \vec{y}_* = \begin{bmatrix} y_*^{(1)} \\ y_*^{(2)} \\ \vdots \\ y_*^{(n*)} \end{bmatrix} \in \mathbf{R}^{n_*}.$$

Given the training data  $S$ , the prior  $p(h)$ , and the testing inputs  $X_*$ , how can we compute the posterior predictive distribution over the testing outputs  $\vec{y}_*$ ? For Bayesian linear regression in Section 2, we used Bayes's rule in order to compute the parameter posterior, which we then used to compute posterior predictive distribution  $p(y_* | x_*, S)$  for a new test point  $x_*$ . For Gaussian process regression, however, it turns out that an even simpler solution exists!

## 4.2 Prediction

Recall that for any function  $f(\cdot)$  drawn from our zero-mean Gaussian process prior with covariance function  $k(\cdot, \cdot)$ , the marginal distribution over any set of input points belonging to  $\mathcal{X}$  must have a joint multivariate Gaussian distribution. In particular, this must hold for the training and test points, so we have

$$\left[ \begin{array}{c} \vec{f} \\ \vec{f}_* \end{array} \right] \middle| X, X_* \sim \mathcal{N}\left(\vec{0}, \begin{bmatrix} K(X, X) & K(X, X_*) \\ K(X_*, X) & K(X_*, X_*) \end{bmatrix}\right),$$

where

$$\begin{aligned} \vec{f} &\in \mathbf{R}^n \text{ such that } \vec{f} = [f(x^{(1)}) \ \cdots \ f(x^{(n)})]^T \\ \vec{f}_* &\in \mathbf{R}^{n_*} \text{ such that } \vec{f}_* = [f(x_*^{(1)}) \ \cdots \ f(x_*^{(n)})]^T \\ K(X, X) &\in \mathbf{R}^{n \times n} \text{ such that } (K(X, X))_{ij} = k(x^{(i)}, x^{(j)}) \\ K(X, X_*) &\in \mathbf{R}^{n \times n_*} \text{ such that } (K(X, X_*))_{ij} = k(x^{(i)}, x_*^{(j)}) \\ K(X_*, X) &\in \mathbf{R}^{n_* \times n} \text{ such that } (K(X_*, X))_{ij} = k(x_*^{(i)}, x^{(j)}) \\ K(X_*, X_*) &\in \mathbf{R}^{n_* \times n_*} \text{ such that } (K(X_*, X_*))_{ij} = k(x_*^{(i)}, x_*^{(j)}). \end{aligned}$$

From our i.i.d. noise assumption, we have that

$$\left[ \begin{array}{c} \vec{\varepsilon} \\ \vec{\varepsilon}_* \end{array} \right] \sim \mathcal{N}\left(\vec{0}, \begin{bmatrix} \sigma^2 I & \vec{0} \\ \vec{0}^T & \sigma^2 I \end{bmatrix}\right).$$

---

<sup>10</sup>We assume also that  $T$  and  $S$  are mutually independent.

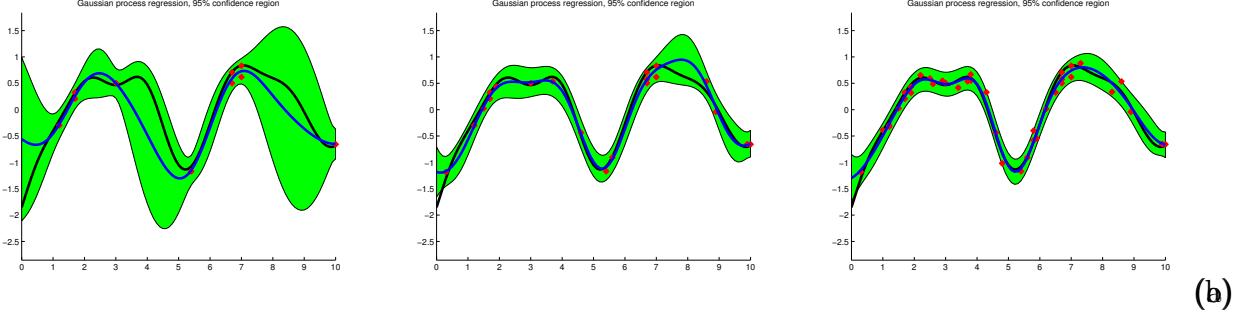


Figure 3: Gaussian process regression using a zero-mean Gaussian process prior with  $k_{SE}(\cdot, \cdot)$  covariance function (where  $\tau = 0.1$ ), with noise level  $\sigma = 1$ , and (a)  $m = 10$ , (b)  $m = 20$ , and (c)  $m = 40$  training examples. The blue line denotes the mean of the posterior predictive distribution, and the green shaded region denotes the 95% confidence region based on the model's variance estimates. As the number of training examples increases, the size of the confidence region shrinks to reflect the diminishing uncertainty in the model estimates. Note also that in panel (a), the 95% confidence region shrinks near training points but is much larger far away from training points, as one would expect.

The sums of independent Gaussian random variables is also Gaussian, so

$$\begin{bmatrix} \vec{y} \\ \vec{y}_* \end{bmatrix} \Big| X, X_* = \begin{bmatrix} \vec{f} \\ \vec{f}_* \end{bmatrix} + \begin{bmatrix} \vec{\varepsilon} \\ \vec{\varepsilon}_* \end{bmatrix} \sim \mathcal{N}\left(\vec{0}, \begin{bmatrix} K(X, X) + \sigma^2 I & K(X, X_*) \\ K(X_*, X) & K(X_*, X_*) + \sigma^2 I \end{bmatrix}\right).$$

Now, using the rules for conditioning Gaussians, it follows that

$$\vec{y}_* \mid \vec{y}, X, X_* \sim \mathcal{N}(\mu^*, \Sigma^*)$$

where

$$\begin{aligned} \mu^* &= K(X_*, X) (K(X, X) + \sigma^2 I)^{-1} \vec{y} \\ \Sigma^* &= K(X_*, X_*) + \sigma^2 I - K(X_*, X) (K(X, X) + \sigma^2 I)^{-1} K(X, X_*) \end{aligned}$$

And that's it! Remarkably, performing prediction in a Gaussian process regression model is very simple, despite the fact that Gaussian processes in themselves are fairly complicated!<sup>11</sup>

## 5 Summary

We close our discussion of our Gaussian processes by pointing out some reasons why Gaussian processes are an attractive model for use in regression problems and in some cases may be preferable to alternative models (such as linear and locally-weighted linear regression):

---

<sup>11</sup>Interestingly, it turns out that Bayesian linear regression, when “kernelized” in the proper way, turns out to be exactly equivalent to Gaussian process regression! But the derivation of the posterior predictive distribution is far more complicated for Bayesian linear regression, and the effort needed to kernelize the algorithm is even greater. The Gaussian process perspective is certainly much easier!

1. As Bayesian methods, Gaussian process models allow one to quantify uncertainty in predictions resulting not just from intrinsic noise in the problem but also the errors in the parameter estimation procedure. Furthermore, many methods for model selection and hyperparameter selection in Bayesian methods are immediately applicable to Gaussian processes (though we did not address any of these advanced topics here).
2. Like locally-weighted linear regression, Gaussian process regression is non-parametric and hence can model essentially arbitrary functions of the input points.
3. Gaussian process regression models provide a natural way to introduce kernels into a regression modeling framework. By careful choice of kernels, Gaussian process regression models can sometimes take advantage of structure in the data (though, we also did not examine this issue here).
4. Gaussian process regression models, though perhaps somewhat tricky to understand conceptually, nonetheless lead to simple and straightforward linear algebra implementations.

## References

- [1] Carl E. Rasmussen and Christopher K. I. Williams. *Gaussian Processes for Machine Learning*. MIT Press, 2006. Online: <http://www.gaussianprocess.org/gpml/>

## Appendix A.1

In this example, we show how the normalization property for multivariate Gaussians can be used to compute rather intimidating multidimensional integrals without performing any real calculus! Suppose you wanted to compute the following multidimensional integral,

$$I(A, b, c) = \int_x \exp\left(-\frac{1}{2}x^T Ax - x^T b - c\right) dx,$$

for some  $A \in \mathbf{S}_{++}^n$ ,  $b \in \mathbf{R}^n$ , and  $c \in \mathbf{R}$ . Although one could conceivably perform the multidimensional integration directly (good luck!), a much simpler line of reasoning is based on a mathematical trick known as “completion-of-squares.” In particular,

$$\begin{aligned} I(A, b, c) &= \exp(-c) \cdot \int_x \exp\left(-\frac{1}{2}x^T Ax - x^T AA^{-1}b\right) dx \\ &= \exp(-c) \cdot \int_x \exp\left(-\frac{1}{2}(x - A^{-1}b)^T A(x - A^{-1}b) - b^T A^{-1}b\right) dx \\ &= \exp(-c - b^T A^{-1}b) \cdot \int_x \exp\left(-\frac{1}{2}(x - A^{-1}b)^T A(x - A^{-1}b)\right) dx. \end{aligned}$$

Defining  $\mu = A^{-1}b$  and  $\Sigma = A^{-1}$ , it follows that  $I(A, b, c)$  is equal to

$$\frac{(2\pi)^{n/2}|\Sigma|^{1/2}}{\exp(c + b^T A^{-1}b)} \cdot \left[ \frac{1}{(2\pi)^{n/2}|\Sigma|^{1/2}} \int_x \exp\left(-\frac{1}{2}(x - \mu)^T \Sigma^{-1}(x - \mu)\right) dx \right].$$

However, the term in brackets is identical in form to the integral of a multivariate Gaussian! Since we know that a Gaussian density normalizes, it follows that the term in brackets is equal to 1. Therefore,

$$I(A, b, c) = \frac{(2\pi)^{n/2}|A^{-1}|^{1/2}}{\exp(c + b^T A^{-1}b)}.$$

## Appendix A.2

We derive the form of the distribution of  $x_A$  given  $x_B$ ; the other result follows immediately by symmetry. Note that

$$\begin{aligned} p(x_A | x_B) &= \frac{1}{\int_{x_A} p(x_A, x_B; \mu, \Sigma) dx_A} \cdot \left[ \frac{1}{(2\pi)^{n/2}|\Sigma|^{1/2}} \exp\left(-\frac{1}{2}(x - \mu)^T \Sigma^{-1}(x - \mu)\right) \right] \\ &= \frac{1}{Z_1} \exp\left\{-\frac{1}{2}\left(\begin{bmatrix} x_A \\ x_B \end{bmatrix} - \begin{bmatrix} \mu_A \\ \mu_B \end{bmatrix}\right)^T \begin{bmatrix} V_{AA} & V_{AB} \\ V_{BA} & V_{BB} \end{bmatrix} \left(\begin{bmatrix} x_A \\ x_B \end{bmatrix} - \begin{bmatrix} \mu_A \\ \mu_B \end{bmatrix}\right)\right\} \end{aligned}$$

where  $Z_1$  is a proportionality constant which does not depend on  $x_A$ , and

$$\Sigma^{-1} = V = \begin{bmatrix} V_{AA} & V_{AB} \\ V_{BA} & V_{BB} \end{bmatrix}.$$

To simplify this expression, observe that

$$\begin{aligned} & \left( \begin{bmatrix} x_A \\ x_B \end{bmatrix} - \begin{bmatrix} \mu_A \\ \mu_B \end{bmatrix} \right)^T \begin{bmatrix} V_{AA} & V_{AB} \\ V_{BA} & V_{BB} \end{bmatrix} \left( \begin{bmatrix} x_A \\ x_B \end{bmatrix} - \begin{bmatrix} \mu_A \\ \mu_B \end{bmatrix} \right) \\ &= (x_A - \mu_A)^T V_{AA} (x_A - \mu_A) + (x_A - \mu_A)^T V_{AB} (x_B - \mu_B) \\ &\quad + (x_B - \mu_B)^T V_{BA} (x_A - \mu_A) + (x_B - \mu_B)^T V_{BB} (x_B - \mu_B). \end{aligned}$$

Retaining only terms dependent on  $x_A$  (and using the fact that  $V_{AB} = V_{BA}^T$ ), we have

$$p(x_A | x_B) = \frac{1}{Z_2} \exp \left( -\frac{1}{2} [x_A^T V_{AA} x_A - 2x_A^T V_{AA} \mu_A + 2x_A^T V_{AB} (x_B - \mu_B)] \right)$$

where  $Z_2$  is a new proportionality constant which again does not depend on  $x_A$ . Finally, using the ‘‘completion-of-squares’’ argument (see Appendix A.1), we have

$$p(x_A | x_B) = \frac{1}{Z_3} \exp \left( -\frac{1}{2} (x_A - \mu')^T V_{AA} (x_A - \mu') \right)$$

where  $Z_3$  is again a new proportionality constant not depending on  $x_A$ , and where  $\mu' = \mu_A - V_{AA}^{-1} V_{AB} (x_B - \mu_B)$ . This last statement shows that the distribution of  $x_A$ , conditioned on  $x_B$ , again has the form of a multivariate Gaussian. In fact, from the normalization property, it follows immediately that

$$x_A | x_B \sim \mathcal{N}(\mu_A - V_{AA}^{-1} V_{AB} (x_B - \mu_B), V_{AA}^{-1}).$$

To complete the proof, we simply note that

$$\begin{bmatrix} V_{AA} & V_{AB} \\ V_{BA} & V_{BB} \end{bmatrix} = \begin{bmatrix} (\Sigma_{AA} - \Sigma_{AB}\Sigma_{BB}^{-1}\Sigma_{BA})^{-1} & -(\Sigma_{AA} - \Sigma_{AB}\Sigma_{BB}^{-1}\Sigma_{BA})^{-1}\Sigma_{AB}\Sigma_{BB}^{-1} \\ -\Sigma_{BB}^{-1}\Sigma_{BA}(\Sigma_{AA} - \Sigma_{AB}\Sigma_{BB}^{-1}\Sigma_{BA})^{-1} & (\Sigma_{BB} - \Sigma_{BA}\Sigma_{AA}^{-1}\Sigma_{AB})^{-1} \end{bmatrix}$$

follows from standard formulas for the inverse of a partitioned matrix. Substituting the relevant blocks into the previous expression gives the desired result.  $\square$

## Appendix A.3

In this section, we present an alternative (and easier) derivation of the conditional distribution of multivariate Gaussian distribution. Note that, as in Appendix A.2, we can write  $p(x_A | x_B)$  as following:

$$p(x_A | x_B) = \frac{1}{\int_{x_A} p(x_A, x_B; \mu, \Sigma) dx_A} \cdot \left[ \frac{1}{(2\pi)^{n/2} |\Sigma|^{1/2}} \exp \left( -\frac{1}{2} (x - \mu)^T \Sigma^{-1} (x - \mu) \right) \right] \quad (4)$$

$$= \frac{1}{Z_1} \exp \left\{ -\frac{1}{2} \left( \begin{bmatrix} x_A - \mu_A \\ x_B - \mu_B \end{bmatrix} \right)^T \begin{bmatrix} V_{AA} & V_{AB} \\ V_{BA} & V_{BB} \end{bmatrix} \begin{bmatrix} x_A - \mu_A \\ x_B - \mu_B \end{bmatrix} \right\} \quad (5)$$

where  $Z_1$  is a proportionality constant which does not depend on  $x_A$ .

This derivation uses an additional assumption that the conditional distribution is a multivariate Gaussian distribution; in other words, we assume that  $p(x_A | x_B) \sim \mathcal{N}(\mu^*, \Sigma^*)$  for some  $\mu^*, \Sigma^*$ . (Alternatively, you can think about this derivation as another way of finding “completion-of-squares”.)

The key intuition in this derivation is that  $p(x_A | x_B)$  will be maximized when  $x_A = \mu^* \triangleq x_A^*$ . To maximize  $p(x_A | x_B)$ , we compute the gradient of  $\log p(x_A | x_B)$  w.r.t.  $x_A$  and set it to zero. Using Equation (5), we have

$$\nabla_{x_A} \log p(x_A | x_B)|_{x_A=x_A^*} \quad (6)$$

$$= -V_{AA}(x_A^* - \mu_A) - V_{AB}(x_B - \mu_B) \quad (7)$$

$$= 0. \quad (8)$$

This implies that

$$\mu^* = x_A^* = \mu_A - V_{AA}^{-1}V_{AB}(x_B - \mu_B). \quad (9)$$

Similarly, we use the fact that the inverse covariance matrix of a Gaussian distribution  $p(\cdot)$  is a negative Hessian of  $\log p(\cdot)$ . In other words, the inverse covariance matrix of a Gaussian distribution  $p(x_A|x_B)$  is a negative Hessian of  $\log p(x_A|x_B)$ . Using Equation (5), we have

$$\Sigma^{*-1} = -\nabla_{x_A} \nabla_{x_A}^T \log p(x_A | x_B) \quad (10)$$

$$= V_{AA}. \quad (11)$$

Therefore, we get

$$\Sigma^* = V_{AA}^{-1}. \quad (12)$$

□

# The Multivariate Gaussian Distribution

Chuong B. Do

July 10, 2019

A vector-valued random variable  $X = [X_1 \cdots X_d]^T$  is said to have a **multivariate normal (or Gaussian) distribution** with mean  $\mu \in \mathbf{R}^d$  and covariance matrix  $\Sigma \in \mathbf{S}_{++}^{d-1}$  if its probability density function<sup>2</sup> is given by

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

We write this as  $X \sim \mathcal{N}(\mu, \Sigma)$ . In these notes, we describe multivariate Gaussians and some of their basic properties.

## 1 Relationship to univariate Gaussians

Recall that the density function of a **univariate normal (or Gaussian) distribution** is given by

$$p(x; \mu, \sigma^2) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{1}{2\sigma^2}(x - \mu)^2\right).$$

Here, the argument of the exponential function,  $-\frac{1}{2\sigma^2}(x - \mu)^2$ , is a quadratic function of the variable  $x$ . Furthermore, the parabola points downwards, as the coefficient of the quadratic term is negative. The coefficient in front,  $\frac{1}{\sqrt{2\pi}\sigma}$ , is a constant that does not depend on  $x$ ; hence, we can think of it as simply a “normalization factor” used to ensure that

$$\frac{1}{\sqrt{2\pi}\sigma} \int_{-\infty}^{\infty} \exp\left(-\frac{1}{2\sigma^2}(x - \mu)^2\right) dx = 1.$$

---

<sup>1</sup>Recall from the section notes on linear algebra that  $\mathbf{S}_{++}^d$  is the space of symmetric positive definite  $n \times d$  matrices, defined as

$$\mathbf{S}_{++}^d = \{A \in \mathbf{R}^{d \times d} : A = A^T \text{ and } x^T A x > 0 \text{ for all } x \in \mathbf{R}^d \text{ such that } x \neq 0\}.$$

<sup>2</sup>In these notes, we use the notation  $p(\bullet)$  to denote density functions, instead of  $f_X(\bullet)$  (as in the section notes on probability theory).

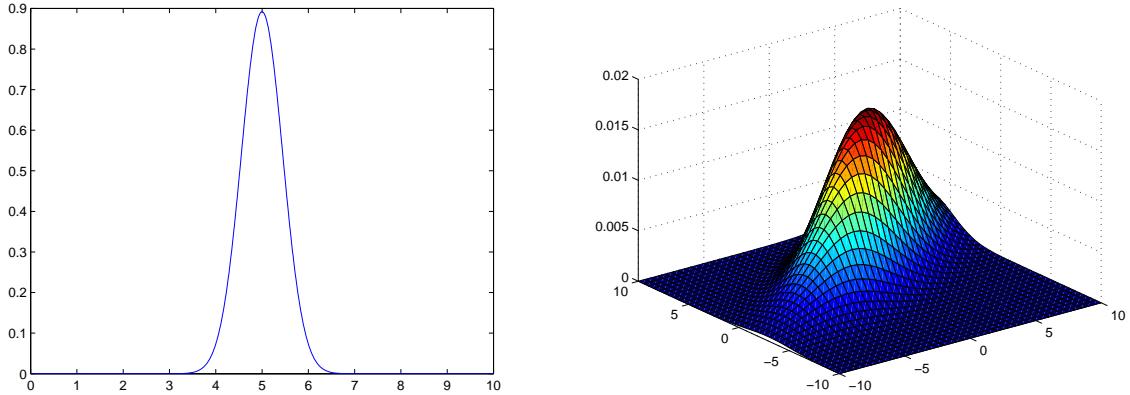


Figure 1: The figure on the left shows a univariate Gaussian density for a single variable  $X$ . The figure on the right shows a multivariate Gaussian density over two variables  $X_1$  and  $X_2$ .

In the case of the multivariate Gaussian density, the argument of the exponential function,  $-\frac{1}{2}(x - \mu)^T \Sigma^{-1}(x - \mu)$ , is a **quadratic form** in the vector variable  $x$ . Since  $\Sigma$  is positive definite, and since the inverse of any positive definite matrix is also positive definite, then for any non-zero vector  $z$ ,  $z^T \Sigma^{-1} z > 0$ . This implies that for any vector  $x \neq \mu$ ,

$$\begin{aligned} (x - \mu)^T \Sigma^{-1}(x - \mu) &> 0 \\ -\frac{1}{2}(x - \mu)^T \Sigma^{-1}(x - \mu) &< 0. \end{aligned}$$

Like in the univariate case, you can think of the argument of the exponential function as being a downward opening quadratic bowl. The coefficient in front (i.e.,  $\frac{1}{(2\pi)^{d/2} |\Sigma|^{1/2}}$ ) has an even more complicated form than in the univariate case. However, it still does not depend on  $x$ , and hence it is again simply a normalization factor used to ensure that

$$\frac{1}{(2\pi)^{d/2} |\Sigma|^{1/2}} \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \cdots \int_{-\infty}^{\infty} \exp\left(-\frac{1}{2}(x - \mu)^T \Sigma^{-1}(x - \mu)\right) dx_1 dx_2 \cdots dx_d = 1.$$

## 2 The covariance matrix

The concept of the **covariance matrix** is vital to understanding multivariate Gaussian distributions. Recall that for a pair of random variables  $X$  and  $Y$ , their **covariance** is defined as

$$Cov[X, Y] = E[(X - E[X])(Y - E[Y])] = E[XY] - E[X]E[Y].$$

When working with multiple variables, the covariance matrix provides a succinct way to summarize the covariances of all pairs of variables. In particular, the covariance matrix, which we usually denote as  $\Sigma$ , is the  $n \times d$  matrix whose  $(i, j)$ th entry is  $Cov[X_i, X_j]$ .

The following proposition (whose proof is provided in the Appendix A.1) gives an alternative way to characterize the covariance matrix of a random vector  $X$ :

**Proposition 1.** *For any random vector  $X$  with mean  $\mu$  and covariance matrix  $\Sigma$ ,*

$$\Sigma = E[(X - \mu)(X - \mu)^T] = E[XX^T] - \mu\mu^T. \quad (1)$$

In the definition of multivariate Gaussians, we required that the covariance matrix  $\Sigma$  be symmetric positive definite (i.e.,  $\Sigma \in \mathbf{S}_{++}^d$ ). Why does this restriction exist? As seen in the following proposition, the covariance matrix of *any* random vector must always be symmetric positive semidefinite:

**Proposition 2.** *Suppose that  $\Sigma$  is the covariance matrix corresponding to some random vector  $X$ . Then  $\Sigma$  is symmetric positive semidefinite.*

*Proof.* The symmetry of  $\Sigma$  follows immediately from its definition. Next, for any vector  $z \in \mathbf{R}^d$ , observe that

$$z^T \Sigma z = \sum_{i=1}^d \sum_{j=1}^d (\Sigma_{ij} z_i z_j) \quad (2)$$

$$= \sum_{i=1}^d \sum_{j=1}^d (Cov[X_i, X_j] \cdot z_i z_j)$$

$$= \sum_{i=1}^d \sum_{j=1}^d (E[(X_i - E[X_i])(X_j - E[X_j])] \cdot z_i z_j)$$

$$= E \left[ \sum_{i=1}^d \sum_{j=1}^d (X_i - E[X_i])(X_j - E[X_j]) \cdot z_i z_j \right]. \quad (3)$$

Here, (2) follows from the formula for expanding a quadratic form (see section notes on linear algebra), and (3) follows by linearity of expectations (see probability notes).

To complete the proof, observe that the quantity inside the brackets is of the form  $\sum_i \sum_j x_i x_j z_i z_j = (x^T z)^2 \geq 0$  (see problem set #1). Therefore, the quantity inside the expectation is always nonnegative, and hence the expectation itself must be nonnegative. We conclude that  $z^T \Sigma z \geq 0$ .  $\square$

From the above proposition it follows that  $\Sigma$  must be symmetric positive semidefinite in order for it to be a valid covariance matrix. However, in order for  $\Sigma^{-1}$  to exist (as required in the definition of the multivariate Gaussian density), then  $\Sigma$  must be invertible and hence full rank. Since any full rank symmetric positive semidefinite matrix is necessarily symmetric positive definite, it follows that  $\Sigma$  must be symmetric positive definite.

### 3 The diagonal covariance matrix case

To get an intuition for what a multivariate Gaussian is, consider the simple case where  $n = 2$ , and where the covariance matrix  $\Sigma$  is diagonal, i.e.,

$$x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \quad \mu = \begin{bmatrix} \mu_1 \\ \mu_2 \end{bmatrix} \quad \Sigma = \begin{bmatrix} \sigma_1^2 & 0 \\ 0 & \sigma_2^2 \end{bmatrix}$$

In this case, the multivariate Gaussian density has the form,

$$\begin{aligned} p(x; \mu, \Sigma) &= \frac{1}{2\pi \begin{vmatrix} \sigma_1^2 & 0 \\ 0 & \sigma_2^2 \end{vmatrix}^{1/2}} \exp \left( -\frac{1}{2} \begin{bmatrix} x_1 - \mu_1 \\ x_2 - \mu_2 \end{bmatrix}^T \begin{bmatrix} \sigma_1^2 & 0 \\ 0 & \sigma_2^2 \end{bmatrix}^{-1} \begin{bmatrix} x_1 - \mu_1 \\ x_2 - \mu_2 \end{bmatrix} \right) \\ &= \frac{1}{2\pi(\sigma_1^2 \cdot \sigma_2^2 - 0 \cdot 0)^{1/2}} \exp \left( -\frac{1}{2} \begin{bmatrix} x_1 - \mu_1 \\ x_2 - \mu_2 \end{bmatrix}^T \begin{bmatrix} \frac{1}{\sigma_1^2} & 0 \\ 0 & \frac{1}{\sigma_2^2} \end{bmatrix} \begin{bmatrix} x_1 - \mu_1 \\ x_2 - \mu_2 \end{bmatrix} \right), \end{aligned}$$

where we have relied on the explicit formula for the determinant of a  $2 \times 2$  matrix<sup>3</sup>, and the fact that the inverse of a diagonal matrix is simply found by taking the reciprocal of each diagonal entry. Continuing,

$$\begin{aligned} p(x; \mu, \Sigma) &= \frac{1}{2\pi\sigma_1\sigma_2} \exp \left( -\frac{1}{2} \begin{bmatrix} x_1 - \mu_1 \\ x_2 - \mu_2 \end{bmatrix}^T \begin{bmatrix} \frac{1}{\sigma_1^2}(x_1 - \mu_1) \\ \frac{1}{\sigma_2^2}(x_2 - \mu_2) \end{bmatrix} \right) \\ &= \frac{1}{2\pi\sigma_1\sigma_2} \exp \left( -\frac{1}{2\sigma_1^2}(x_1 - \mu_1)^2 - \frac{1}{2\sigma_2^2}(x_2 - \mu_2)^2 \right) \\ &= \frac{1}{\sqrt{2\pi}\sigma_1} \exp \left( -\frac{1}{2\sigma_1^2}(x_1 - \mu_1)^2 \right) \cdot \frac{1}{\sqrt{2\pi}\sigma_2} \exp \left( -\frac{1}{2\sigma_2^2}(x_2 - \mu_2)^2 \right). \end{aligned}$$

The last equation we recognize to simply be the product of two independent Gaussian densities, one with mean  $\mu_1$  and variance  $\sigma_1^2$ , and the other with mean  $\mu_2$  and variance  $\sigma_2^2$ .

More generally, one can show that an  $d$ -dimensional Gaussian with mean  $\mu \in \mathbf{R}^d$  and diagonal covariance matrix  $\Sigma = \text{diag}(\sigma_1^2, \sigma_2^2, \dots, \sigma_d^2)$  is the same as a collection of  $d$  independent Gaussian random variables with mean  $\mu_i$  and variance  $\sigma_i^2$ , respectively.

### 4 Isocontours

Another way to understand a multivariate Gaussian conceptually is to understand the shape of its **isocontours**. For a function  $f : \mathbf{R}^2 \rightarrow \mathbf{R}$ , an isocontour is a set of the form

$$\{x \in \mathbf{R}^2 : f(x) = c\}.$$

for some  $c \in \mathbf{R}$ .<sup>4</sup>

---

<sup>3</sup>Namely,  $\begin{vmatrix} a & b \\ c & d \end{vmatrix} = ad - bc$ .

<sup>4</sup>Isocontours are often also known as **level curves**. More generally, a **level set** of a function  $f : \mathbf{R}^d \rightarrow \mathbf{R}$ , is a set of the form  $\{x \in \mathbf{R}^d : f(x) = c\}$  for some  $c \in \mathbf{R}$ .

## 4.1 Shape of isocontours

What do the isocontours of a multivariate Gaussian look like? As before, let's consider the case where  $n = 2$ , and  $\Sigma$  is diagonal, i.e.,

$$x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \quad \mu = \begin{bmatrix} \mu_1 \\ \mu_2 \end{bmatrix} \quad \Sigma = \begin{bmatrix} \sigma_1^2 & 0 \\ 0 & \sigma_2^2 \end{bmatrix}$$

As we showed in the last section,

$$p(x; \mu, \Sigma) = \frac{1}{2\pi\sigma_1\sigma_2} \exp\left(-\frac{1}{2\sigma_1^2}(x_1 - \mu_1)^2 - \frac{1}{2\sigma_2^2}(x_2 - \mu_2)^2\right). \quad (4)$$

Now, let's consider the level set consisting of all points where  $p(x; \mu, \Sigma) = c$  for some constant  $c \in \mathbf{R}$ . In particular, consider the set of all  $x_1, x_2 \in \mathbf{R}$  such that

$$\begin{aligned} c &= \frac{1}{2\pi\sigma_1\sigma_2} \exp\left(-\frac{1}{2\sigma_1^2}(x_1 - \mu_1)^2 - \frac{1}{2\sigma_2^2}(x_2 - \mu_2)^2\right) \\ 2\pi c \sigma_1 \sigma_2 &= \exp\left(-\frac{1}{2\sigma_1^2}(x_1 - \mu_1)^2 - \frac{1}{2\sigma_2^2}(x_2 - \mu_2)^2\right) \\ \log(2\pi c \sigma_1 \sigma_2) &= -\frac{1}{2\sigma_1^2}(x_1 - \mu_1)^2 - \frac{1}{2\sigma_2^2}(x_2 - \mu_2)^2 \\ \log\left(\frac{1}{2\pi c \sigma_1 \sigma_2}\right) &= \frac{1}{2\sigma_1^2}(x_1 - \mu_1)^2 + \frac{1}{2\sigma_2^2}(x_2 - \mu_2)^2 \\ 1 &= \frac{(x_1 - \mu_1)^2}{2\sigma_1^2 \log\left(\frac{1}{2\pi c \sigma_1 \sigma_2}\right)} + \frac{(x_2 - \mu_2)^2}{2\sigma_2^2 \log\left(\frac{1}{2\pi c \sigma_1 \sigma_2}\right)}. \end{aligned}$$

Defining

$$r_1 = \sqrt{2\sigma_1^2 \log\left(\frac{1}{2\pi c \sigma_1 \sigma_2}\right)} \quad r_2 = \sqrt{2\sigma_2^2 \log\left(\frac{1}{2\pi c \sigma_1 \sigma_2}\right)},$$

it follows that

$$1 = \left(\frac{x_1 - \mu_1}{r_1}\right)^2 + \left(\frac{x_2 - \mu_2}{r_2}\right)^2. \quad (5)$$

Equation (5) should be familiar to you from high school analytic geometry: it is the equation of an **axis-aligned ellipse**, with center  $(\mu_1, \mu_2)$ , where the  $x_1$  axis has length  $2r_1$  and the  $x_2$  axis has length  $2r_2$ !

## 4.2 Length of axes

To get a better understanding of how the shape of the level curves vary as a function of the variances of the multivariate Gaussian distribution, suppose that we are interested in

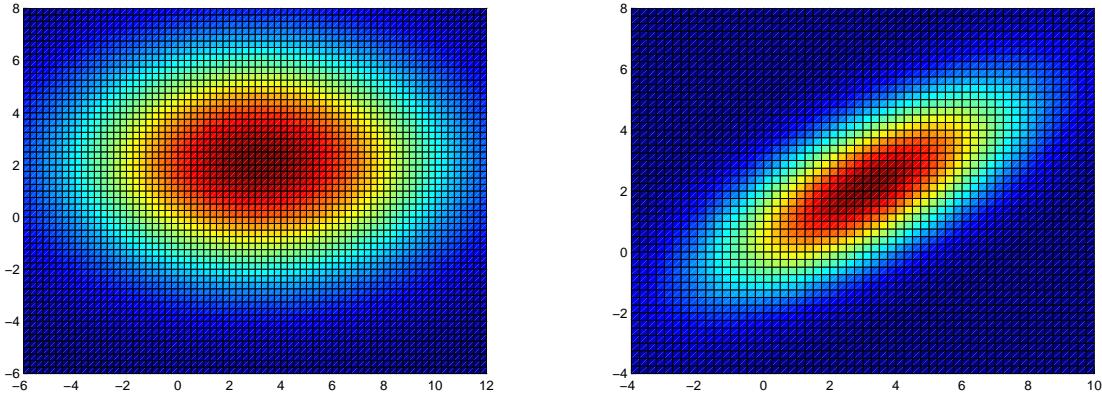


Figure 2:

The figure on the left shows a heatmap indicating values of the density function for an axis-aligned multivariate Gaussian with mean  $\mu = \begin{bmatrix} 3 \\ 2 \end{bmatrix}$  and diagonal covariance matrix  $\Sigma = \begin{bmatrix} 25 & 0 \\ 0 & 9 \end{bmatrix}$ . Notice that the Gaussian is centered at (3, 2), and that the isocontours are all elliptically shaped with major/minor axis lengths in a 5:3 ratio. The figure on the right shows a heatmap indicating values of the density function for a non axis-aligned multivariate Gaussian with mean  $\mu = \begin{bmatrix} 3 \\ 2 \end{bmatrix}$  and covariance matrix  $\Sigma = \begin{bmatrix} 10 & 5 \\ 5 & 5 \end{bmatrix}$ . Here, the ellipses are again centered at (3, 2), but now the major and minor axes have been rotated via a linear transformation.

the values of  $r_1$  and  $r_2$  at which  $c$  is equal to a fraction  $1/e$  of the peak height of Gaussian density.

First, observe that maximum of Equation (4) occurs where  $x_1 = \mu_1$  and  $x_2 = \mu_2$ . Substituting these values into Equation (4), we see that the peak height of the Gaussian density is  $\frac{1}{2\pi\sigma_1\sigma_2}$ .

Second, we substitute  $c = \frac{1}{e} \left( \frac{1}{2\pi\sigma_1\sigma_2} \right)$  into the equations for  $r_1$  and  $r_2$  to obtain

$$r_1 = \sqrt{2\sigma_1^2 \log \left( \frac{1}{2\pi\sigma_1\sigma_2 \cdot \frac{1}{e} \left( \frac{1}{2\pi\sigma_1\sigma_2} \right)} \right)} = \sigma_1\sqrt{2}$$

$$r_2 = \sqrt{2\sigma_2^2 \log \left( \frac{1}{2\pi\sigma_1\sigma_2 \cdot \frac{1}{e} \left( \frac{1}{2\pi\sigma_1\sigma_2} \right)} \right)} = \sigma_2\sqrt{2}.$$

From this, it follows that the axis length needed to reach a fraction  $1/e$  of the peak height of the Gaussian density in the  $i$ th dimension grows in proportion to the standard deviation  $\sigma_i$ . Intuitively, this again makes sense: the smaller the variance of some random variable  $x_i$ , the more “tightly” peaked the Gaussian distribution in that dimension, and hence the smaller the radius  $r_i$ .

### 4.3 Non-diagonal case, higher dimensions

Clearly, the above derivations rely on the assumption that  $\Sigma$  is a diagonal matrix. However, in the non-diagonal case, it turns out that the picture is not all that different. Instead of being an axis-aligned ellipse, the isocontours turn out to be simply **rotated ellipses**. Furthermore, in the  $d$ -dimensional case, the level sets form geometrical structures known as **ellipsoids** in  $\mathbf{R}^d$ .

## 5 Linear transformation interpretation

In the last few sections, we focused primarily on providing an intuition for how multivariate Gaussians with diagonal covariance matrices behaved. In particular, we found that an  $d$ -dimensional multivariate Gaussian with diagonal covariance matrix could be viewed simply as a collection of  $d$  independent Gaussian-distributed random variables with means and variances  $\mu_i$  and  $\sigma_i^2$ , respectively. In this section, we dig a little deeper and provide a quantitative interpretation of multivariate Gaussians when the covariance matrix is not diagonal.

The key result of this section is the following theorem (see proof in Appendix A.2).

**Theorem 1.** *Let  $X \sim \mathcal{N}(\mu, \Sigma)$  for some  $\mu \in \mathbf{R}^d$  and  $\Sigma \in \mathbf{S}_{++}^d$ . Then, there exists a matrix  $B \in \mathbf{R}^{d \times d}$  such that if we define  $Z = B^{-1}(X - \mu)$ , then  $Z \sim \mathcal{N}(0, I)$ .*

To understand the meaning of this theorem, note that if  $Z \sim \mathcal{N}(0, I)$ , then using the analysis from Section 4,  $Z$  can be thought of as a collection of  $d$  independent standard normal random variables (i.e.,  $Z_i \sim \mathcal{N}(0, 1)$ ). Furthermore, if  $Z = B^{-1}(X - \mu)$  then  $X = BZ + \mu$  follows from simple algebra.

Consequently, the theorem states that any random variable  $X$  with a multivariate Gaussian distribution can be interpreted as the result of applying a linear transformation ( $X = BZ + \mu$ ) to some collection of  $d$  independent standard normal random variables ( $Z$ ).

## Appendix A.1

*Proof.* We prove the first of the two equalities in (1); the proof of the other equality is similar.

$$\begin{aligned}
\Sigma &= \begin{bmatrix} Cov[X_1, X_1] & \cdots & Cov[X_1, X_d] \\ \vdots & \ddots & \vdots \\ Cov[X_d, X_1] & \cdots & Cov[X_d, X_d] \end{bmatrix} \\
&= \begin{bmatrix} E[(X_1 - \mu_1)^2] & \cdots & E[(X_1 - \mu_1)(X_d - \mu_d)] \\ \vdots & \ddots & \vdots \\ E[(X_d - \mu_d)(X_1 - \mu_1)] & \cdots & E[(X_d - \mu_d)^2] \end{bmatrix} \\
&= E \begin{bmatrix} (X_1 - \mu_1)^2 & \cdots & (X_1 - \mu_1)(X_d - \mu_d) \\ \vdots & \ddots & \vdots \\ (X_d - \mu_d)(X_1 - \mu_1) & \cdots & (X_d - \mu_d)^2 \end{bmatrix} \tag{6} \\
&= E \begin{bmatrix} [X_1 - \mu_1] \\ \vdots \\ [X_d - \mu_d] \end{bmatrix} [X_1 - \mu_1 \ \cdots \ X_d - \mu_d] \\
&= E [(X - \mu)(X - \mu)^T]. \tag{7}
\end{aligned}$$

Here, (6) follows from the fact that the expectation of a matrix is simply the matrix found by taking the componentwise expectation of each entry. Also, (7) follows from the fact that for any vector  $z \in \mathbf{R}^d$ ,

$$zz^T = \begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_d \end{bmatrix} [z_1 \ z_2 \ \cdots z_d] = \begin{bmatrix} z_1 z_1 & z_1 z_2 & \cdots & z_1 z_d \\ z_2 z_1 & z_2 z_2 & \cdots & z_2 z_d \\ \vdots & \vdots & \ddots & \vdots \\ z_d z_1 & z_d z_2 & \cdots & z_d z_d \end{bmatrix}.$$

□

## Appendix A.2

We restate the theorem below:

**Theorem 1.** *Let  $X \sim \mathcal{N}(\mu, \Sigma)$  for some  $\mu \in \mathbf{R}^d$  and  $\Sigma \in \mathbf{S}_{++}^d$ . Then, there exists a matrix  $B \in \mathbf{R}^{d \times d}$  such that if we define  $Z = B^{-1}(X - \mu)$ , then  $Z \sim \mathcal{N}(0, I)$ .*

The derivation of this theorem requires some advanced linear algebra and probability theory and can be skipped for the purposes of this class. Our argument will consist of two parts. First, we will show that the covariance matrix  $\Sigma$  can be factorized as  $\Sigma = BB^T$  for some invertible matrix  $B$ . Second, we will perform a “change-of-variable” from  $X$  to a different vector valued random variable  $Z$  using the relation  $Z = B^{-1}(X - \mu)$ .

**Step 1: Factorizing the covariance matrix.** Recall the following two properties of symmetric matrices from the notes on linear algebra<sup>5</sup>:

1. Any real symmetric matrix  $A \in \mathbf{R}^{d \times d}$  can always be represented as  $A = U\Lambda U^T$ , where  $U$  is a full rank orthogonal matrix containing of the eigenvectors of  $A$  as its columns, and  $\Lambda$  is a diagonal matrix containing  $A$ 's eigenvalues.
2. If  $A$  is symmetric positive definite, all its eigenvalues are positive.

Since the covariance matrix  $\Sigma$  is positive definite, using the first fact, we can write  $\Sigma = U\Lambda U^T$  for some appropriately defined matrices  $U$  and  $\Lambda$ . Using the second fact, we can define  $\Lambda^{1/2} \in \mathbf{R}^{d \times d}$  to be the diagonal matrix whose entries are the square roots of the corresponding entries from  $\Lambda$ . Since  $\Lambda = \Lambda^{1/2}(\Lambda^{1/2})^T$ , we have

$$\Sigma = U\Lambda U^T = U\Lambda^{1/2}(\Lambda^{1/2})^T U^T = U\Lambda^{1/2}(U\Lambda^{1/2})^T = BB^T,$$

where  $B = U\Lambda^{1/2}$ .<sup>6</sup> In this case, then  $\Sigma^{-1} = B^{-T}B^{-1}$ , so we can rewrite the standard formula for the density of a multivariate Gaussian as

$$p(x; \mu, \Sigma) = \frac{1}{(2\pi)^{d/2}|BB^T|^{1/2}} \exp\left(-\frac{1}{2}(x - \mu)^T B^{-T}B^{-1}(x - \mu)\right). \quad (8)$$

**Step 2: Change of variables.** Now, define the vector-valued random variable  $Z = B^{-1}(X - \mu)$ . A basic formula of probability theory, which we did not introduce in the section notes on probability theory, is the “change-of-variables” formula for relating vector-valued random variables:

Suppose that  $X = [X_1 \dots X_d]^T \in \mathbf{R}^d$  is a vector-valued random variable with joint density function  $f_X : \mathbf{R}^d \rightarrow \mathbf{R}$ . If  $Z = H(X) \in \mathbf{R}^d$  where  $H$  is a bijective, differentiable function, then  $Z$  has joint density  $f_Z : \mathbf{R}^d \rightarrow \mathbf{R}$ , where

$$f_Z(z) = f_X(x) \cdot \left| \det \left( \begin{bmatrix} \frac{\partial x_1}{\partial z_1} & \cdots & \frac{\partial x_1}{\partial z_d} \\ \vdots & \ddots & \vdots \\ \frac{\partial x_d}{\partial z_1} & \cdots & \frac{\partial x_d}{\partial z_d} \end{bmatrix} \right) \right|.$$

Using the change-of-variable formula, one can show (after some algebra, which we'll skip) that the vector variable  $Z$  has the following joint density:

$$p_Z(z) = \frac{1}{(2\pi)^{d/2}} \exp\left(-\frac{1}{2}z^T z\right). \quad (9)$$

The claim follows immediately. □

---

<sup>5</sup>See section on “Eigenvalues and Eigenvectors of Symmetric Matrices.”

<sup>6</sup>To show that  $B$  is invertible, it suffices to observe that  $U$  is an invertible matrix, and right-multiplying  $U$  by a diagonal matrix (with no zero diagonal entries) will rescale its columns but will not change its rank.

# More on Multivariate Gaussians

Chuong B. Do

July 10, 2019

Up to this point in class, you have seen multivariate Gaussians arise in a number of applications, such as the probabilistic interpretation of linear regression, Gaussian discriminant analysis, mixture of Gaussians clustering, and most recently, factor analysis. In these lecture notes, we attempt to demystify some of the fancier properties of multivariate Gaussians that were introduced in the recent factor analysis lecture. The goal of these notes is to give you some intuition into where these properties come from, so that you can use them with confidence on your homework (hint hint!) and beyond.

## 1 Definition

A vector-valued random variable  $x \in \mathbf{R}^d$  is said to have a **multivariate normal (or Gaussian) distribution** with mean  $\mu \in \mathbf{R}^d$  and covariance matrix  $\Sigma \in \mathbf{S}_{++}^{d-1}$  if its probability density function is given by

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

We write this as  $x \sim \mathcal{N}(\mu, \Sigma)$ .

## 2 Gaussian facts

Multivariate Gaussians turn out to be extremely handy in practice due to the following facts:

- **Fact #1:** If you know the mean  $\mu$  and covariance matrix  $\Sigma$  of a Gaussian random variable  $x$ , you can write down the probability density function for  $x$  directly.

---

<sup>1</sup>Recall from the section notes on linear algebra that  $\mathbf{S}_{++}^d$  is the space of symmetric positive definite  $n \times d$  matrices, defined as

$$\mathbf{S}_{++}^d = \{A \in \mathbf{R}^{d \times d} : A = A^T \text{ and } x^T A x > 0 \text{ for all } x \in \mathbf{R}^d \text{ such that } x \neq 0\}.$$

- **Fact #2:** The following Gaussian integrals have closed-form solutions:

$$\int_{x \in \mathbf{R}^d} p(x; \mu, \Sigma) dx = \int_{-\infty}^{\infty} \cdots \int_{-\infty}^{\infty} p(x; \mu, \Sigma) dx_1 \dots dx_d = 1$$

$$\int_{x \in \mathbf{R}^d} x_i p(x; \mu, \sigma^2) dx = \mu_i$$

$$\int_{x \in \mathbf{R}^d} (x_i - \mu_i)(x_j - \mu_j) p(x; \mu, \sigma^2) dx = \Sigma_{ij}.$$

- **Fact #3:** Gaussians obey a number of *closure* properties:

- The sum of independent Gaussian random variables is Gaussian.
- The marginal of a joint Gaussian distribution is Gaussian.
- The conditional of a joint Gaussian distribution is Gaussian.

At first glance, some of these facts, in particular facts #1 and #2, may seem either intuitively obvious or at least plausible. What is probably not so clear, however, is why these facts are so powerful. In this document, we'll provide some intuition for how these facts can be used when performing day-to-day manipulations dealing with multivariate Gaussian random variables.

## 3 Closure properties

In this section, we'll go through each of the closure properties described earlier, and we'll either prove the property using facts #1 and #2, or we'll at least give some type of intuition as to why the property is true.

The following is a quick roadmap of what we'll cover:

	sums	marginals	conditionals
why is it Gaussian?	no	yes	yes
resulting density function	yes	yes	yes

### 3.1 Sum of independent Gaussians is Gaussian

The formal statement of this rule is:

Suppose that  $y \sim \mathcal{N}(\mu, \Sigma)$  and  $z \sim \mathcal{N}(\mu', \Sigma')$  are independent Gaussian distributed random variables, where  $\mu, \mu' \in \mathbf{R}^d$  and  $\Sigma, \Sigma' \in \mathbf{S}_{++}^d$ . Then, their sum is also Gaussian:

$$y + z \sim \mathcal{N}(\mu + \mu', \Sigma + \Sigma').$$

Before we prove anything, here are some observations:

1. The first thing to point out is that the importance of the independence assumption in the above rule. To see why this matters, suppose that  $y \sim \mathcal{N}(\mu, \Sigma)$  for some mean vector  $\mu$  and covariance matrix  $\Sigma$ , and suppose that  $z = -y$ . Clearly,  $z$  also has a Gaussian distribution (in fact,  $z \sim \mathcal{N}(-\mu, \Sigma)$ ), but  $y + z$  is identically zero!
2. The second thing to point out is a point of confusion for many students: if we add together two Gaussian densities (“bumps” in multidimensional space), wouldn’t we get back some bimodal (i.e., “two-humped” density)? Here, the thing to realize is that the density of the random variable  $y + z$  in this rule is NOT found by simply adding the densities of the individual random variables  $y$  and  $z$ . Rather, the density of  $y + z$  will actually turn out to be a *convolution* of the densities for  $y$  and  $z$ .<sup>2</sup> To show that the convolution of two Gaussian densities gives a Gaussian density, however, is beyond the scope of this class.

Instead, let’s just use the observation that the convolution does give some type of Gaussian density, along with Fact #1, to figure out what the density,  $p(y + z | \mu, \Sigma)$  would be, if we were to actually compute the convolution. How can we do this? Recall that from Fact #1, a Gaussian distribution is fully specified by its mean vector and covariance matrix. If we can determine what these are, then we’re done.

But this is easy! For the mean, we have

$$E[y_i + z_i] = E[y_i] + E[z_i] = \mu_i + \mu'_i$$

from linearity of expectations. Therefore, the mean of  $y + z$  is simply  $\mu + \mu'$ . Also, the  $(i, j)$ th entry of the covariance matrix is given by

$$\begin{aligned} & E[(y_i + z_i)(y_j + z_j)] - E[y_i + z_i]E[y_j + z_j] \\ &= E[y_i y_j + z_i y_j + y_i z_j + z_i z_j] - (E[y_i] + E[z_i])(E[y_j] + E[z_j]) \\ &= E[y_i y_j] + E[z_i y_j] + E[y_i z_j] + E[z_i z_j] - E[y_i]E[y_j] - E[z_i]E[y_j] - E[y_i]E[z_j] - E[z_i]E[z_j] \\ &= (E[y_i y_j] - E[y_i]E[y_j]) + (E[z_i z_j] - E[z_i]E[z_j]) \\ &\quad + (E[z_i y_j] - E[z_i]E[y_j]) + (E[y_i z_j] - E[y_i]E[z_j]). \end{aligned}$$

Using the fact that  $y$  and  $z$  are independent, we have  $E[z_i y_j] = E[z_i]E[y_j]$  and  $E[y_i z_j] = E[y_i]E[z_j]$ . Therefore, the last two terms drop out, and we are left with,

$$\begin{aligned} & E[(y_i + z_i)(y_j + z_j)] - E[y_i + z_i]E[y_j + z_j] \\ &= (E[y_i y_j] - E[y_i]E[y_j]) + (E[z_i z_j] - E[z_i]E[z_j]) \\ &= \Sigma_{ij} + \Sigma'_{ij}. \end{aligned}$$

---

<sup>2</sup>For example, if  $y$  and  $z$  were univariate Gaussians (i.e.,  $y \sim \mathcal{N}(\mu, \sigma^2)$ ,  $z \sim \mathcal{N}(\mu', \sigma'^2)$ ), then the convolution of their probability densities is given by

$$\begin{aligned} p(y + z; \mu, \mu', \sigma^2, \sigma'^2) &= \int_{-\infty}^{\infty} p(w; \mu, \sigma^2)p(y + z - w; \mu', \sigma'^2)dw \\ &= \int_{-\infty}^{\infty} \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{1}{2\sigma^2}(w - \mu)^2\right) \cdot \frac{1}{\sqrt{2\pi}\sigma'} \exp\left(-\frac{1}{2\sigma'^2}(y + z - w - \mu')^2\right)dw \end{aligned}$$

From this, we can conclude that the covariance matrix of  $y + z$  is simply  $\Sigma + \Sigma'$ .

At this point, take a step back and think about what we have just done. Using some simple properties of expectations and independence, we have computed the mean and covariance matrix of  $y + z$ . Because of Fact #1, we can thus write down the density for  $y + z$  immediately, without the need to perform a convolution!<sup>3</sup>

## 3.2 Marginal of a joint Gaussian is Gaussian

The formal statement of this rule is:

Suppose that

$$\begin{bmatrix} x_A \\ x_B \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} \mu_A \\ \mu_B \end{bmatrix}, \begin{bmatrix} \Sigma_{AA} & \Sigma_{AB} \\ \Sigma_{BA} & \Sigma_{BB} \end{bmatrix}\right),$$

where  $x_A \in \mathbf{R}^n$ ,  $x_B \in \mathbf{R}^d$ , and the dimensions of the mean vectors and covariance matrix subblocks are chosen to match  $x_A$  and  $x_B$ . Then, the marginal densities,

$$\begin{aligned} p(x_A) &= \int_{x_B \in \mathbf{R}^d} p(x_A, x_B; \mu, \Sigma) dx_B \\ p(x_B) &= \int_{x_A \in \mathbf{R}^n} p(x_A, x_B; \mu, \Sigma) dx_A \end{aligned}$$

are Gaussian:

$$\begin{aligned} x_A &\sim \mathcal{N}(\mu_A, \Sigma_{AA}) \\ x_B &\sim \mathcal{N}(\mu_B, \Sigma_{BB}). \end{aligned}$$

To justify this rule, let's just focus on the marginal distribution with respect to the variables  $x_A$ .<sup>4</sup>

First, note that computing the mean and covariance matrix for a marginal distribution is easy: simply take the corresponding subblocks from the mean and covariance matrix of the joint density. To make sure this is absolutely clear, let's look at the covariance between  $x_{A,i}$  and  $x_{A,j}$  (the  $i$ th component of  $x_A$  and the  $j$ th component of  $x_A$ ). Note that  $x_{A,i}$  and  $x_{A,j}$  are also the  $i$ th and  $j$ th components of

$$\begin{bmatrix} x_A \\ x_B \end{bmatrix}$$

---

<sup>3</sup>Of course, we needed to know that  $y + z$  had a Gaussian distribution in the first place.

<sup>4</sup>In general, for a random vector  $x$  which has a Gaussian distribution, we can always permute entries of  $x$  so long as we permute the entries of the mean vector and the rows/columns of the covariance matrix in the corresponding way. As a result, it suffices to look only at  $x_A$ , and the result for  $x_B$  follows immediately.

(since  $x_A$  appears at the top of this vector). To find their covariance, we need to simply look at the  $(i, j)$ th element of the covariance matrix,

$$\begin{bmatrix} \Sigma_{AA} & \Sigma_{AB} \\ \Sigma_{BA} & \Sigma_{BB} \end{bmatrix}.$$

The  $(i, j)$ th element is found in the  $\Sigma_{AA}$  subblock, and in fact, is precisely  $\Sigma_{AA,ij}$ . Using this argument for all  $i, j \in \{1, \dots, m\}$ , we see that the covariance matrix for  $x_A$  is simply  $\Sigma_{AA}$ . A similar argument can be used to find that the mean of  $x_A$  is simply  $\mu_A$ . Thus, the above argument tells us that if we knew that the marginal distribution over  $x_A$  is Gaussian, then we could immediately write down a density function for  $x_A$  in terms of the appropriate submatrices of the mean and covariance matrices for the joint density!

The above argument, though simple, however, is somewhat unsatisfying: how can we actually be sure that  $x_A$  has a multivariate Gaussian distribution? The argument for this is slightly long-winded, so rather than saving up the punchline, here's our plan of attack up front:

1. Write the integral form of the marginal density explicitly.
2. Rewrite the integral by partitioning the inverse covariance matrix.
3. Use a “completion-of-squares” argument to evaluate the integral over  $x_B$ .
4. Argue that the resulting density is Gaussian.

Let's see each of these steps in action.

### 3.2.1 The marginal density in integral form

Suppose that we wanted to compute the density function of  $x_A$  directly. Then, we would need to compute the integral,

$$\begin{aligned} p(x_A) &= \int_{x_B \in \mathbf{R}^d} p(x_A, x_B; \mu, \Sigma) dx_B \\ &= \frac{1}{(2\pi)^{\frac{n+n}{2}} \left| \begin{bmatrix} \Sigma_{AA} & \Sigma_{AB} \\ \Sigma_{BA} & \Sigma_{BB} \end{bmatrix} \right|^{1/2}} \int_{x_B \in \mathbf{R}^d} \exp \left( -\frac{1}{2} \begin{bmatrix} x_A - \mu_A \\ x_B - \mu_B \end{bmatrix}^T \begin{bmatrix} \Sigma_{AA} & \Sigma_{AB} \\ \Sigma_{BA} & \Sigma_{BB} \end{bmatrix}^{-1} \begin{bmatrix} x_A - \mu_A \\ x_B - \mu_B \end{bmatrix} \right) dx_B. \end{aligned}$$

### 3.2.2 Partitioning the inverse covariance matrix

To make any sort of progress, we'll need to write the matrix product in the exponent in a slightly different form. In particular, let us define the matrix  $V \in \mathbf{R}^{(m+n) \times (m+n)}$  as<sup>5</sup>

$$V = \begin{bmatrix} V_{AA} & V_{AB} \\ V_{BA} & V_{BB} \end{bmatrix} = \Sigma^{-1}.$$

---

<sup>5</sup>Sometimes,  $V$  is called the “precision” matrix.

It might be tempting to think that

$$V = \begin{bmatrix} V_{AA} & V_{AB} \\ V_{BA} & V_{BB} \end{bmatrix} = \begin{bmatrix} \Sigma_{AA} & \Sigma_{AB} \\ \Sigma_{BA} & \Sigma_{BB} \end{bmatrix}^{-1} \text{ ``='' } \begin{bmatrix} \Sigma_{AA}^{-1} & \Sigma_{AB}^{-1} \\ \Sigma_{BA}^{-1} & \Sigma_{BB}^{-1} \end{bmatrix}$$

However, the rightmost equality does not hold! We'll return to this issue in a later step; for now, though, it suffices to define  $V$  as above without worrying what actual contents of each submatrix are.

Using this definition of  $V$ , the integral expands to

$$\begin{aligned} p(x_A) = \frac{1}{Z} \int_{x_B \in \mathbf{R}^d} \exp\left(-\left[\frac{1}{2}(x_A - \mu_A)^T V_{AA}(x_A - \mu_A) + \frac{1}{2}(x_A - \mu_A)^T V_{AB}(x_B - \mu_B) \right.\right. \\ \left.\left. + \frac{1}{2}(x_B - \mu_B)^T V_{BA}(x_A - \mu_A) + \frac{1}{2}(x_B - \mu_B)^T V_{BB}(x_B - \mu_B)\right]\right) dx_B, \end{aligned}$$

where  $Z$  is some constant not depending on either  $x_A$  or  $x_B$  that we'll choose to ignore for the moment. If you haven't worked with partitioned matrices before, then the expansion above may seem a little magical to you. It is analogous to the idea that when defining a quadratic form based on some  $2 \times 2$  matrix  $A$ , then

$$x^T A x = \sum_i \sum_j A_{ij} x_i x_j = x_1 A_{11} x_1 + x_1 A_{12} x_2 + x_2 A_{21} x_1 + x_2 A_{22} x_2.$$

Take some time to convince yourself that the matrix generalization above also holds.

### 3.2.3 Integrating out $x_B$

To evaluate the integral, we'll somehow want to integrate out  $x_B$ . In general, however, Gaussian integrals are hard to compute by hand. Is there anything we can do to save time? There are, in fact, a number of Gaussian integrals for which the answer is already known (see Fact #2). The basic idea in this section, then, will be to transform the integral we had in the last section into a form where we can apply one of the results from Fact #2 in order to perform the required integration easily.

The key to this is a mathematical trick known as "completion of squares." Consider the quadratic function  $z^T A z + b^T z + c$  where  $A$  is a symmetric, nonsingular matrix. Then, one can verify directly that

$$\frac{1}{2} z^T A z + b^T z + c = \frac{1}{2} (z + A^{-1}b)^T A (z + A^{-1}b) + c - \frac{1}{2} b^T A^{-1}b.$$

This is the multivariate generalization of the "completion of squares" argument used in single variable algebra:

$$\frac{1}{2} az^2 + bz + c = \frac{1}{2} a \left( z + \frac{b}{a} \right)^2 + c - \frac{b^2}{2a}$$

To apply the completion of squares in our situation above, let

$$\begin{aligned} z &= x_B - \mu_B \\ A &= V_{BB} \\ b &= V_{BA}(x_A - \mu_A) \\ c &= \frac{1}{2}(x_A - \mu_A)^T V_{AA}(x_A - \mu_A). \end{aligned}$$

Then, it follows that the integral can be rewritten as

$$p(x_A) = \frac{1}{Z} \int_{x_B \in \mathbf{R}^d} \exp \left( - \left[ \frac{1}{2} (x_B - \mu_B + V_{BB}^{-1} V_{BA}(x_A - \mu_A))^T V_{BB} (x_B - \mu_B + V_{BB}^{-1} V_{BA}(x_A - \mu_A)) \right. \right. \\ \left. \left. + \frac{1}{2} (x_A - \mu_A)^T V_{AA}(x_A - \mu_A) - \frac{1}{2} (x_A - \mu_A)^T V_{AB} V_{BB}^{-1} V_{BA}(x_A - \mu_A) \right] \right] dx_B$$

We can factor out the terms not including  $x_B$  to obtain,

$$p(x_A) = \exp \left( -\frac{1}{2} (x_A - \mu_A)^T V_{AA}(x_A - \mu_A) + \frac{1}{2} (x_A - \mu_A)^T V_{AB} V_{BB}^{-1} V_{BA}(x_A - \mu_A) \right) \\ \cdot \frac{1}{Z} \int_{x_B \in \mathbf{R}^d} \exp \left( -\frac{1}{2} \left[ (x_B - \mu_B + V_{BB}^{-1} V_{BA}(x_A - \mu_A))^T V_{BB} (x_B - \mu_B + V_{BB}^{-1} V_{BA}(x_A - \mu_A)) \right] \right] dx_B$$

At this point, we can now apply Fact #2. In particular, we know that generically speaking, for a multivariate Gaussian distributed random variable  $x$  with mean  $\mu$  and covariance matrix  $\Sigma$ , the density function normalizes, i.e.,

$$\frac{1}{(2\pi)^{d/2} |\Sigma|^{1/2}} \int_{\mathbf{R}^d} \exp \left( -\frac{1}{2} (x - \mu)^T \Sigma^{-1} (x - \mu) \right) = 1,$$

or equivalently,

$$\int_{\mathbf{R}^d} \exp \left( -\frac{1}{2} (x - \mu)^T \Sigma^{-1} (x - \mu) \right) = (2\pi)^{d/2} |\Sigma|^{1/2}.$$

We use this fact to get rid of the remaining integral in our expression for  $p(x_A)$ :

$$p(x_A) = \frac{1}{Z} \cdot (2\pi)^{d/2} |V_{BB}|^{1/2} \cdot \exp \left( -\frac{1}{2} (x_A - \mu_A)^T (V_{AA} - V_{AB} V_{BB}^{-1} V_{BA})(x_A - \mu_A) \right).$$

### 3.2.4 Arguing that resulting density is Gaussian

At this point, we are almost done! Ignoring the normalization constant in front, we see that the density of  $x_A$  is the exponential of a quadratic form in  $x_A$ . We can quickly recognize that our density is none other than a Gaussian with mean vector  $\mu_A$  and covariance matrix  $(V_{AA} - V_{AB} V_{BB}^{-1} V_{BA})^{-1}$ . Although the form of the covariance matrix may seem a bit complex,

we have already achieved what we set out to show in the first place—namely, that  $x_A$  has a marginal Gaussian distribution. Using the logic before, we can conclude that this covariance matrix must somehow reduce to  $\Sigma_{AA}$ .

But, in case you are curious, it's also possible to show that our derivation is consistent with this earlier justification. To do this, we use the following result for partitioned matrices:

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix}^{-1} = \begin{bmatrix} M^{-1} & -M^{-1}BD^{-1} \\ -D^{-1}CM^{-1} & D^{-1} + D^{-1}CM^{-1}BD^{-1} \end{bmatrix}.$$

where  $M = A - BD^{-1}C$ . This formula can be thought of as the multivariable generalization of the explicit inverse for a  $2 \times 2$  matrix,

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}^{-1} = \frac{1}{ad - bc} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix}.$$

Using the formula, it follows that

$$\begin{aligned} \begin{bmatrix} \Sigma_{AA} & \Sigma_{AB} \\ \Sigma_{BA} & \Sigma_{BB} \end{bmatrix} &= \begin{bmatrix} V_{AA} & V_{AB} \\ V_{BA} & V_{BB} \end{bmatrix}^{-1} \\ &= \begin{bmatrix} (V_{AA} - V_{AB}V_{BB}^{-1}V_{BA})^{-1} & -(V_{AA} - V_{AB}V_{BB}^{-1}V_{BA})^{-1}V_{AB}V_{BB}^{-1} \\ -V_{BB}^{-1}V_{BA}(V_{AA} - V_{AB}V_{BB}^{-1}V_{BA})^{-1} & (V_{BB} - V_{BA}V_{AA}^{-1}V_{AB})^{-1} \end{bmatrix} \end{aligned}$$

We immediately see that  $(V_{AA} - V_{AB}V_{BB}^{-1}V_{BA})^{-1} = \Sigma_{AA}$ , just as we expected!

### 3.3 Conditional of a joint Gaussian is Gaussian

The formal statement of this rule is:

Suppose that

$$\begin{bmatrix} x_A \\ x_B \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} \mu_A \\ \mu_B \end{bmatrix}, \begin{bmatrix} \Sigma_{AA} & \Sigma_{AB} \\ \Sigma_{BA} & \Sigma_{BB} \end{bmatrix}\right),$$

where  $x_A \in \mathbf{R}^n$ ,  $x_B \in \mathbf{R}^d$ , and the dimensions of the mean vectors and covariance matrix subblocks are chosen to match  $x_A$  and  $x_B$ . Then, the conditional densities

$$\begin{aligned} p(x_A | x_B) &= \frac{p(x_A, x_B; \mu, \Sigma)}{\int_{x_A \in \mathbf{R}^n} p(x_A, x_B; \mu, \Sigma) dx_A} \\ p(x_B | x_A) &= \frac{p(x_A, x_B; \mu, \Sigma)}{\int_{x_B \in \mathbf{R}^d} p(x_A, x_B; \mu, \Sigma) dx_B} \end{aligned}$$

are also Gaussian:

$$\begin{aligned} x_A | x_B &\sim \mathcal{N}(\mu_A + \Sigma_{AB}\Sigma_{BB}^{-1}(x_B - \mu_B), \Sigma_{AA} - \Sigma_{AB}\Sigma_{BB}^{-1}\Sigma_{BA}) \\ x_B | x_A &\sim \mathcal{N}(\mu_B + \Sigma_{BA}\Sigma_{AA}^{-1}(x_A - \mu_A), \Sigma_{BB} - \Sigma_{BA}\Sigma_{AA}^{-1}\Sigma_{AB}). \end{aligned}$$

As before, we'll just examine the conditional distribution  $x_B \mid x_A$ , and the other result will hold by symmetry. Our plan of attack will be as follows:

1. Write the form of the conditional density explicitly.
2. Rewrite the expression by partitioning the inverse covariance matrix.
3. Use a “completion-of-squares” argument.
4. Argue that the resulting density is Gaussian.

Let's see each of these steps in action.

### 3.3.1 The conditional density written explicitly

Suppose that we wanted to compute the density function of  $x_B$  given  $x_A$  directly. Then, we would need to compute

$$\begin{aligned} p(x_B \mid x_A) &= \frac{p(x_A, x_B; \mu, \Sigma)}{\int_{x_B \in \mathbf{R}^n} p(x_A, x_B; \mu, \Sigma) dx_A} \\ &= \frac{1}{Z'} \exp \left( -\frac{1}{2} \begin{bmatrix} x_A - \mu_A \\ x_B - \mu_B \end{bmatrix}^T \begin{bmatrix} \Sigma_{AA} & \Sigma_{AB} \\ \Sigma_{BA} & \Sigma_{BB} \end{bmatrix}^{-1} \begin{bmatrix} x_A - \mu_A \\ x_B - \mu_B \end{bmatrix} \right) \end{aligned}$$

where  $Z'$  is a normalization constant that we used to absorb factors not depending on  $x_B$ . Note that this time, we don't even need to compute any integrals – the value of the integral does not depend on  $x_B$ , and hence the integral can be folded into the normalization constant  $Z'$ .

### 3.3.2 Partitioning the inverse covariance matrix

As before, we reparameterize our density using the matrix  $V$ , to obtain

$$\begin{aligned} p(x_B \mid x_A) &= \frac{1}{Z'} \exp \left( -\frac{1}{2} \begin{bmatrix} x_A - \mu_A \\ x_B - \mu_B \end{bmatrix}^T \begin{bmatrix} V_{AA} & V_{AB} \\ V_{BA} & V_{BB} \end{bmatrix} \begin{bmatrix} x_A - \mu_A \\ x_B - \mu_B \end{bmatrix} \right) \\ &= \frac{1}{Z'} \exp \left( -\left[ \frac{1}{2}(x_A - \mu_A)^T V_{AA} (x_A - \mu_A) + \frac{1}{2}(x_A - \mu_A)^T V_{AB} (x_B - \mu_B) \right. \right. \\ &\quad \left. \left. + \frac{1}{2}(x_B - \mu_B)^T V_{BA} (x_A - \mu_A) + \frac{1}{2}(x_B - \mu_B)^T V_{BB} (x_B - \mu_B) \right] \right). \end{aligned}$$

### 3.3.3 Use a “completion of squares” argument

Recall that

$$\frac{1}{2} z^T A z + b^T z + c = \frac{1}{2} (z + A^{-1}b)^T A (z + A^{-1}b) + c - \frac{1}{2} b^T A^{-1}b$$

provided  $A$  is a symmetric, nonsingular matrix. As before, to apply the completion of squares in our situation above, let

$$\begin{aligned} z &= x_B - \mu_B \\ A &= V_{BB} \\ b &= V_{BA}(x_A - \mu_A) \\ c &= \frac{1}{2}(x_A - \mu_A)^T V_{AA}(x_A - \mu_A). \end{aligned}$$

Then, it follows that the expression for  $p(x_B | x_A)$  can be rewritten as

$$\begin{aligned} p(x_B | x_A) &= \frac{1}{Z'} \exp \left( - \left[ \frac{1}{2} (x_B - \mu_B + V_{BB}^{-1} V_{BA}(x_A - \mu_A))^T V_{BB} (x_B - \mu_B + V_{BB}^{-1} V_{BA}(x_A - \mu_A)) \right. \right. \\ &\quad \left. \left. + \frac{1}{2} (x_A - \mu_A)^T V_{AA}(x_A - \mu_A) - \frac{1}{2} (x_A - \mu_A)^T V_{AB} V_{BB}^{-1} V_{BA}(x_A - \mu_A) \right] \right) \end{aligned}$$

Absorbing the portion of the exponent which does not depend on  $x_B$  into the normalization constant, we have

$$p(x_B | x_A) = \frac{1}{Z''} \exp \left( - \frac{1}{2} (x_B - \mu_B + V_{BB}^{-1} V_{BA}(x_A - \mu_A))^T V_{BB} (x_B - \mu_B + V_{BB}^{-1} V_{BA}(x_A - \mu_A)) \right)$$

### 3.3.4 Arguing that resulting density is Gaussian

Looking at the last form,  $p(x_B | x_A)$  has the form of a Gaussian density with mean  $\mu_B - V_{BB}^{-1} V_{BA}(x_A - \mu_A)$  and covariance matrix  $V_{BB}^{-1}$ . As before, recall our matrix identity,

$$\begin{bmatrix} \Sigma_{AA} & \Sigma_{AB} \\ \Sigma_{BA} & \Sigma_{BB} \end{bmatrix} = \begin{bmatrix} (V_{AA} - V_{AB} V_{BB}^{-1} V_{BA})^{-1} & -(V_{AA} - V_{AB} V_{BB}^{-1} V_{BA})^{-1} V_{AB} V_{BB}^{-1} \\ -V_{BB}^{-1} V_{BA} (V_{AA} - V_{AB} V_{BB}^{-1} V_{BA})^{-1} & (V_{BB} - V_{BA} V_{AA}^{-1} V_{AB})^{-1} \end{bmatrix}.$$

From this, it follows that

$$\mu_{B|A} = \mu_B - V_{BB}^{-1} V_{BA}(x_A - \mu_A) = \mu_B + \Sigma_{BA} \Sigma_{AA}^{-1} (x_A - \mu_A).$$

Conversely, we can also apply our matrix identity to obtain:

$$\begin{bmatrix} V_{AA} & V_{AB} \\ V_{BA} & V_{BB} \end{bmatrix} = \begin{bmatrix} (\Sigma_{AA} - \Sigma_{AB} \Sigma_{BB}^{-1} \Sigma_{BA})^{-1} & -(\Sigma_{AA} - \Sigma_{AB} \Sigma_{BB}^{-1} \Sigma_{BA})^{-1} \Sigma_{AB} \Sigma_{BB}^{-1} \\ -\Sigma_{BB}^{-1} \Sigma_{BA} (\Sigma_{AA} - \Sigma_{AB} \Sigma_{BB}^{-1} \Sigma_{BA})^{-1} & (\Sigma_{BB} - \Sigma_{BA} \Sigma_{AA}^{-1} \Sigma_{AB})^{-1} \end{bmatrix},$$

from which it follows that

$$\Sigma_{B|A} = V_{BB}^{-1} = \Sigma_{BB} - \Sigma_{BA} \Sigma_{AA}^{-1} \Sigma_{AB}.$$

And, we're done!

## 4 Summary

In these notes, we used a few simple properties of multivariate Gaussians (plus a couple matrix algebra tricks) in order to argue that multivariate Gaussians satisfy a number of closure properties. In general, multivariate Gaussians are exceedingly useful representations of probability distributions because the closure properties ensure that most of the types of operations we would ever want to perform using a multivariate Gaussian can be done in closed form. Analytically, integrals involving multivariate Gaussians are often nice in practice since we can rely on known Gaussian integrals to avoid having to ever perform the integration ourselves.

## 5 Exercise

Test your understanding! Let  $A \in \mathbf{R}^{d \times d}$  be a symmetric nonsingular square matrix,  $b \in \mathbf{R}^d$ , and  $c$ . Prove that

$$\int_{x \in \mathbf{R}^d} \exp\left(-\frac{1}{2}x^T Ax - x^T b - c\right) dx = \frac{(2\pi)^{d/2}}{|A|^{1/2} \exp(c - b^T A^{-1} b)}.$$

## References

For more information on multivariate Gaussians, see

Bishop, Christopher M. *Pattern Recognition and Machine Learning*. Springer, 2006.

# 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

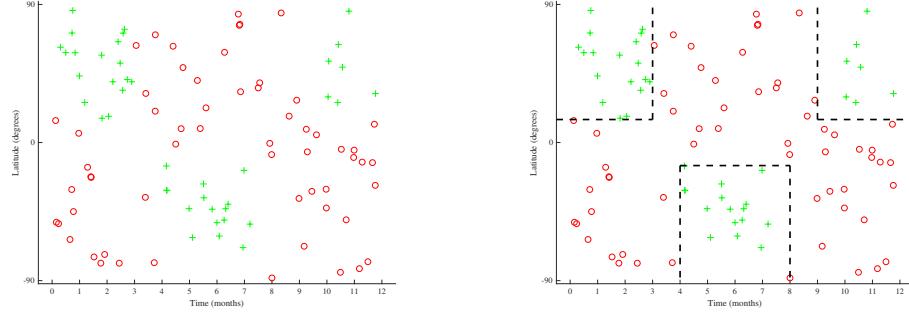
Importantly, decision trees are one of the first inherently **non-linear** machine learning techniques we will cover, as compared to methods such as vanilla SVMs or GLMs. Formally, a method is linear if for an input  $x \in \mathbb{R}^n$  (with intercept term  $x_0 = 1$ ) it only produces hypothesis 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 Canadian) 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  $-90^\circ$ ,  $0^\circ$ , and  $90^\circ$  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**)  $R_i$ :

$$\mathcal{X} = \bigcup_{i=0}^n R_i$$

s.t.       $R_i \cap R_j = \emptyset$  for  $i \neq j$

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

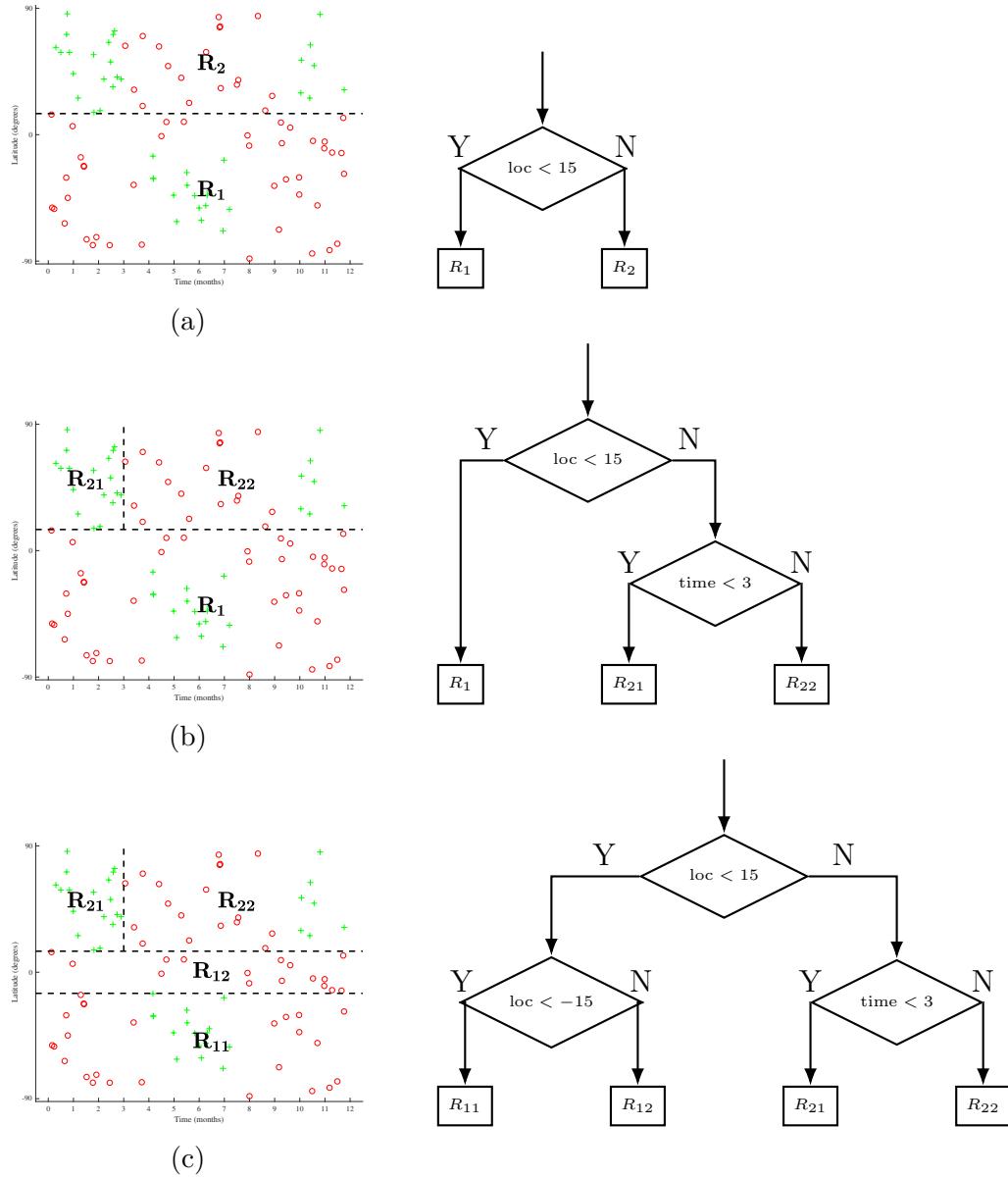
## 2 Selecting Regions

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 meet a given stop criterion (more on this later), and then predict the majority class at each leaf node.



### 3 Defining a Loss Function

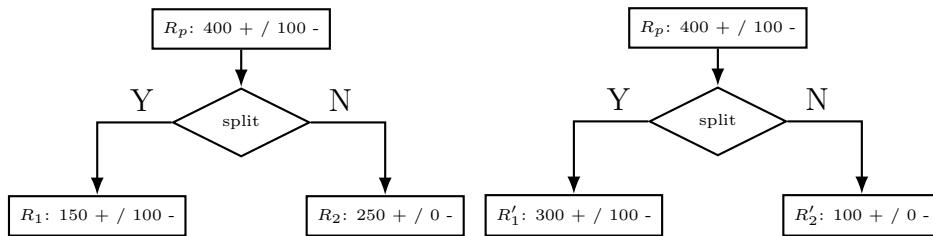
A natural question to ask at this point is how to choose our splits. To do so, it is first useful to define our loss  $L$  as a set function on a region  $R$ . Given a split of a parent  $R_p$  into two child regions  $R_1$  and  $R_2$ , we can compute the loss of the parent  $L(R_p)$  as well as the 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**  $L_{\text{misclass}}$ . 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:

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

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.

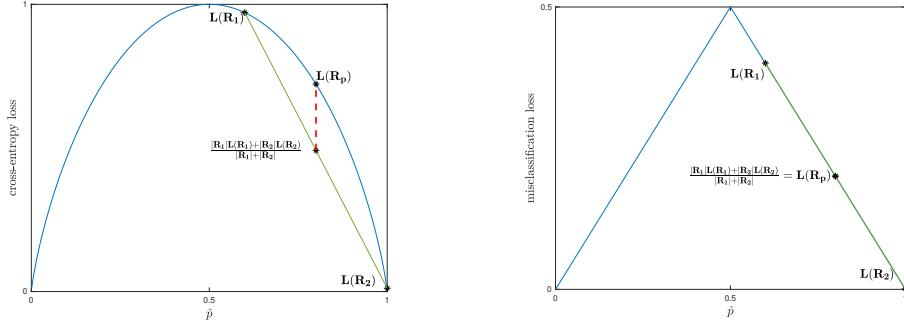
We therefore are interested in defining a more sensitive loss. While several have been proposed, we will focus here on the **cross-entropy** loss  $L_{cross}$ :

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

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  $R_i$ :

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



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:

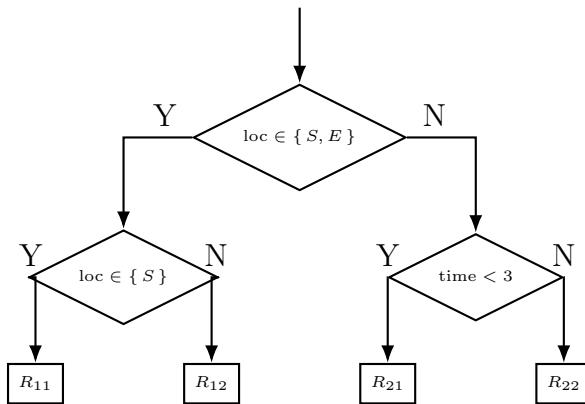
$$L_{\text{squared}}(R) = \frac{\sum_{i \in R} (y_i - \hat{y})^2}{|R|}$$

## 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.  $\text{loc} \in \{N, S, E\}$ )). 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 selection 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" growing 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

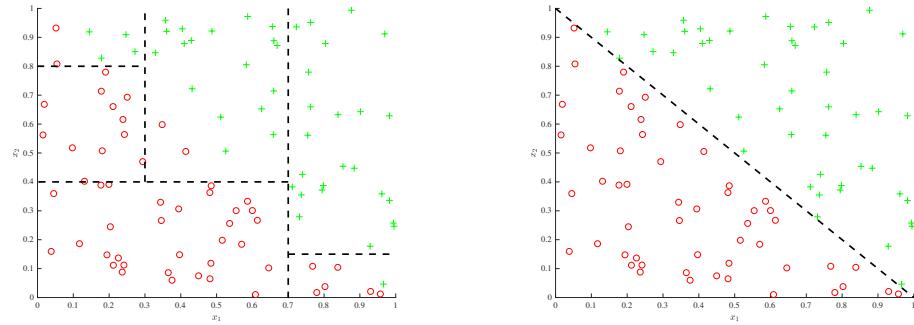
We briefly turn to considering the runtime of decision trees. For ease of analysis, we will consider binary classification with  $n$  examples,  $f$  features, and a tree of depth  $d$ . At test time, for a data point we traverse the tree

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

Andrew Ng

# 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,  $f : x \rightarrow y$ . 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:  $x$  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

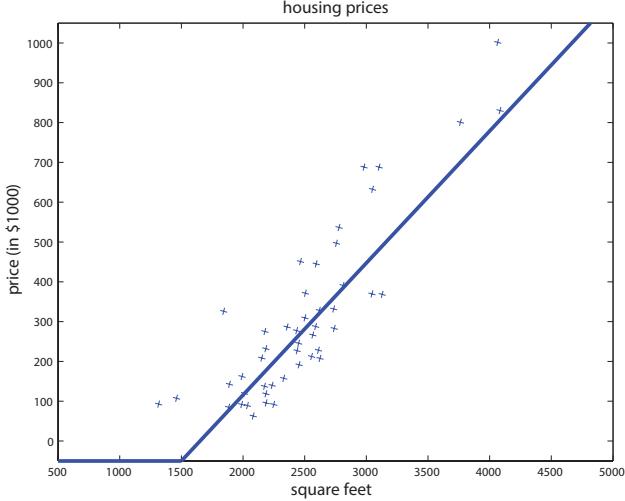


Figure 1: Housing prices with a “kink” in the graph.

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.

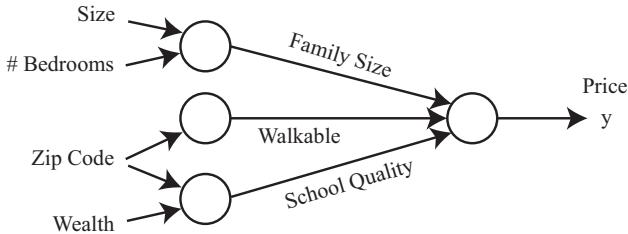


Figure 2: Diagram of a small neural network for predicting housing prices.

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:

$$x_1 = a_1^{[0]} \quad (1.1)$$

$$x_2 = a_2^{[0]} \quad (1.2)$$

$$x_3 = a_3^{[0]} \quad (1.3)$$

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

activation of the  $j^{th}$  unit in layer  $\ell$ . 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:

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

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

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

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

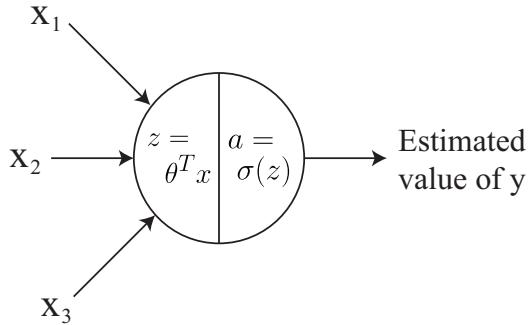


Figure 3: Logistic regression as a single neuron.

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

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

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:

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

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

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

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

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

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  $z_1, \dots, z_4$  and  $a_1, \dots, a_4$ .

$$z_1^{[1]} = W_1^{[1]T}x + b_1^{[1]} \quad \text{and} \quad a_1^{[1]} = g(z_1^{[1]}) \quad (2.1)$$

$$\vdots \qquad \vdots \qquad \vdots \quad (2.2)$$

$$z_4^{[1]} = W_4^{[1]T}x + b_4^{[1]} \quad \text{and} \quad a_4^{[1]} = g(z_4^{[1]}) \quad (2.3)$$

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  $z_1, \dots, z_4$  without a for loop. Using our matrix algebra, we can compute the activations:

$$\underbrace{\begin{bmatrix} z_1^{[1]} \\ \vdots \\ z_4^{[1]} \end{bmatrix}}_{z^{[1]} \in \mathbb{R}^{4 \times 1}} = \underbrace{\begin{bmatrix} -W_1^{[1]T} - \\ -W_2^{[1]T} - \\ \vdots \\ -W_4^{[1]T} - \end{bmatrix}}_{W^{[1]} \in \mathbb{R}^{4 \times 3}} \underbrace{x}_{x \in \mathbb{R}^{3 \times 1}} + \underbrace{\begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ \vdots \\ b_4^{[1]} \end{bmatrix}}_{b^{[1]} \in \mathbb{R}^{4 \times 1}} \quad (2.4)$$

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:

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

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:

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

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):

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

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:

$$z^{[2]} = W^{[2]}a^{[1]} \quad (2.8)$$

$$= W^{[2]}g(z^{[1]}) \quad \text{by definition} \quad (2.9)$$

$$= W^{[2]}z^{[1]} \quad \text{since } g(z) = z \quad (2.10)$$

$$= W^{[2]}W^{[1]}x \quad \text{from Equation (2.4)} \quad (2.11)$$

$$= \tilde{W}x \quad \text{where } \tilde{W} = W^{[2]}W^{[1]} \quad (2.12)$$

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:

$$z^{[1](1)} = W^{[1]}x^{(1)} + b^{[1]}$$

$$z^{[1](2)} = W^{[1]}x^{(2)} + b^{[1]}$$

$$z^{[1](3)} = W^{[1]}x^{(3)} + b^{[1]}$$

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:

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

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

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

Putting it together: Suppose we have a training set  $(x^{(1)}, y^{(1)}), \dots, (x^{(m)}, y^{(m)})$  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  $a^{[2](i)}$ . Using the logistic regression log likelihood:

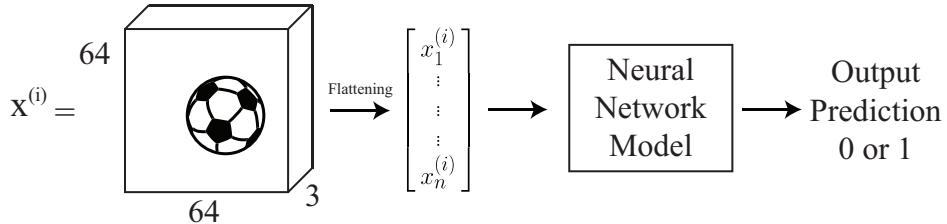
$$\sum_{i=1}^m \left( y^{(i)} \log a^{[2](i)} + (1 - y^{(i)}) \log(1 - a^{[2](i)}) \right) \quad (2.15)$$

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.

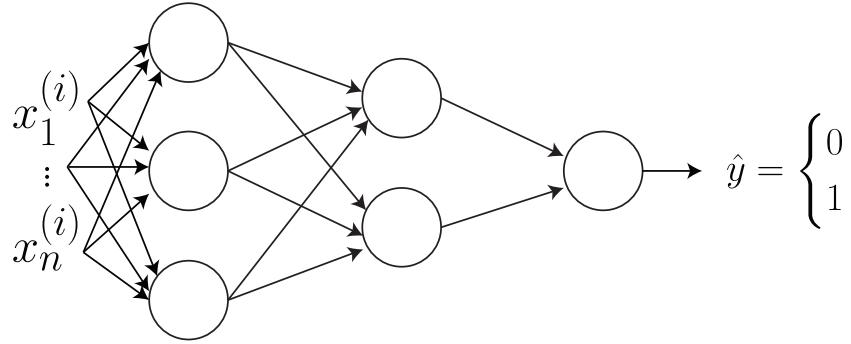
Aside: Images can be represented as a matrix with number of elements equal to the number of pixels. However, color images are digitally represented as a volume (i.e., three-channels; or three matrices stacked on each other). The number three is used because colors are represented as red-green-blue (RGB) 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  $x^{(1)}, \dots, x_n^{(i)}$ . 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.

$$z^{[1]} = W^{[1]}x^{(i)} + b^{[1]} \quad (3.1)$$

$$a^{[1]} = g(z^{[1]}) \quad (3.2)$$

$$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]} \quad (3.3)$$

$$a^{[2]} = g(z^{[2]}) \quad (3.4)$$

$$z^{[3]} = W^{[3]}a^{[2]} + b^{[3]} \quad (3.5)$$

$$\hat{y}^{(i)} = a^{[3]} = g(z^{[3]}) \quad (3.6)$$

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

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

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

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

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:

$$\mathcal{L}(\hat{y}, y) = -\left[ (1 - y) \log(1 - \hat{y}) + y \log \hat{y} \right] \quad (3.9)$$

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  $\ell$ , we update them:

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

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

where  $\alpha$  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:

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

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:

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

where  $n^{[\ell]}$  is the number of neurons in layer  $\ell$ . 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.

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

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

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

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

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

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

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

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

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  $\hat{y} = a^{[3]}$ .

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

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  $\partial a^{[3]}/\partial a^{[3]}$ :

$$\frac{\partial\mathcal{L}}{\partial W^{[2]}} = \frac{\partial\mathcal{L}}{\partial a^{[3]}} \frac{\partial a^{[3]}}{\partial z^{[3]}} \frac{\partial z^{[3]}}{?} \frac{\partial ?}{\partial W^{[2]}} \quad (3.23)$$

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

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

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.

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

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

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

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

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

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:

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

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:

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

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:

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

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  $\ell$ , the update rule is defined as:

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

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:

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

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  $\ell$ , the update rule is as follows:

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

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  $\beta$ . 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:

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

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

$$= J + \frac{\lambda}{2} W^T W \quad (3.36)$$

where  $J$  is the standard cost function from before,  $\lambda$  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:

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

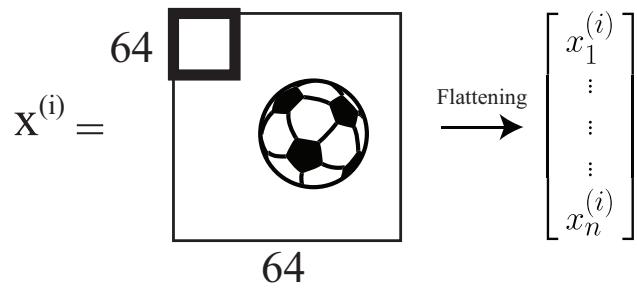
$$= (1 - \alpha\lambda)W - \alpha \frac{\partial J}{\partial W} \quad (3.38)$$

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  $\theta = (\theta_1, \dots, \theta_n)$  must have the same number of elements as the input vector  $x = (x_1, \dots, x_n)$ . In our image soccer ball example, this means  $\theta_1$  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  $\theta_1$  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  $\theta$  is no longer a vector but instead is a matrix. For our soccer ball example, suppose  $\theta = \mathbb{R}^{4 \times 4}$ . For simplicity, we show the image as  $64 \times 64$  but recall it is actually three-

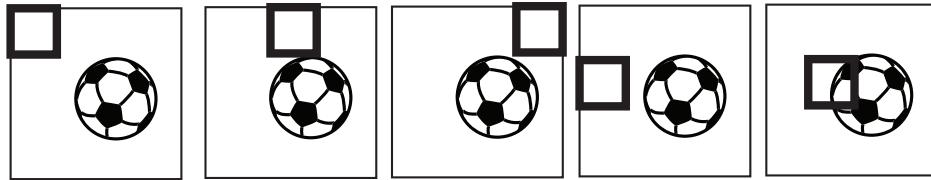


dimensional and contains 3 channels. We now take our matrix of parameters  $\theta$  and slide it over the image. This is shown above by the thick square in the upper left of the image. To compute the activation  $a$ , we compute the element-wise product between  $\theta$  and  $x_{1:4,1:4}$ , where the subscripts for  $x$

indicate we are taking the top left  $4 \times 4$  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:

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

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  $\theta$  have “seen” all pixels of the image:  $\theta_1$  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  $\ell = 1, 2, \dots, N$ , 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  $l$ th 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.<sup>1</sup>  
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  $\sigma$ ) 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  $\ell = N - 1, N - 2, \dots, 1$ , we have

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

3. Finally, we can compute the gradients for layer  $\ell$  as

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

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  $\sigma$ ) 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>

---

<sup>1</sup>These notes are closely adapted from:

<http://ufldl.stanford.edu/tutorial/supervised/MultiLayerNeuralNetworks/>  
Scribe: Ziang Xie

# Bias-Variance Analysis: Theory and Practice

Anand Avati

## 1 Introduction

In this set of notes, we will explore the fundamental Bias-Variance tradeoff in Statistics and Machine Learning under the squared error loss. The concepts of Bias and Variance are slightly different in the contexts of Statistics vs Machine Learning, though the two are closely related in spirit. We will first start with the classical notions from Statistics, using Linear Regression with  $L_2$ -regularization as a case study. The simplicity of Linear Regression allows us to derive closed form expressions for the Bias and Variance terms and appreciate the tradeoff better. Then we will study the notion of Bias and Variance and their decomposition in the context of Machine Learning (prediction), and see the connections to the classical notions using  $L_2$ -regularized Linear Regression as an example.

Throughout this document we will use the following notation. We are given an i.i.d. data set  $S = \{(x^{(i)}, y^{(i)})\}_{i=1}^n$  that was generated from some data generating probability distribution having some unknown (constant) parameter  $\theta^* \in \mathbb{R}^d$ . Here  $x^{(i)} \in \mathbb{R}^d$  and  $y^{(i)} \in \mathbb{R}$ . For notational convenience, let  $X \in \mathbb{R}^{n \times d}$  denote the design matrix, and  $\vec{y} \in \mathbb{R}^n$  the vector of labels. In the case of regression  $X$  is considered given (constant).

## 2 Bias and Variance in Statistical Inference

We start with the classical setting of statistical inference. Our goal in statistical inference is to construct an estimator for the unknown parameter  $\theta^*$  given the observed data set  $S$ .

Let us indicate our estimator as  $\hat{\theta}_n$ , where  $n$  is the size of the dataset used to fit the model. For example, in the case of Linear Regression,  $\hat{\theta}_n = (X^T X)^{-1} X^T \vec{y}$ . Note that  $\hat{\theta}_n$  is a random variable even though  $\theta^*$  was not. This is because  $\hat{\theta}_n$  is a (deterministic) function of the noisy data set  $S$ , where noise is typically in the labels  $\vec{y}$ . It is worth noting that the randomness in  $\hat{\theta}_n$  therefore indirectly depends on  $\theta^*$ , due to this noise. The distribution of  $\hat{\theta}_n$  is commonly called the *Sampling distribution*. The Bias and Variance of the estimator  $\hat{\theta}_n$  are just the (centered) first and second moments of its sampling distribution.

We call  $\text{Bias}(\hat{\theta}_n) \equiv \mathbb{E}[\hat{\theta}_n - \theta^*]$  the *Bias* of the estimator  $\hat{\theta}_n$ . The estimator  $\hat{\theta}_n$  is called *Unbiased* if  $\mathbb{E}[\hat{\theta}_n - \theta^*] = 0$  (i.e.  $\mathbb{E}[\hat{\theta}_n] = \theta^*$ ) for all values of  $\theta^*$ .

Similarly, we call  $\text{Var}(\hat{\theta}_n) \equiv \text{Cov}[\hat{\theta}_n]$  the *Variance* of the estimator. Note that, unlike Bias, the Variance of the estimator does not directly depend on the true parameter  $\theta^*$ .

The Bias and Variance of an estimator are not necessarily directly related (just as how the first and second moment of any distribution are not necessarily related). It is possible to have estimators that have high or low bias and have either high or low variance. Under the squared error, the Bias and Variance of an estimator are related as:

$$\begin{aligned}\text{MSE}(\hat{\theta}_n) &= \mathbb{E} [\|\hat{\theta}_n - \theta^*\|^2] \\ &= \mathbb{E} [\|\hat{\theta}_n - \mathbb{E}[\hat{\theta}_n] + \mathbb{E}[\hat{\theta}_n] - \theta^*\|^2] \\ &= \mathbb{E} \left[ \|\hat{\theta}_n - \mathbb{E}[\hat{\theta}_n]\|^2 + \underbrace{\|\mathbb{E}[\hat{\theta}_n] - \theta^*\|^2}_{\text{Constant}} + 2(\hat{\theta}_n - \mathbb{E}[\hat{\theta}_n])^T (\mathbb{E}[\hat{\theta}_n] - \theta^*) \right] \\ &= \mathbb{E} [\|\hat{\theta}_n - \mathbb{E}[\hat{\theta}_n]\|^2] + \|\mathbb{E}[\hat{\theta}_n] - \theta^*\|^2 \\ &= \mathbb{E} \left[ \text{tr} [(\hat{\theta}_n - \mathbb{E}[\hat{\theta}_n])(\hat{\theta}_n - \mathbb{E}[\hat{\theta}_n])^T] \right] + \|\mathbb{E}[\hat{\theta}_n] - \theta^*\|^2 \\ &= \text{tr} [\text{Var}(\hat{\theta}_n)] + \|\text{Bias}(\hat{\theta}_n)\|^2.\end{aligned}$$

It is quite often the case that techniques employed to reduce Variance results in an increase in Bias, and vice versa. This phenomenon is called the *Bias Variance Tradeoff*. Balancing the two evils (Bias and Variance) in an optimal way is at the heart of successful model development. Now we will do a case study of Linear Regression with  $L_2$ -regularization, where this trade-off can be easily formalized.

## 2.1 Bias Variance Tradeoff in Linear Regression with $L_2$ -regularization

Recall that in Linear Regression we make the assumption  $y^{(i)} = \theta^*{}^T x^{(i)} + \epsilon^{(i)}$  where each  $\epsilon^{(i)} \sim \mathcal{N}(0, \tau^2)$  i.i.d.. We assume  $X$  is given, and hence constant. For notational simplicity let  $\vec{\epsilon} \in \mathbb{R}^n \sim \mathcal{N}(\vec{0}, \tau^2 I)$  where  $\vec{\epsilon}_i = \epsilon^{(i)}$ . So,  $\vec{y} = X\theta^* + \vec{\epsilon}$ . Recall that Linear Regression with  $L_2$ -regularization (with regularization parameter  $\lambda > 0$ ) minimizes the cost function

$$J(\theta) = \frac{\lambda}{2} \|\theta\|_2^2 + \frac{1}{2} \sum_{i=1}^n (y^{(i)} - \theta^T x^{(i)})^2,$$

and enjoys a closed form solution

$$\begin{aligned} \hat{\theta}_n &= \arg \min_{\theta \in \mathbb{R}^d} J(\theta) \\ &= \arg \min_{\theta \in \mathbb{R}^d} \left[ \frac{\lambda}{2} \|\theta\|_2^2 + \frac{1}{2} \sum_{i=1}^n (y^{(i)} - \theta^T x^{(i)})^2 \right] \\ &= \arg \min_{\theta \in \mathbb{R}^d} \left[ \frac{\lambda}{2} \|\theta\|_2^2 + \frac{1}{2} \|X\theta - \vec{y}\|_2^2 \right] \\ &= (X^T X + \lambda I)^{-1} X^T \vec{y}. \end{aligned}$$

Consider the eigendecomposition of the symmetric Positive Semi Definite (PSD) matrix  $X^T X$ :

$$X^T X = U \underbrace{\begin{bmatrix} \sigma_1^2 & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & \sigma_d^2 \end{bmatrix}}_{\text{diag}(\sigma_1^2, \dots, \sigma_d^2)} U^T,$$

where  $U^T U = UU^T = I$ , and  $\{\sigma_1^2, \sigma_2^2, \dots, \sigma_d^2\}$  are the eigenvalues, where some of the  $\sigma_i^2$  could be 0. However, even when  $X$  (and hence  $X^T X$ ) is not full rank,  $(X^T X + \lambda I)$  is always symmetric and Positive Definite (PD) since we are adding  $\lambda > 0$  to all the diagonal elements of  $X^T X$ :

$$X^T X + \lambda I = U \begin{bmatrix} \sigma_1^2 + \lambda & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & \sigma_d^2 + \lambda \end{bmatrix} U^T.$$

This implies

$$(X^T X + \lambda I)^{-1} = U \begin{bmatrix} \frac{1}{\sigma_1^2 + \lambda} & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & \frac{1}{\sigma_d^2 + \lambda} \end{bmatrix} U^T.$$

Therefore  $\hat{\theta}_n$  always exists and is unique. Now we analyze the Bias and Variance of  $\hat{\theta}_n$ . We start with the expression for the estimator

$$\begin{aligned} \hat{\theta}_n &= (X^T X + \lambda I)^{-1} X^T \vec{y} \\ &= (X^T X + \lambda I)^{-1} X^T (X \theta^* + \vec{\epsilon}) \\ &= [(X^T X + \lambda I)^{-1} X^T X] \theta^* + [(X^T X + \lambda I)^{-1} X^T] \vec{\epsilon} \end{aligned}$$

To compute the Bias of this model, we take the expectation of the above and observe that (remember,  $X$  is considered constant in regression):

$$\begin{aligned} \mathbb{E}[\hat{\theta}_n] &= \mathbb{E}\left[[(X^T X + \lambda I)^{-1} X^T X] \theta^* + [(X^T X + \lambda I)^{-1} X^T] \vec{\epsilon}\right] \\ &= [(X^T X + \lambda I)^{-1} X^T X] \theta^* + [(X^T X + \lambda I)^{-1} X^T] \mathbb{E}[\vec{\epsilon}] \\ &= [(X^T X + \lambda I)^{-1} X^T X] \theta^* + [(X^T X + \lambda I)^{-1} X^T] \vec{0} \\ &= [(X^T X + \lambda I)^{-1} X^T X] \theta^* \\ &= U \begin{bmatrix} \frac{1}{\sigma_1^2 + \lambda} & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & \frac{1}{\sigma_d^2 + \lambda} \end{bmatrix} U^T X^T X \theta^* \\ &= U \begin{bmatrix} \frac{1}{\sigma_1^2 + \lambda} & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & \frac{1}{\sigma_d^2 + \lambda} \end{bmatrix} U^T U \begin{bmatrix} \sigma_1^2 & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & \sigma_d^2 \end{bmatrix} U^T \theta^* \\ &= U \begin{bmatrix} \frac{1}{\sigma_1^2 + \lambda} & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & \frac{1}{\sigma_d^2 + \lambda} \end{bmatrix} \begin{bmatrix} \sigma_1^2 & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & \sigma_d^2 \end{bmatrix} U^T \theta^* \\ &= U \begin{bmatrix} \frac{\sigma_1^2}{\sigma_1^2 + \lambda} & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & \frac{\sigma_d^2}{\sigma_d^2 + \lambda} \end{bmatrix} U^T \theta^*. \end{aligned}$$

From the above, we can make a few observations. First, when  $\lambda = 0$ , we see that  $\mathbb{E}[\hat{\theta}_n] = \theta^*$ . This implies that standard linear regression estimator (without regularization) is Unbiased. Second, the above expression is essentially a “shrunk”  $\theta^*$  because all the eigenvalues of the matrix are less than one. In fact, the more regularization we add (i.e. larger  $\lambda$ ), the smaller the eigenvalues will be, and hence the stronger the “shrinkage” towards 0. This implies that the estimator  $\hat{\theta}_n$  of  $L_2$ -regularized Linear Regression is Biased (towards 0 in this case).

Though we paid the price of adding regularization in the form of having a Biased estimator, we do however gain something in return: reduced variance. In order to analyze the variance of the estimator  $\hat{\theta}_n$ , first recall the following property of multivariate Gaussians:

$$\text{If } \vec{\epsilon} \sim \mathcal{N}(\vec{0}, \tau^2 I), \text{ then } A\vec{\epsilon} \sim \mathcal{N}(\vec{0}, A(\tau^2 I)A^T).$$

This gives us (again, remember  $X$  is given, and hence constant in regression):

$$\begin{aligned} \text{Cov}[\hat{\theta}_n] &= \text{Cov}\left[\left(X^T X + \lambda I\right)^{-1} X^T \vec{y}\right] \\ &= \text{Cov}\left[\left(X^T X + \lambda I\right)^{-1} X^T (X\theta^* + \vec{\epsilon})\right] \\ &= \text{Cov}\left[\underbrace{\left(X^T X + \lambda I\right)^{-1} X^T X\theta^*}_{\text{Constant}} + \underbrace{\left(\left(X^T X + \lambda I\right)^{-1} X^T\right)}_{\text{Constant}} \vec{\epsilon}\right] \\ &= \text{Cov}\left[\left(\left(X^T X + \lambda I\right)^{-1} X^T\right) \vec{\epsilon}\right] \\ &= \left[\left(X^T X + \lambda I\right)^{-1} X^T\right] \text{Cov}[\vec{\epsilon}] \left[\left(X^T X + \lambda I\right)^{-1} X^T\right]^T \quad (\text{using above property}) \\ &= \left[\left(X^T X + \lambda I\right)^{-1} X^T\right] \tau^2 I \left[X \left(X^T X + \lambda I\right)^{-1}\right] \\ &= \tau^2 \left(X^T X + \lambda I\right)^{-1} \left(X^T X\right) \left(X^T X + \lambda I\right)^{-1} \\ &= \tau^2 U \begin{bmatrix} \frac{1}{\sigma_1^2 + \lambda} & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & \frac{1}{\sigma_d^2 + \lambda} \end{bmatrix} U^T U \begin{bmatrix} \sigma_1^2 & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & \sigma_d^2 \end{bmatrix} U^T U \begin{bmatrix} \frac{1}{\sigma_1^2 + \lambda} & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & \frac{1}{\sigma_d^2 + \lambda} \end{bmatrix} U^T \\ &= U \begin{bmatrix} \frac{\tau^2 \sigma_1^2}{(\sigma_1^2 + \lambda)^2} & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & \frac{\tau^2 \sigma_d^2}{(\sigma_d^2 + \lambda)^2} \end{bmatrix} U^T. \end{aligned}$$

From the above expression we observe that as we add more regularization (i.e. larger  $\lambda$ ), the smaller the spectrum (i.e. all the eigenvalues) of the covariance of the estimator  $\hat{\theta}_n$ , and hence smaller  $\text{tr} [\text{Var}(\hat{\theta}_n)]$ .

Thus we clearly see the Bias Variance trade-off as a function of  $\lambda$ . The larger the value of  $\lambda$ , the higher the Bias (undesirable) but also smaller the Variance (desirable) of  $\hat{\theta}_n$ , and vice versa. There exists a sweet spot for  $\lambda$  that minimizes the sum of the two evils, and finding that sweet spot is better explained in the context of prediction, which is the next section.

### 3 Bias and Variance in Prediction

In a prediction (Supervised Machine Learning) setting, our goals are different from statistical inference. Instead of constructing an estimator  $\hat{\theta}_n$  for the unknown parameter, we wish to learn a function  $f$  that can predict  $y$  given  $x$  well (with respect to some loss function). As before, we are given a training set  $S = \{(x^{(i)}, y^{(i)})\}_{i=1}^n$ . We make the assumption that  $y = f(x) + \epsilon$ , where  $\epsilon$  satisfies  $\mathbb{E}[\epsilon] = 0$  and  $\mathbb{V}[\epsilon] = \tau^2$  ( $\epsilon$  is not necessarily Gaussian). Further, we *define (not assume)* the “true”  $f$  as

$$f(x') \equiv \mathbb{E}[y|x=x'].$$

Our task now is to construct a hypothesis  $\hat{f}_n$  given a fixed size training set  $S$  that mimics  $f$  well on all future unseen examples. In other words,  $\hat{f}_n$  needs to have good *generalization error*. We will only consider the case where the generalization error is the expected squared error loss on an unseen example.

Suppose  $\hat{f}_n$  is obtained with some (unspecified) training process over  $S$ . As before, note that  $\hat{f}_n$  is random, and the randomness comes due to the  $\epsilon^{(i)}$ 's embedded in the training set examples. Consider a new unseen example pair  $(y_*, x_*)$  and the corresponding generalization error, where the expectation is over the randomness in  $\epsilon$  embedded in the test example, and in  $\hat{f}_n$ :

$$\begin{aligned} \text{MSE}(\hat{f}_n) &= \mathbb{E} \left[ \left( y_* - \hat{f}_n(x_*) \right)^2 \right] \\ &= \mathbb{E} \left[ \left( \epsilon + f(x_*) - \hat{f}_n(x_*) \right)^2 \right] \end{aligned}$$

$$\begin{aligned}
&= \mathbb{E} [\epsilon^2] + \mathbb{E} \left[ (f(x_*) - \hat{f}_n(x_*))^2 \right] + \mathbb{E} [2\epsilon(f(x_*) - \hat{f}_n(x_*))] \\
&= \mathbb{E} [\epsilon^2] + \mathbb{E} \left[ (f(x_*) - \hat{f}_n(x_*))^2 \right] + \underbrace{\mathbb{E} [\epsilon]}_{=0} \mathbb{E} [2(f(x_*) - \hat{f}_n(x_*))] \quad (\text{i.i.d. } \epsilon) \\
&= \mathbb{E} [\epsilon^2] + \mathbb{E} \left[ (f(x_*) - \hat{f}_n(x_*))^2 \right] \\
&= \mathbb{E} [\epsilon^2] + \mathbb{E} \left[ f(x_*) - \hat{f}_n(x_*) \right]^2 + \mathbb{V} [f(x_*) - \hat{f}_n(x_*)] \quad (\mathbb{E}[X^2] = \mathbb{V}[X] + \mathbb{E}[X]^2) \\
&= \underbrace{\tau^2}_{\text{Irreducible error}} + \underbrace{\mathbb{E} \left[ \hat{f}_n(x_*) - f(x_*) \right]^2}_{\text{Bias}^2} + \underbrace{\mathbb{V} [\hat{f}_n(x_*)]}_{\text{Variance}} \quad (\mathbb{V}[a - X] = \mathbb{V}[X])
\end{aligned}$$

The above decomposition suggests similarities with the statistical inference setting. The Bias and Variance are, as before, just the (centered) first and second moments of  $\hat{f}_n$  (skipping  $x_*$  in the notation). Bias of  $\hat{f}_n$  at input  $x_*$  is defined as  $\mathbb{E}[\hat{f}_n - f]$ , and Variance is  $\mathbb{V}[\hat{f}_n]$ . Such a clean decomposition into Bias and Variance terms exists only for the squared error loss. Proposals have been made for more general losses, though none are widely accepted.

### 3.1 Prediction with $L_2$ -regularized Linear Regression

Now let us tie all this back to the case of  $L_2$ -regularized Linear Regression. Let us assume  $f(x) = \theta^{*T}x$  where  $\theta^*$  is the true unknown parameter. Now  $\hat{f}_n(x) = \hat{\theta}_n^T x$  where  $\hat{\theta}_n$  is the  $L_2$ -regularized Linear Regression estimator. We can see the relation between the Bias and Variance terms of prediction and of inference as follows:

$$\begin{aligned}
\text{Bias}(\hat{f}_n) &= \mathbb{E}[\hat{f}_n(x) - f(x)] \\
&= \mathbb{E}[\hat{\theta}_n^T x - \theta^{*T} x] \\
&= \mathbb{E}[\hat{\theta}_n - \theta^*]^T x \\
&= \text{Bias}(\hat{\theta}_n)^T x. \\
(\text{and similarly}) \quad \text{Var}(\hat{f}_n) &= x^T \left[ \text{Var}(\hat{\theta}_n) \right] x.
\end{aligned}$$

The irreducible error appears only in the prediction setting, as it is an artifact of the noise in the *test example* (there is no such test example in the inference setting). In other words, the noise in the training data contributes to the

Variance term, and the noise in the test example manifests itself as the irreducible error term.

In order to minimize the generalization error, we need to reduce one or more of the decomposed components. There is nothing we can do to reduce irreducible error, since it is just noise in the data (i.e., the same  $x$  could have different  $y$  values in different examples). Thus we are left with balancing the Bias and Variance terms. In the case of  $L_2$ -regularized Linear Regression, we could consider adjusting the  $\lambda$  value. In the inference setting, we saw that increasing  $\lambda$  reduces the Variance but increases the Bias. This tradeoff directly translates into the prediction setting as well, based on the above relations. Back then it was not clear what might be a good sweet spot for setting the  $\lambda$  value. However in the prediction setting, there is an obvious answer: choose  $\lambda$  to be the value that minimizes the squared error (generalization error) in cross-validation.

## 4 Bias and Variance in practice

To wrap things up, we can relate the Bias Variance decomposition to the commonly used terms *overfitting* and *underfitting* in the following informal way:

- *Overfitting* relates to having a *High Variance* model or estimator. To fight overfitting, we need to focus on reducing the Variance of the estimator, such as: increase regularization, obtain larger data set, decrease number of features, use a smaller model, etc.
- *Underfitting* relates to having a *High Bias* model or estimator. To fight underfitting, we need to focus on reducing the Bias in the estimator, such as: decrease regularization, use more features, use a larger model, etc.

The first step in improving generalization error is to characterize which component in the decomposition has the highest contribution, and go after that component. Unfortunately there is no theoretically sound yet tractable way of calculating the breakdown. However there are certain heuristics that are extremely useful. Loosely speaking:

- Training error can be treated as the amount of Bias in the model or estimator. If the model is unable to fit the training data itself well, then it is likely that the model has High Bias. This is the underfitting regime.

- Gap between cross-validation error and Training error can be treated as the Variance of the model or the estimator. If the Training error is low but the Cross Validation error is high, it is very likely that model has High Variance. This is the overfitting regime.

We should ***always*** analyze the model performance by looking at the training error and cross-validation error *simultaneously*. This is the only tractable (albeit heuristic) way to obtain an estimate of the Bias and Variance components. Only then should we take steps that are targeted towards addressing either Bias or Variance purposefully.

Steps taken to fight overfitting (i.e. fight High Variance) generally do not necessarily help fight underfitting (i.e. High Bias). For example, it is futile to spend time and resources in obtaining more data (technique to fight High Variance) when the training error itself is high (symptom of High Bias).

Similarly steps taken to fight underfitting (i.e. fight High Bias) generally do not necessarily help fight overfitting (i.e. High Variance). For example, it is futile to switch to a larger neural network (technique to fight High Bias) when the gap between cross-validation error and training error is high (symptom of High Variance).

Many times steps taken to fight one (either High Bias or High Variance) can end up worsening the other. This is essentially how the Bias Variance trade-off is encountered in practice.

# CS229 Bias-Variance and Error Analysis

Yoann Le Calonnec

October 2, 2017

## 1 The Bias-Variance Tradeoff

Assume you are given a well fitted machine learning model  $\hat{f}$  that you want to apply on some test dataset. For instance, the model could be a linear regression whose parameters were computed using some training set different from your test set. For each point  $x$  in your test set, you want to predict the associated target  $y \in \mathbb{R}$ , and compute the mean squared error (MSE)

$$\mathbb{E}_{(x,y) \sim \text{test set}} |\hat{f}(x) - y|^2$$

You now realize that this MSE is too high, and try to find an explanation to this result:

- Overfitting: the model is too closely related to the examples in the training set and doesn't generalize well to other examples.
- Underfitting: the model didn't gather enough information from the training set, and doesn't capture the link between the features  $x$  and the target  $y$ .
- The data is simply noisy, that is the model is neither overfitting or underfitting, and the high MSE is simply due to the amount of noise in the dataset.

Our intuition can be formalized by the **Bias-Variance tradeoff**.

Assume that the points in your training/test set are all taken from a similar distribution, with

$$y_i = f(x_i) + \epsilon_i, \quad \text{where the noise } \epsilon_i \text{ satisfies } \mathbb{E}(\epsilon_i) = 0, \text{Var}(\epsilon_i) = \sigma^2$$

and your goal is to compute  $f$ . By looking at your training set, you obtain an estimate  $\hat{f}$ . Now use this estimate with your test set, meaning that for each example  $j$  in the test set, your prediction for  $y_j = f(x_j) + \epsilon_j$  is  $\hat{f}(x_j)$ . Here,  $x_j$  is a fixed real number (or vector if the feature space is multi-dimensional) thus  $f(x_j)$  is fixed, and  $\epsilon_j$  is a real random variable with mean 0 and variance  $\sigma^2$ . The crucial observation is that  $\hat{f}(x_j)$  is random since it depends on the values  $\epsilon_i$  from the training set. That's why talking about the bias  $\mathbb{E}(\hat{f}(x) - f(x))$  and the variance of  $\hat{f}$  makes sense.

We can now compute our MSE on the test set by computing the following expectation with respect to the possible training sets (since  $\hat{f}$  is a random variable function of the choice of the training set)

$$\begin{aligned}
\text{Test MSE} &= \mathbb{E} \left( (y - \hat{f}(x))^2 \right) \\
&= \mathbb{E} \left( (\epsilon + f(x) - \hat{f}(x))^2 \right) \\
&= \mathbb{E}(\epsilon^2) + \mathbb{E} \left( (f(x) - \hat{f}(x))^2 \right) \\
&= \sigma^2 + \left( \mathbb{E}(f(x) - \hat{f}(x)) \right)^2 + \text{Var} \left( f(x) - \hat{f}(x) \right) \\
&= \sigma^2 + \left( \text{Bias } \hat{f}(x) \right)^2 + \text{Var} \left( \hat{f}(x) \right)
\end{aligned}$$

There is nothing we can do about the first term  $\sigma^2$  as we can not predict the noise  $\epsilon$  by definition. The bias term is due to underfitting, meaning that on average,  $\hat{f}$  does not predict  $f$ . The last term is closely related to overfitting, the prediction  $\hat{f}$  is too close from the values  $y_{\text{train}}$  and varies a lot with the choice of our training set.

To sum up, we can understand our MSE as follows

$$\begin{array}{lll}
\text{High Bias} & \longleftrightarrow & \text{Underfitting} \\
\text{High Variance} & \longleftrightarrow & \text{Overfitting} \\
\text{Large } \sigma^2 & \longleftrightarrow & \text{Noisy data}
\end{array}$$

Hence, when analyzing the performance of a machine learning algorithm, we must always ask ourselves how to reduce the bias without increasing the variance, and respectively how to reduce the variance without increasing the bias. Most of the time, reducing one will increase the other, and there is a tradeoff between bias and variance.

## 2 Error Analysis

Even though understanding whether our poor test error is due to high bias or high variance is important, knowing which parts of the machine learning algorithm lead to this error or score is crucial.

Consider the machine learning pipeline on figure 1.

The algorithms is divided into several steps

1. The inputs are taken from a camera image
2. Preprocessing to remove the background on the image. For instance, if the image are taken from a security camera, the background is always the same, and we could remove it easily by keeping the pixels that changed on the image.
3. Detect the position of the face.
4. Detect the eyes - Detect the nose - Detect the mouth

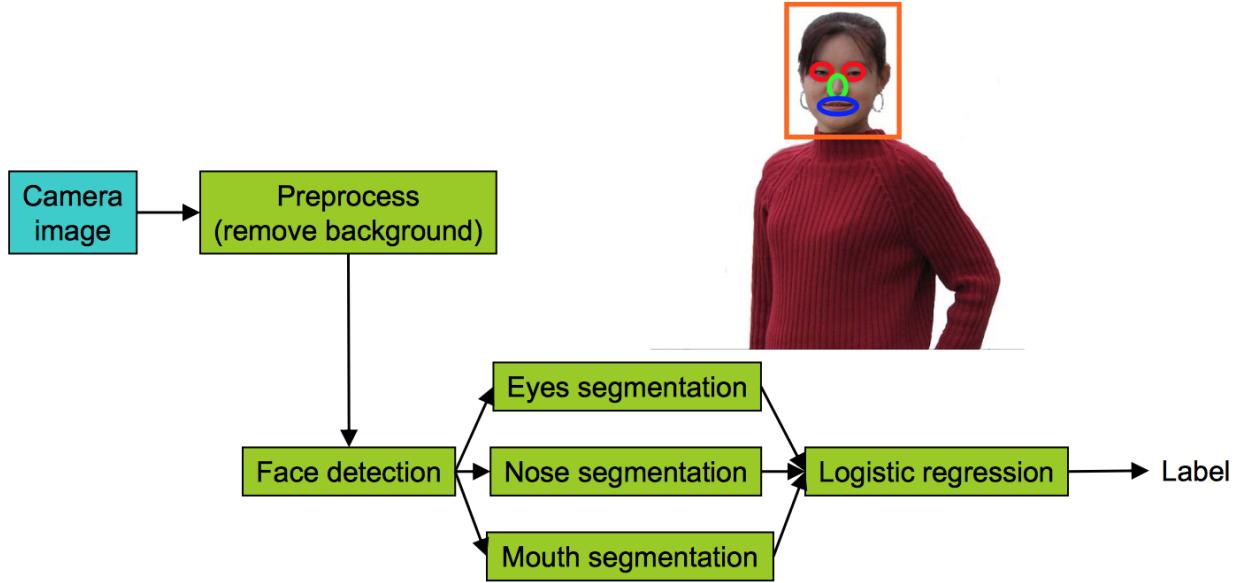


Figure 1: Face recognition pipeline

## 5. Final logistic regression step to predict the label

If you build a complicated system like this one, you might want to figure out how much error is attributable to each of the components, how good is each of these green boxes. Indeed, if one of these boxes is really problematic, you might want to spend more time trying to improve the performance of that one green box. How do you decide what part to focus on?

One thing we can do is plug in the ground-truth for each component, and see how accuracy changes. Let's say the overall accuracy of the system is 85% (pretty bad). You can now take your development set and manually give it the perfect background removal, that is, instead of using your background removal algorithm, manually specify the perfect background removal yourself (using photoshop for instance), and look at how much that affect the performance of the overall system.

Now let's say the accuracy only improves by 0.1%. This gives us an upperbound, that is even if we worked for years on background removal, it wouldn't help our system by more than 0.1%.

Now let's give the pipeline the perfect face detection by specifying the position of the face manually, see how much we improve the performance, and so on.

The results are specified in the table 1.

Looking at the table, we know that working on the background removal won't help much. It also tells us where the biggest jumps are. We notice that having an accurate face detection mechanism really improves the performance, and similarly, the eyes really help making the prediction more accurate.

Error analysis is also useful when publishing a paper, since it's a convenient way to

Component	Accuracy
Overall system	85%
Preprocess (remove background)	85.1%
Face detection	91%
Eyes segmentation	95%
Nose segmentation	96%
Mouth segmentation	97%
Logistic regression	100%

Table 1: Accuracy when providing the system with the perfect component

analyze the error of an algorithm and explain which parts should be improved.

## Ablative analysis

While error analysis tries to explain the difference between current performance and perfect performance, ablative analysis tries to explain the difference between some baseline (much poorer) performance and current performance.

For instance, suppose you have built a good anti-spam classifier by adding lots of clever features to logistic regression

- Spelling correction
- Sender host features
- Email header features
- Email text parser features
- Javascript parser
- Features from embedded images

and your question is: How much did each of these components really help?

In this example, let's say that simple logistic regression without any clever features gets 94% performance, but when adding these clever features, we get 99.9% performance. In abaltive analysis, what we do is start from the current level of performance 99.9%, and slowly take away all of these features to see how it affects performance. The results are provided in table 2.

When presenting the results in a paper, ablative analysis really helps analyzing the features that helped decreasing the misclassification rate. Instead of simply giving the loss/error rate of the algorithm, we can provide evidence that some specific features are actually more important than others.

Component	Accuracy
Overall system	99.9%
Spelling correction	99.0%
Sender host features	98.9%
Email header features	98.9%
Email text parser features	95%
Javascript parser	94.5%
Features from images	94.0%

Table 2: Accuracy when removing feature from logistic regression

## Analyze your mistakes

Assume you are given a dataset with pictures of animals, and your goal is to identify pictures of cats that you would eventually send to the members of a community of cat lovers. You notice that there are many pictures of dogs in the original dataset, and wonders whether you should build a special algorithm to identify the pictures of dogs and avoid sending dogs pictures to cat lovers or not.

One thing you can do is take a 100 examples from your development set that are misclassified, and count up how many of these 100 mistakes are dogs. If 5% of them are dogs, then even if you come up with a solution to identidy your dogs, your error would only go down by 5%, that is your accuracy would go up from 90% to 90.5%. However, if 50 of these 100 errors are dogs, then you could improve your accuracy to reach 95%.

By analyzing your mistakes, you can focus on what's really important. If you notice that 80 out of your 100 mistakes are blurry images, then work hard on classifying correctly these blurry images. If you notice that 70 out of the 100 errors are great cats, then focus on this specific task of identifying great cats.

In brief, do not waste your time improving parts of your algorithm that won't really help decreasing your error rate, and focus on what really matters.

# 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 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  $\phi_j : \mathbb{R}^n \rightarrow \{-1, 1\}$  and an infinite vector  $\theta = [\theta_1 \ \theta_2 \ \dots]^T$ , 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  $\phi_j$  *weak hypotheses*. Given a training set  $(x^{(1)}, y^{(1)}), \dots, (x^{(m)}, y^{(m)})$ , we call a vector  $p = (p^{(1)}, \dots, p^{(m)})$  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  $\phi_j$  such that

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

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  $p^{(1)}, \dots, p^{(m)}$  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  $\phi_j : \mathbb{R}^n \rightarrow \{-1, 1\}$  such that

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

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 classifier 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  $\theta_j$  to

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

while leaving  $\theta_k$  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  $\phi_k$  classifies correctly and incorrectly, respectively. Then finding  $\theta_k$  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  $\phi_t$  satisfying the margin condition (1). We present the full boosting algorithm in Figure 2. It proceeds in iterations  $t = 1, 2, 3, \dots$ . We represent the set of hypotheses returned by the weak learning algorithm at time  $t$  by  $\{\phi_1, \dots, \phi_t\}$ .

## 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  $t = 1, 2, \dots$ :

- (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  $\phi_t : \mathbb{R}^n \rightarrow \{-1, 1\}$  from the distribution  $p = (p^{(1)}, \dots, p^{(m)})$  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  $\theta$ , the misclassification error satisfies

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

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

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

and so if  $J(\theta) < \frac{1}{m}$  then the vector  $\theta$  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  $\log(1 - 4\gamma^2) \leq -4\gamma^2$ , we see that if the number of rounds of boosting—the number of weak classifiers we use—satisfies

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

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  $j \in \{1, 2, \dots, n\}$ , and returns

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

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  $p^{(1)}, \dots, p^{(m)}$  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

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

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  $j = 1, 2, \dots, n$ , we sort the raw input features so that

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

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

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

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 calculation 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  $1 - \widehat{\text{Err}}(\phi_{j,s}, p)$ , that is, the error of

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

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

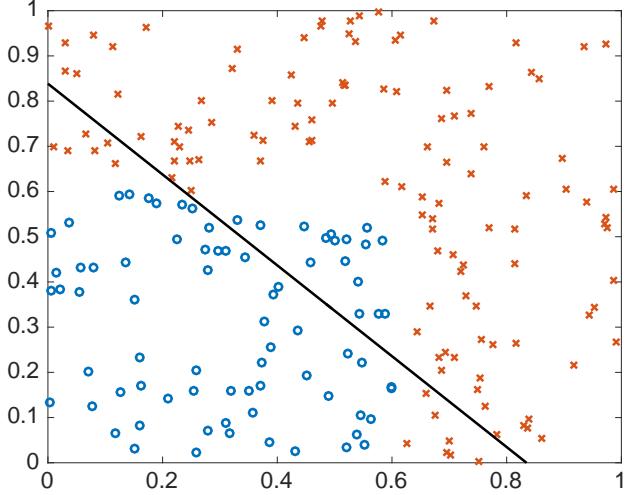


Figure 3: Best logistic regression classifier using the raw features  $x \in \mathbb{R}^2$  (and a bias term  $x_0 = 1$ ) for the example considered here.

### 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

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

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  $s \approx .23$ .

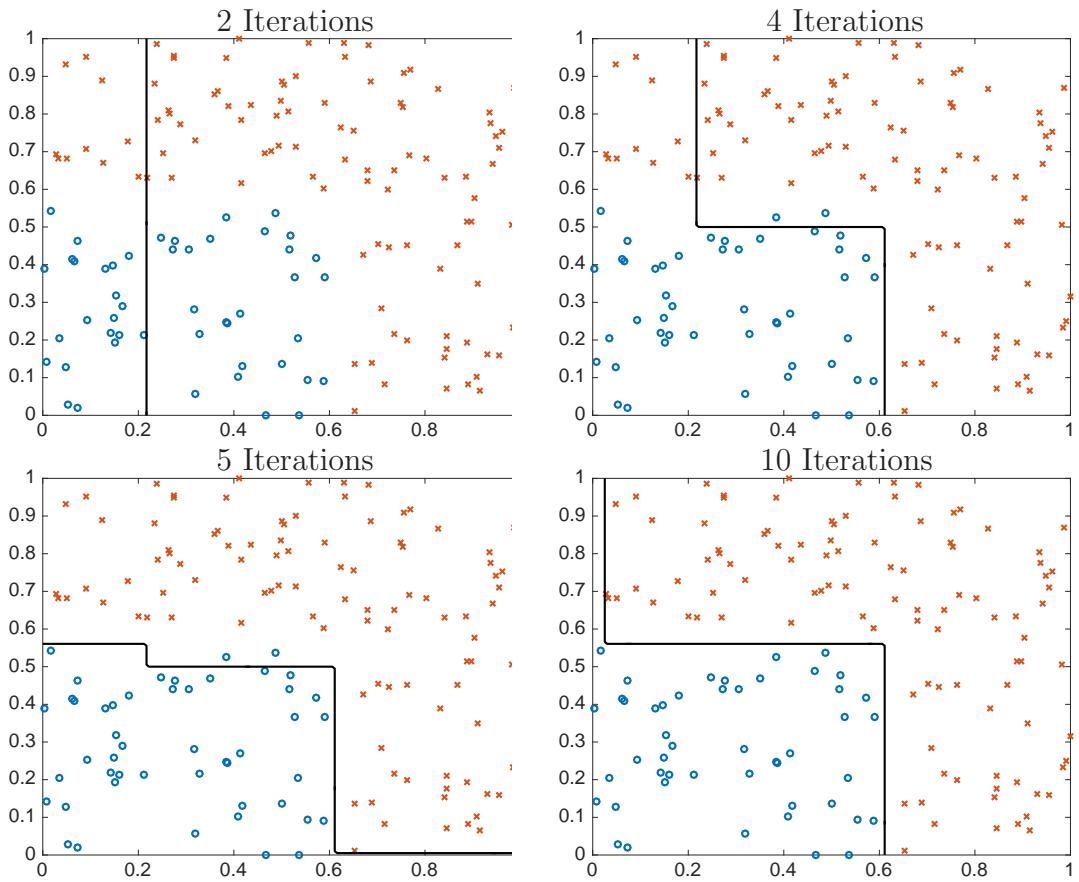


Figure 4: Boosted decision stumps after  $t = 2, 4, 5$ , and  $10$  iterations of boosting, respectively.

### 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  $x_j \in \{1, 2, \dots, k\}$  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  $C \subset \{1, \dots, k\}$  of categories.

Another natural variation is the *boosted decision tree*, in which instead of a single level decision for the weak learners, we consider conjunctions 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

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

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

# 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:

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

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  $\rho$ , we can show that:

$$\text{Var}(\bar{X}) = \text{Var}\left(\frac{1}{n} \sum_i X_i\right) \tag{1}$$

$$= \frac{1}{n^2} \sum_{i,j} \text{Cov}(X_i, X_j) \tag{2}$$

$$= \frac{n\sigma^2}{n^2} + \frac{n(n-1)\rho\sigma^2}{n^2} \tag{3}$$

$$= \rho\sigma^2 + \frac{1-\rho}{n}\sigma^2 \tag{4}$$

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 Bootstrap

Bagging stands for "Bootstrap 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  $S_1, S_2, \dots$  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  $Z_1, Z_2, \dots, Z_M$ . 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:

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

Bagging creates less correlated predictors than if they were all simply trained on  $S$ , thereby decreasing  $\rho$ . 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  $\rho$  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  $M \rightarrow \infty$ , 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 one 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  $\rho$  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

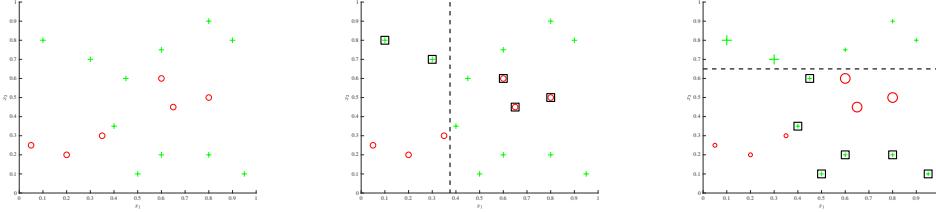
While some of the disadvantages include:

- Increase 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  $(x_1, y_1), (x_2, y_2), \dots (x_N, y_N)$   
**Output:** Ensemble classifier  $f(x)$

- 1  $w_i \leftarrow \frac{1}{N}$  for  $i = 1, 2, \dots, N$
- 2 **for**  $m = 0$  **to**  $M$  **do**
- 3     Fit weak classifier  $G_m$  to training data weighted by  $w_i$
- 4     Compute weighted error  $err_m = \frac{\sum_i w_i \mathbb{1}(y_i \neq G_m(x_i))}{\sum w_i}$
- 5     Compute weight  $\alpha_m = \log(\frac{1-err_m}{err_m})$
- 6      $w_i \leftarrow w_i * \exp(\alpha_m \mathbb{1}(y_i \neq G_m(x_i)))$
- 7 **end**
- 8  $f(x) = \text{sign}(\sum_m \alpha_m G_m(x))$

---

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  $(x_1, y_1), (x_2, y_2), \dots (x_N, y_N)$   
**Output:** Ensemble classifier  $f(x)$

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

---

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  $\gamma$ . 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:

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

## 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

## Hoeffding's inequality

John Duchi

## 1 Basic probability bounds

A basic question in probability, statistics, and machine learning is the following: given a random variable  $Z$  with expectation  $\mathbb{E}[Z]$ , how likely is  $Z$  to be close to its expectation? And more precisely, how close is it likely to be? With that in mind, these notes give a few tools for computing bounds of the form

$$\mathbb{P}(Z \geq \mathbb{E}[Z] + t) \text{ and } \mathbb{P}(Z \leq \mathbb{E}[Z] - t) \quad (1)$$

for  $t \geq 0$ .

Our first bound is perhaps the most basic of all probability inequalities, and it is known as Markov's inequality. Given its basic-ness, it is perhaps unsurprising that its proof is essentially only one line.

**Proposition 1** (Markov's inequality). *Let  $Z \geq 0$  be a non-negative random variable. Then for all  $t \geq 0$ ,*

$$\mathbb{P}(Z \geq t) \leq \frac{\mathbb{E}[Z]}{t}.$$

**Proof** We note that  $\mathbb{P}(Z \geq t) = \mathbb{E}[\mathbf{1}\{Z \geq t\}]$ , and that if  $Z \geq t$ , then it must be the case that  $Z/t \geq 1 \geq \mathbf{1}\{Z \geq t\}$ , while if  $Z < t$ , then we still have  $Z/t \geq 0 = \mathbf{1}\{Z \geq t\}$ . Thus

$$\mathbb{P}(Z \geq t) = \mathbb{E}[\mathbf{1}\{Z \geq t\}] \leq \mathbb{E}\left[\frac{Z}{t}\right] = \frac{\mathbb{E}[Z]}{t},$$

as desired. □

Essentially all other bounds on the probabilities (1) are variations on Markov's inequality. The first variation uses second moments—the variance—of a random variable rather than simply its mean, and is known as Chebyshev's inequality.

**Proposition 2** (Chebyshev's inequality). *Let  $Z$  be any random variable with  $\text{Var}(Z) < \infty$ . Then*

$$\mathbb{P}(Z \geq \mathbb{E}[Z] + t \text{ or } Z \leq \mathbb{E}[Z] - t) \leq \frac{\text{Var}(Z)}{t^2}$$

for  $t \geq 0$ .

**Proof** The result is an immediate consequence of Markov's inequality. We note that if  $Z \geq \mathbb{E}[Z] + t$ , then certainly we have  $(Z - \mathbb{E}[Z])^2 \geq t^2$ , and similarly if  $Z \leq \mathbb{E}[Z] - t$  we have  $(Z - \mathbb{E}[Z])^2 \geq t^2$ . Thus

$$\begin{aligned} \mathbb{P}(Z \geq \mathbb{E}[Z] + t \text{ or } Z \leq \mathbb{E}[Z] - t) &= \mathbb{P}((Z - \mathbb{E}[Z])^2 \geq t^2) \\ &\stackrel{(i)}{\leq} \frac{\mathbb{E}[(Z - \mathbb{E}[Z])^2]}{t^2} = \frac{\text{Var}(Z)}{t^2}, \end{aligned}$$

where step (i) is Markov's inequality.  $\square$

A nice consequence of Chebyshev's inequality is that averages of random variables with finite variance converge to their mean. Let us give an example of this fact. Suppose that  $Z_i$  are i.i.d. and satisfy  $\mathbb{E}[Z_i] = 0$ . Then  $\mathbb{E}[Z_i] = 0$ , while if we define  $\bar{Z} = \frac{1}{n} \sum_{i=1}^n Z_i$  then

$$\text{Var}(\bar{Z}) = \mathbb{E} \left[ \left( \frac{1}{n} \sum_{i=1}^n Z_i \right)^2 \right] = \frac{1}{n^2} \sum_{i,j \leq n} \mathbb{E}[Z_i Z_j] = \frac{1}{n^2} \sum_{i=1}^n \mathbb{E}[Z_i^2] = \frac{\text{Var}(Z_1)}{n}.$$

In particular, for any  $t \geq 0$  we have

$$\mathbb{P} \left( \left| \frac{1}{n} \sum_{i=1}^n Z_i \right| \geq t \right) \leq \frac{\text{Var}(Z_1)}{nt^2},$$

so that  $\mathbb{P}(|\bar{Z}| \geq t) \rightarrow 0$  for any  $t > 0$ .

## 2 Moment generating functions

Often, we would like sharper—even exponential—bounds on the probability that a random variable  $Z$  exceeds its expectation by much. With that in mind, we need a stronger condition than finite variance, for which moment generating functions are natural candidates. (Conveniently, they also play nicely with sums, as we will see.) Recall that for a random variable  $Z$ , the *moment generating function* of  $Z$  is the function

$$M_Z(\lambda) := \mathbb{E}[\exp(\lambda Z)], \quad (2)$$

which may be infinite for some  $\lambda$ .

### 2.1 Chernoff bounds

Chernoff bounds use of moment generating functions in an essential way to give exponential deviation bounds.

**Proposition 3** (Chernoff bounds). *Let  $Z$  be any random variable. Then for any  $t \geq 0$ ,*

$$\mathbb{P}(Z \geq \mathbb{E}[Z] + t) \leq \min_{\lambda \geq 0} \mathbb{E}[e^{\lambda(Z - \mathbb{E}[Z])}] e^{-\lambda t} = \min_{\lambda \geq 0} M_{Z - \mathbb{E}[Z]}(\lambda) e^{-\lambda t}$$

and

$$\mathbb{P}(Z \leq \mathbb{E}[Z] - t) \leq \min_{\lambda \geq 0} \mathbb{E}[e^{\lambda(\mathbb{E}[Z] - Z)}] e^{-\lambda t} = \min_{\lambda \geq 0} M_{\mathbb{E}[Z] - Z}(\lambda) e^{-\lambda t}.$$

**Proof** We only prove the first inequality, as the second is completely identical. We use Markov's inequality. For any  $\lambda > 0$ , we have  $Z \geq \mathbb{E}[Z] + t$  if and only if  $e^{\lambda Z} \geq e^{\lambda \mathbb{E}[Z] + \lambda t}$ , or  $e^{\lambda(Z - \mathbb{E}[Z])} \geq e^{\lambda t}$ . Thus, we have

$$\mathbb{P}(Z - \mathbb{E}[Z] \geq t) = \mathbb{P}(e^{\lambda(Z - \mathbb{E}[Z])} \geq e^{\lambda t}) \stackrel{(i)}{\leq} \mathbb{E}[e^{\lambda(Z - \mathbb{E}[Z])}] e^{-\lambda t},$$

where the inequality (i) follows from Markov's inequality. As our choice of  $\lambda > 0$  did not matter, we can take the best one by minimizing the right side of the bound. (And noting that certainly the bound holds at  $\lambda = 0$ .)  $\square$

The important result is that Chernoff bounds “play nicely” with summations, which is a consequence of the moment generating function. Let us assume that  $Z_i$  are independent. Then we have that

$$M_{Z_1+\dots+Z_n}(\lambda) = \prod_{i=1}^n M_{Z_i}(\lambda),$$

which we see because

$$\mathbb{E} \left[ \exp \left( \lambda \sum_{i=1}^n Z_i \right) \right] = \mathbb{E} \left[ \prod_{i=1}^n \exp(\lambda Z_i) \right] = \prod_{i=1}^n \mathbb{E}[\exp(\lambda Z_i)],$$

by of the independence of the  $Z_i$ . This means that when we calculate a Chernoff bound of a sum of i.i.d. variables, we need only calculate the moment generating function for *one* of them. Indeed, suppose that  $Z_i$  are i.i.d. and (for simplicity) mean zero. Then

$$\begin{aligned} \mathbb{P} \left( \sum_{i=1}^n Z_i \geq t \right) &\leq \frac{\prod_{i=1}^n \mathbb{E}[\exp(\lambda Z_i)]}{e^{\lambda t}} \\ &= (\mathbb{E}[e^{\lambda Z_1}])^n e^{-\lambda t}, \end{aligned}$$

by the Chernoff bound.

## 2.2 Moment generating function examples

Now we give several examples of moment generating functions, which enable us to give a few nice deviation inequalities as a result. For all of our examples, we will have very convienent bounds of the form

$$M_Z(\lambda) = \mathbb{E}[e^{\lambda Z}] \leq \exp \left( \frac{C^2 \lambda^2}{2} \right) \quad \text{for all } \lambda \in \mathbb{R},$$

for some  $C \in \mathbb{R}$  (which depends on the distribution of  $Z$ ); this form is *very* nice for applying Chernoff bounds.

We begin with the classical normal distribution, where  $Z \sim \mathcal{N}(0, \sigma^2)$ . Then we have

$$\mathbb{E}[\exp(\lambda Z)] = \exp \left( \frac{\lambda^2 \sigma^2}{2} \right),$$

which one obtains via a calculation that we omit. (You should work this out if you are curious!)

A second example is known as a Rademacher random variable, or the random sign variable. Let  $S = 1$  with probability  $\frac{1}{2}$  and  $S = -1$  with probability  $\frac{1}{2}$ . Then we claim that

$$\mathbb{E}[e^{\lambda S}] \leq \exp\left(\frac{\lambda^2}{2}\right) \quad \text{for all } \lambda \in \mathbb{R}. \quad (3)$$

To see inequality (3), we use the Taylor expansion of the exponential function, that is, that  $e^x = \sum_{k=0}^{\infty} \frac{x^k}{k!}$ . Note that  $\mathbb{E}[S^k] = 0$  whenever  $k$  is odd, while  $\mathbb{E}[S^k] = 1$  whenever  $k$  is even. Then we have

$$\begin{aligned} \mathbb{E}[e^{\lambda S}] &= \sum_{k=0}^{\infty} \frac{\lambda^k \mathbb{E}[S^k]}{k!} \\ &= \sum_{k=0,2,4,\dots} \frac{\lambda^k}{k!} = \sum_{k=0}^{\infty} \frac{\lambda^{2k}}{(2k)!}. \end{aligned}$$

Finally, we use that  $(2k)! \geq 2^k \cdot k!$  for all  $k = 0, 1, 2, \dots$ , so that

$$\mathbb{E}[e^{\lambda S}] \leq \sum_{k=0}^{\infty} \frac{(\lambda^2)^k}{2^k \cdot k!} = \sum_{k=0}^{\infty} \left(\frac{\lambda^2}{2}\right)^k \frac{1}{k!} = \exp\left(\frac{\lambda^2}{2}\right).$$

Let us apply inequality (3) in a Chernoff bound to see how large a sum of i.i.d. random signs is likely to be.

We have that if  $Z = \sum_{i=1}^n S_i$ , where  $S_i \in \{\pm 1\}$  is a random sign, then  $\mathbb{E}[Z] = 0$ . By the Chernoff bound, it becomes immediately clear that

$$\mathbb{P}(Z \geq t) \leq \mathbb{E}[e^{\lambda Z}] e^{-\lambda t} = \mathbb{E}[e^{\lambda S_1}]^n e^{-\lambda t} \leq \exp\left(\frac{n\lambda^2}{2}\right) e^{-\lambda t}.$$

Applying the Chernoff bound technique, we may minimize this in  $\lambda \geq 0$ , which is equivalent to finding

$$\min_{\lambda \geq 0} \left\{ \frac{n\lambda^2}{2} - \lambda t \right\}.$$

Luckily, this is a convenient function to minimize: taking derivatives and setting to zero, we have  $n\lambda - t = 0$ , or  $\lambda = t/n$ , which gives

$$\mathbb{P}(Z \geq t) \leq \exp\left(-\frac{t^2}{2n}\right).$$

In particular, taking  $t = \sqrt{2n \log \frac{1}{\delta}}$ , we have

$$\mathbb{P} \left( \sum_{i=1}^n S_i \geq \sqrt{2n \log \frac{1}{\delta}} \right) \leq \delta.$$

So  $Z = \sum_{i=1}^n S_i = O(\sqrt{n})$  with extremely high probability—the sum of  $n$  independent random signs is essentially never larger than  $O(\sqrt{n})$ .

### 3 Hoeffding's lemma and Hoeffding's inequality

Hoeffding's inequality is a powerful technique—perhaps the most important inequality in learning theory—for bounding the probability that sums of bounded random variables are too large or too small. We will state the inequality, and then we will prove a weakened version of it based on our moment generating function calculations earlier.

**Theorem 4** (Hoeffding's inequality). *Let  $Z_1, \dots, Z_n$  be independent bounded random variables with  $Z_i \in [a, b]$  for all  $i$ , where  $-\infty < a \leq b < \infty$ . Then*

$$\mathbb{P} \left( \frac{1}{n} \sum_{i=1}^n (Z_i - \mathbb{E}[Z_i]) \geq t \right) \leq \exp \left( -\frac{2nt^2}{(b-a)^2} \right)$$

and

$$\mathbb{P} \left( \frac{1}{n} \sum_{i=1}^n (Z_i - \mathbb{E}[Z_i]) \leq -t \right) \leq \exp \left( -\frac{2nt^2}{(b-a)^2} \right)$$

for all  $t \geq 0$ .

We prove Theorem 4 by using a combination of (1) Chernoff bounds and (2) a classic lemma known as Hoeffding's lemma, which we now state.

**Lemma 5** (Hoeffding's lemma). *Let  $Z$  be a bounded random variable with  $Z \in [a, b]$ . Then*

$$\mathbb{E}[\exp(\lambda(Z - \mathbb{E}[Z]))] \leq \exp \left( \frac{\lambda^2(b-a)^2}{8} \right) \text{ for all } \lambda \in \mathbb{R}.$$

**Proof** We prove a slightly weaker version of this lemma with a factor of 2 instead of 8 using our random sign moment generating bound and an inequality known as *Jensen's inequality* (we will see this very important inequality later in our derivation of the EM algorithm). Jensen's inequality states the following: if  $f : \mathbb{R} \rightarrow \mathbb{R}$  is a *convex* function, meaning that  $f$  is bowl-shaped, then

$$f(\mathbb{E}[Z]) \leq \mathbb{E}[f(Z)].$$

The simplest way to remember this inequality is to think of  $f(t) = t^2$ , and note that if  $\mathbb{E}[Z] = 0$  then  $f(\mathbb{E}[Z]) = 0$ , while we generally have  $\mathbb{E}[Z^2] > 0$ . In any case,  $f(t) = \exp(t)$  and  $f(t) = \exp(-t)$  are convex functions.

We use a clever technique in probability theory known as *symmetrization* to give our result (you are not expected to know this, but it is a very common technique in probability theory, machine learning, and statistics, so it is good to have seen). First, let  $Z'$  be an independent copy of  $Z$  with the same distribution, so that  $Z' \in [a, b]$  and  $\mathbb{E}[Z'] = \mathbb{E}[Z]$ , but  $Z$  and  $Z'$  are independent. Then

$$\mathbb{E}_Z[\exp(\lambda(Z - \mathbb{E}_Z[Z]))] = \mathbb{E}_Z[\exp(\lambda(Z - \mathbb{E}_{Z'}[Z']))] \stackrel{(i)}{\leq} \mathbb{E}_Z[\mathbb{E}_{Z'}\exp(\lambda(Z - Z'))],$$

where  $\mathbb{E}_Z$  and  $\mathbb{E}_{Z'}$  indicate expectations taken with respect to  $Z$  and  $Z'$ . Here, step (i) uses Jensen's inequality applied to  $f(x) = e^{-x}$ . Now, we have

$$\mathbb{E}[\exp(\lambda(Z - \mathbb{E}[Z]))] \leq \mathbb{E}[\exp(\lambda(Z - Z'))].$$

Now, we note a curious fact: the difference  $Z - Z'$  is symmetric about zero, so that if  $S \in \{-1, 1\}$  is a random sign variable, then  $S(Z - Z')$  has exactly the same distribution as  $Z - Z'$ . So we have

$$\begin{aligned} \mathbb{E}_{Z,Z'}[\exp(\lambda(Z - Z'))] &= \mathbb{E}_{Z,Z',S}[\exp(\lambda S(Z - Z'))] \\ &= \mathbb{E}_{Z,Z'}[\mathbb{E}_S[\exp(\lambda S(Z - Z')) | Z, Z']]. \end{aligned}$$

Now we use inequality (3) on the moment generating function of the random sign, which gives that

$$\mathbb{E}_S[\exp(\lambda S(Z - Z')) | Z, Z'] \leq \exp\left(\frac{\lambda^2(Z - Z')^2}{2}\right).$$

But of course, by assumption we have  $|Z - Z'| \leq (b - a)$ , so  $(Z - Z')^2 \leq (b - a)^2$ . This gives

$$\mathbb{E}_{Z,Z'}[\exp(\lambda(Z - Z'))] \leq \exp\left(\frac{\lambda^2(b - a)^2}{2}\right).$$

This is the result (except with a factor of 2 instead of 8).  $\square$

Now we use Hoeffding's lemma to prove Theorem 4, giving only the upper tail (i.e. the probability that  $\frac{1}{n} \sum_{i=1}^n (Z_i - \mathbb{E}[Z_i]) \geq t$ ) as the lower tail has a similar proof. We use the Chernoff bound technique, which immediately tells us that

$$\begin{aligned} \mathbb{P}\left(\frac{1}{n} \sum_{i=1}^n (Z_i - \mathbb{E}[Z_i]) \geq t\right) &= \mathbb{P}\left(\sum_{i=1}^n (Z_i - \mathbb{E}[Z_i]) \geq nt\right) \\ &\leq \mathbb{E}\left[\exp\left(\lambda \sum_{i=1}^n (Z_i - \mathbb{E}[Z_i])\right)\right] e^{-\lambda nt} \\ &= \left(\prod_{i=1}^n \mathbb{E}[e^{\lambda(Z_i - \mathbb{E}[Z_i])}]\right) e^{-\lambda nt} \stackrel{(i)}{\leq} \left(\prod_{i=1}^n e^{\frac{\lambda^2(b-a)^2}{8}}\right) e^{-\lambda nt} \end{aligned}$$

where inequality (i) is Hoeffding's Lemma (Lemma 5). Rewriting this slightly and minimizing over  $\lambda \geq 0$ , we have

$$\mathbb{P}\left(\frac{1}{n} \sum_{i=1}^n (Z_i - \mathbb{E}[Z_i]) \geq t\right) \leq \min_{\lambda \geq 0} \exp\left(\frac{n\lambda^2(b-a)^2}{8} - \lambda nt\right) = \exp\left(-\frac{2nt^2}{(b-a)^2}\right),$$

as desired.

# CS229 Supplemental Lecture notes

John Duchi

## 1 Binary classification

In **binary classification** problems, the target  $y$  can take on at only two values. In this set of notes, we show how to model this problem by letting  $y \in \{-1, +1\}$ , where we say that  $y$  is a 1 if the example is a member of the positive class and  $y = -1$  if the example is a member of the negative class. We assume, as usual, that we have input features  $x \in \mathbb{R}^n$ .

As in our standard approach to supervised learning problems, we first pick a representation for our hypothesis class (what we are trying to learn), and after that we pick a loss function that we will minimize. In binary classification problems, it is often convenient to use a hypothesis class of the form  $h_\theta(x) = \theta^T x$ , and, when presented with a new example  $x$ , we classify it as positive or negative depending on the sign of  $\theta^T x$ , that is, our predicted label is

$$\text{sign}(h_\theta(x)) = \text{sign}(\theta^T x) \quad \text{where} \quad \text{sign}(t) = \begin{cases} 1 & \text{if } t > 0 \\ 0 & \text{if } t = 0 \\ -1 & \text{if } t < 0. \end{cases}$$

In a binary classification problem, then, the hypothesis  $h_\theta$  with parameter vector  $\theta$  classifies a particular example  $(x, y)$  correctly if

$$\text{sign}(\theta^T x) = y \quad \text{or equivalently} \quad y\theta^T x > 0. \quad (1)$$

The quantity  $y\theta^T x$  in expression (1) is a very important quantity in binary classification, important enough that we call the value

$$yx^T\theta$$

the *margin* for the example  $(x, y)$ . Often, though not always, one interprets the value  $h_\theta(x) = x^T\theta$  as a measure of the confidence that the parameter

vector  $\theta$  assigns to labels for the point  $x$ : if  $x^T\theta$  is very negative (or very positive), then we more strongly believe the label  $y$  is negative (or positive).

Now that we have chosen a representation for our data, we must choose a loss function. Intuitively, we would like to choose some loss function so that for our training data  $\{(x^{(i)}, y^{(i)})\}_{i=1}^m$ , the  $\theta$  chosen makes the margin  $y^{(i)}\theta^T x^{(i)}$  very large for each training example. Let us fix a hypothetical example  $(x, y)$ , let  $z = yx^T\theta$  denote the margin, and let  $\varphi : \mathbb{R} \rightarrow \mathbb{R}$  be the loss function—that is, the loss for the example  $(x, y)$  with margin  $z = yx^T\theta$  is  $\varphi(z) = \varphi(yx^T\theta)$ . For any particular loss function, the empirical risk that we minimize is then

$$J(\theta) = \frac{1}{m} \sum_{i=1}^m \varphi(y^{(i)}\theta^T x^{(i)}). \quad (2)$$

Consider our desired behavior: we wish to have  $y^{(i)}\theta^T x^{(i)}$  positive for each training example  $i = 1, \dots, m$ , and we should penalize those  $\theta$  for which  $y^{(i)}\theta^T x^{(i)} < 0$  frequently in the training data. Thus, an intuitive choice for our loss would be one with  $\varphi(z)$  small if  $z > 0$  (the margin is positive), while  $\varphi(z)$  is large if  $z < 0$  (the margin is negative). Perhaps the most natural such loss is the *zero-one* loss, given by

$$\varphi_{\text{zo}}(z) = \begin{cases} 1 & \text{if } z \leq 0 \\ 0 & \text{if } z > 0. \end{cases}$$

In this case, the risk  $J(\theta)$  is simply the average number of mistakes—misclassifications—the parameter  $\theta$  makes on the training data. Unfortunately, the loss  $\varphi_{\text{zo}}$  is discontinuous, non-convex (why this matters is a bit beyond the scope of the course), and perhaps even more vexingly, NP-hard to minimize. So we prefer to choose losses that have the shape given in Figure 1. That is, we will essentially always use losses that satisfy

$$\varphi(z) \rightarrow 0 \text{ as } z \rightarrow \infty, \text{ while } \varphi(z) \rightarrow \infty \text{ as } z \rightarrow -\infty.$$

As a few different examples, here are three loss functions that we will see either now or later in the class, all of which are commonly used in machine learning.

(i) The *logistic loss* uses

$$\varphi_{\text{logistic}}(z) = \log(1 + e^{-z})$$

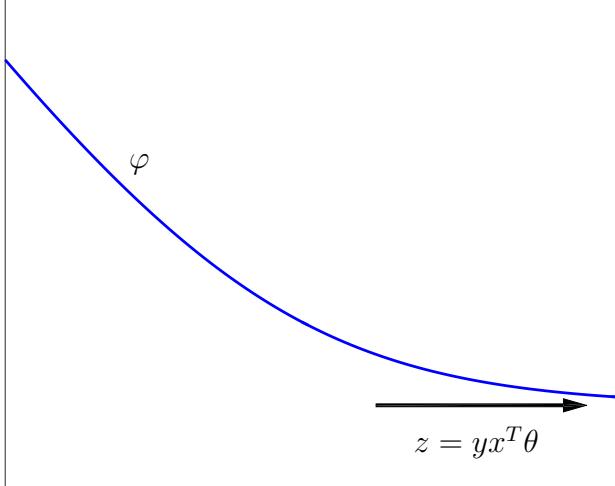


Figure 1: The rough shape of loss we desire: the loss is convex and continuous, and tends to zero as the margin  $z = yx^T\theta \rightarrow \infty$ .

(ii) The *hinge loss* uses

$$\varphi_{\text{hinge}}(z) = [1 - z]_+ = \max\{1 - z, 0\}$$

(iii) The *exponential loss* uses

$$\varphi_{\text{exp}}(z) = e^{-z}.$$

In Figure 2, we plot each of these losses against the margin  $z = yx^T\theta$ , noting that each goes to zero as the margin grows, and each tends to  $+\infty$  as the margin becomes negative. The different loss functions lead to different machine learning procedures; in particular, the logistic loss  $\varphi_{\text{logistic}}$  is logistic regression, the hinge loss  $\varphi_{\text{hinge}}$  gives rise to so-called *support vector machines*, and the exponential loss gives rise to the classical version of *boosting*, both of which we will explore in more depth later in the class.

## 2 Logistic regression

With this general background in place, we now give a complementary view of logistic regression to that in Andrew Ng's lecture notes. When we

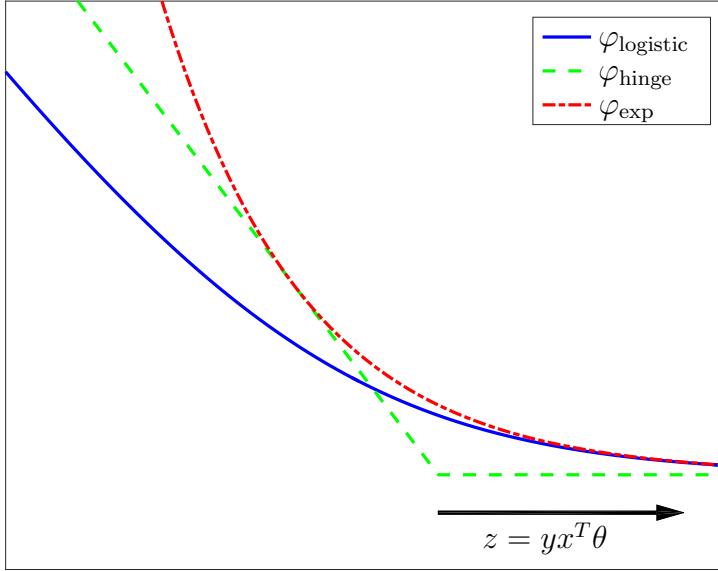


Figure 2: The three margin-based loss functions logistic loss, hinge loss, and exponential loss.

use binary labels  $y \in \{-1, 1\}$ , it is possible to write logistic regression more compactly. In particular, we use the logistic loss

$$\varphi_{\text{logistic}}(yx^T \theta) = \log(1 + \exp(-yx^T \theta)),$$

and the *logistic regression* algorithm corresponds to choosing  $\theta$  that minimizes

$$J(\theta) = \frac{1}{m} \sum_{i=1}^m \varphi_{\text{logistic}}(y^{(i)} \theta^T x^{(i)}) = \frac{1}{m} \sum_{i=1}^m \log(1 + \exp(-y^{(i)} \theta^T x^{(i)})). \quad (3)$$

Roughly, we hope that choosing  $\theta$  to minimize the average logistic loss will yield a  $\theta$  for which  $y^{(i)} \theta^T x^{(i)} > 0$  for most (or even all!) of the training examples.

## 2.1 Probabilistic interpretation

Similar to the linear regression (least-squares) case, it is possible to give a probabilistic interpretation of logistic regression. To do this, we define the

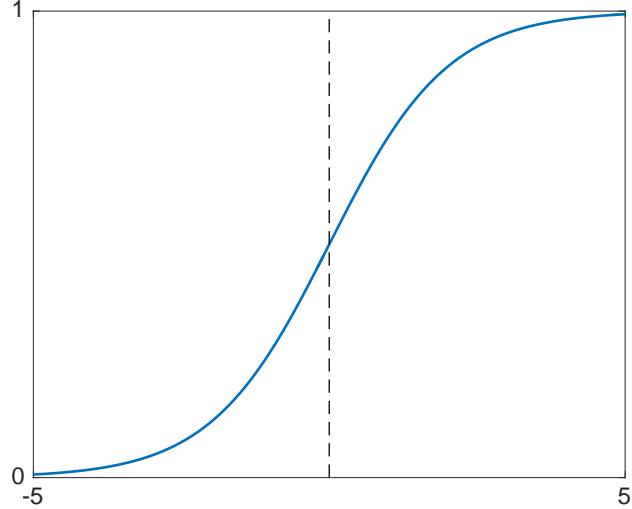


Figure 3: Sigmoid function

*sigmoid function* (also often called the *logistic function*)

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

which is plotted in Fig. 3. In particular, the sigmoid function satisfies

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

so we can use it to define a probability model for binary classification. In particular, for  $y \in \{-1, 1\}$ , we define the *logistic model* for classification as

$$p(Y = y | x; \theta) = g(yx^T \theta) = \frac{1}{1 + e^{-yx^T \theta}}. \quad (4)$$

For interpretation, we see that if the margin  $yx^T \theta$  is large—bigger than, say, 5 or so—then  $p(Y = y | x; \theta) = g(yx^T \theta) \approx 1$ , that is, we assign nearly probability 1 to the event that the label is  $y$ . Conversely, if  $yx^T \theta$  is quite negative, then  $p(Y = y | x; \theta) \approx 0$ .

By redefining our hypothesis class as

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

then we see that the likelihood of the training data is

$$L(\theta) = \prod_{i=1}^m p(Y = y^{(i)} | x^{(i)}; \theta) = \prod_{i=1}^m h_\theta(y^{(i)} x^{(i)}),$$

and the log-likelihood is precisely

$$\ell(\theta) = \sum_{i=1}^m \log h_\theta(y^{(i)} x^{(i)}) = - \sum_{i=1}^m \log \left( 1 + e^{-y^{(i)} \theta^T x^{(i)}} \right) = -mJ(\theta),$$

where  $J(\theta)$  is exactly the logistic regression risk from Eq. (3). That is, maximum likelihood in the logistic model (4) is the same as minimizing the average logistic loss, and we arrive at logistic regression again.

## 2.2 Gradient descent methods

The final part of logistic regression is to actually fit the model. As is usually the case, we consider gradient-descent-based procedures for performing this minimization. With that in mind, we now show how to take derivatives of the logistic loss. For  $\varphi_{\text{logistic}}(z) = \log(1 + e^{-z})$ , we have the one-dimensional derivative

$$\frac{d}{dz} \varphi_{\text{logistic}}(z) = \varphi'_{\text{logistic}}(z) = \frac{1}{1 + e^{-z}} \cdot \frac{d}{dz} e^{-z} = -\frac{e^{-z}}{1 + e^{-z}} = -\frac{1}{1 + e^z} = -g(-z),$$

where  $g$  is the sigmoid function. Then we apply the chain rule to find that for a single training example  $(x, y)$ , we have

$$\frac{\partial}{\partial \theta_k} \varphi_{\text{logistic}}(yx^T \theta) = -g(-yx^T \theta) \frac{\partial}{\partial \theta_k} (yx^T \theta) = -g(-yx^T \theta) yx_k.$$

Thus, a stochastic gradient procedure for minimization of  $J(\theta)$  iteratively performs the following for iterations  $t = 1, 2, \dots$ , where  $\alpha_t$  is a stepsize at time  $t$ :

1. Choose an example  $i \in \{1, \dots, m\}$  uniformly at random
2. Perform the gradient update

$$\begin{aligned} \theta^{(t+1)} &= \theta^{(t)} - \alpha_t \cdot \nabla_{\theta} \varphi_{\text{logistic}}(y^{(i)} x^{(i)T} \theta^{(t)}) \\ &= \theta^{(t)} + \alpha_t g(-y^{(i)} x^{(i)T} \theta^{(t)}) y^{(i)} x^{(i)} = \theta^{(t)} + \alpha_t h_{\theta^{(t)}}(-y^{(i)} x^{(i)}) y^{(i)} x^{(i)}. \end{aligned}$$

This update is intuitive: if our current hypothesis  $h_{\theta^{(t)}}$  assigns probability close to 1 for the *incorrect* label  $-y^{(i)}$ , then we try to reduce the loss by moving  $\theta$  in the direction of  $y^{(i)}x^{(i)}$ . Conversely, if our current hypothesis  $h_{\theta^{(t)}}$  assigns probability close to 0 for the incorrect label  $-y^{(i)}$ , the update essentially does nothing.

# Maximum Entropy and Exponential Families

Christopher Ré  
(edits by Tri Dao and Anand Avati)

August 5, 2019

## Abstract

The goal of this note is to derive the exponential form of probability distribution from more basic considerations, in particular Entropy. It follows a description by ET Jaynes in Chapter 11 of his book *Probability Theory: the Logic of Science* [1].<sup>1</sup>

## 1 Motivating the Exponential Model

This section will motivate the exponential model form that we've seen in lecture.

**The Setup** The setup for our problem is that we are given a finite set of instances  $\mathcal{Y}$  and a set of  $m$  statistics  $(T_j, c_j)$  in which  $T_j : \mathcal{Y} \rightarrow \mathbb{R}$  and  $c_j \in \mathbb{R}$ . An instance (or possible world) is just an element in a set. We can think about a statistic as a measurement of an instance, it tells us the important features of that instance that are important for our model. More precisely, the only information we have about the instances is the values of  $T_i$  on these instances. Our goal is to find a probability function  $p$  such that

$$p : \mathcal{Y} \rightarrow [0, 1] \text{ such that } \sum_{y \in \mathcal{Y}} p(y) = 1.$$

The main goal of this note is to provide a set of assumptions under which such distributions have a specific functional form, the exponential family, that we saw in generalized linear model:

$$p(y; \eta) = \exp \{ \eta \cdot T(y) - a(\eta) \}$$

in which  $\eta \in \mathbb{R}^m$ ,  $T(y) \in \mathbb{R}^m$  and  $T(y)_j = T_j(y)$ . Notice that there is exactly one parameter for each statistic. As we'll see for discrete distributions, we are able to derive this exponential form as a consequence of a maximizing entropy subject to matching the statistics.<sup>2</sup>

---

<sup>1</sup>This work is available online in many places including <http://omega.albany.edu:8008/ETJ-PS/cc11g.ps>.

<sup>2</sup>Unfortunately, for continuous distributions, such a derivation does not work due to some technical issues with Entropy—this hasn't stopped folks from using it as justification.

## 1.1 The problem: Too many distributions!

We'll see the problem of defining a distribution from statistics (measurements). We'll see that often there are often many probability distributions that satisfy our constraints, and we'll be forced to pick among them.<sup>3</sup>

**The Constraints** We interpret a statistic as a constraint on  $p$  of the following form:

$$\mathbb{E}_p[T_j] = c_j \text{ i.e., } \sum_{i=1}^N T_j(y_i)p_i = \langle T_j, p \rangle = c_j$$

Let's get some notation to describe these constraints. Let  $N = |\mathcal{Y}|$ , then the probability we are after is  $p \in \mathbb{R}^N$  subject to constraints.

- There are  $m$  constraints of the form

$$\langle T_j, p \rangle = c_j \text{ for } j = 1, \dots, m.$$

- A single constraint of the form  $\sum_{i=1}^N p_i = 1$  to ensure that  $p$  is a probability distribution. We can write this more succinctly as  $\langle \mathbf{1}, p \rangle = 1$ .
- We also have that  $p_i \geq 0$  for  $i = 1, \dots, N$ .

More compactly, we can write all constraints in a matrix  $G$  as

$$G = \begin{pmatrix} \mathbf{1} \\ T \end{pmatrix} \in \mathbb{R}^{(m+1) \times N} \text{ so that } Gp = \begin{pmatrix} 1 \\ c \end{pmatrix}.$$

If  $\mathsf{N}(G) = \emptyset$ , then this means that  $p$  is uniquely defined as  $G$  has an inverse. In this case,  $p = G^{-1}c$ . However often  $m$  is much smaller than  $N$ , so that  $\mathsf{N}(G) \neq \emptyset$ —and there are many solutions that satisfy the constraints.

**Example 1.1** Suppose we have three possible worlds, i.e.,  $\mathcal{Y} = \{y_1, y_2, y_3\}$  and one statistic  $T(y_i) = i$  and  $c = 2.5$ . Then, we have:

$$G = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 2 & 3 \end{pmatrix} \text{ and } \mathsf{N}(G) = \begin{pmatrix} 1 \\ -2 \\ 1 \end{pmatrix}$$

Let  $p^{(1)} = (1/12, 1/3, 7/12)$  then  $Gp = (1, 2.5)^T$ —but so do (infinitely) many others, in particular  $q(\alpha) = p^{(1)} + \alpha(1, -2, 1)$  is valid so long as  $\alpha \in [-1/12, 1/6]$  (due to positivity).

---

<sup>3</sup>Throughout this section, it will be convenient to view  $p$  and  $T_j$  as functions from  $\mathcal{Y} \rightarrow \mathbb{R}$ —and also as vectors indexed by  $\mathcal{Y}$ . Their use should be clear from the context.

**Picking a probability distribution  $p$**  In the case  $\emptyset \neq \mathbf{N}(G)$ , there are *many* probability distributions we can pick. All of these distributions can be written as follows:

$$p = p^{(0)} + p^{(1)} \text{ in which } p^{(0)} \in \mathbf{N}(G) \text{ and } p^{(1)} \text{ satisfies } Gp^{(1)} = \begin{pmatrix} 1 \\ c \end{pmatrix}$$

**Example 1.2** Continuing the computation above, we see  $p^{(0)} = \alpha(1, -2, 1)$  is a vector in  $\mathbf{N}(G)$ .

Which  $p$  should we pick? Well, we'll use one method called the method of maximum entropy. In turn, this will lead to the fact that our function  $p$  has a very special form—the form of exponential family distributions!

## 1.2 Entropy

To pick among the distributions, we'll need some scoring method.<sup>4</sup> We'll cut to the chase here and define the entropy, which is a function on probability distributions  $p \in \mathbb{R}^N$  such that  $p \geq 0$  and  $\langle \mathbf{1}, p \rangle = 1$ .

$$H(p) = - \sum_{i=1}^N p_i \log p_i$$

Effectively, the entropy rewards one for “spreading” the distribution out more. One can motivate Entropy from axioms, and either Jaynes or the Wikipedia page is pretty good on this account.<sup>5</sup>. The intuition should be that entropy can be used to select the *least informative* prior, it's a way of making as few additional assumptions as possible. In other words, we want to encode the prior information given by the constraints on the statistics while being as “objective” or “agnostic” as possible. This is called the maximum entropy principle.

For example, one can verify that under no constraints,  $H(p)$  is maximized with  $p_i = N^{-1}$ —that is all alternatives have equal probability. This is what we mean by spread out.

We'll pick the distribution that maximizes entropy subject to our constraints. Mathematically, we'll examine:

$$\max_{p \in \mathbb{R}^N} H(p) \text{ s.t. } \langle \mathbf{1}, p \rangle = 1, p \geq 0, \text{ and } Tp = c$$

We will not discuss it, but under appropriate conditions there is a unique solution  $p$ .

<sup>4</sup>A few natural methods don't work as we might think they should (minimizing variance, etc.) See [1, Ch.11] for a description of these alternative approaches.

<sup>5</sup>[https://en.wikipedia.org/wiki/Entropy\\_\(information\\_theory\)#Rationale](https://en.wikipedia.org/wiki/Entropy_(information_theory)#Rationale)

### 1.3 The Lagrangian

We'll create a function called the Lagrangian that has the property that any critical point of the Lagrangian is a critical point of the *constrained* problem. We will show that all critical points of the Lagrangian (and so our original problem) can be written in the exponential format we described above.

To simplify our discussion, let's imagine that  $p > 0$ , i.e., there are no possible worlds  $y$  such that  $p(y) = 0$ . In this case, our problem reduces to:

$$\max_{p \in \mathbb{R}^N} H(p) \text{ s.t. } Tp = c \text{ and } \langle \mathbf{1}, p \rangle = 1$$

We can write the Lagrangian  $\mathcal{L} : \mathbb{R}^N \times (\mathbb{R}^m \times \mathbb{R}) \rightarrow \mathbb{R}$  as follows:

$$\mathcal{L}(p; \eta, \lambda) = H(p) + \langle \eta, Tp - c \rangle + \lambda (\langle \mathbf{1}, p \rangle - 1)$$

The special property of  $\mathcal{L}$  is that any critical point of our original solution, in particular any maximum or minimum corresponds to a critical point of the Lagrangian. Thus, if we prove something about critical points of the Lagrangian, we prove something about the critical points of the original function. Later in the course, we'll see more sophisticated uses of Lagrangians but for now we include a simple derivation below to give a hint what's going on. For this section, we'll assume this special property is true.

Due to that special property, we find the critical points of  $\mathcal{L}$  by differentiating with respect to  $p_i$  and setting the resulting equations to 0.

$$\begin{aligned} \frac{\partial}{\partial p_i} \mathcal{L} &= \frac{\partial}{\partial p_i} [H(p) + \langle \eta, Tp - c \rangle + \lambda (\langle \mathbf{1}, p \rangle - 1)] \\ &= -(\log p_i + 1) + \langle \eta, T(y_i) \rangle + \lambda \end{aligned}$$

Setting this expression equal to 0 and solving for  $p_i$  we learn:

$$\begin{aligned} p_i &= e^{\lambda-1} \exp \{ \langle \eta, T(y_i) \rangle \} \\ \Rightarrow p(y) &\propto \exp \{ \eta \cdot T(y) \} \end{aligned}$$

which is of the right form—except that we have one too many parameters, namely  $\lambda$ . Nevertheless, this is remarkable: at a critical point, it's always the case that the exponential family “pops out”!

**Eliminating  $\lambda$**  The parameter  $\lambda$  can be eliminated, which is the final step to match our original claimed exponential form. To do so, we sum over all the  $p_i$  which we know on one hand is equal to 1, and the other hand, we have the above expression for  $p_i$ . This gives us the following equation:

$$\sum_{i=1}^N p_i = 1 \text{ and } \sum_{i=1}^N p_i = e^{\lambda-1} \left( \sum_{i=1}^N \exp \{ \eta \cdot T(y_i) \} \right) \text{ thus } e^{-\lambda+1} = \left( \sum_{y \in \mathcal{Y}} \exp \{ \eta \cdot T(y) \} \right)$$

Thus, we have expressed  $\lambda$  as a function of  $\eta$  and we can eliminate it. To do so, we write:

$$\begin{aligned} Z(\eta) &= \sum_{y \in \mathcal{Y}} \exp \{\eta \cdot T(y)\} \\ \Rightarrow p(y; \eta) &= Z(\eta)^{-1} \exp \{\eta \cdot T(y)\} \\ &= \exp \{\eta \cdot T(y) - a(\eta)\} \quad \text{where } a(\eta) = \log Z(\eta) \end{aligned}$$

This function  $Z$  is called the *partition function*, and  $a$  is called the *log-partition function*. The above is the claimed exponential form we saw in lecture.

## 2 Why the Lagrangian? [optional]

We observe that this is a constrained optimization problem with *linear* constraints.<sup>6</sup>

Let  $r$  be the rank of  $G$  and so  $\dim(\mathbf{N}(G)) = N - r$ . We create a function  $\phi : \mathbb{R}^{N-r} \rightarrow \mathbb{R}$  such that there is a map between any point in the domain of  $\phi$  and a feasible solution to our constrained problem, and moreover  $\phi$  will take the same value as  $H$ . In contrast to our original constrained problem,  $\phi$  has an unconstrained domain (all of  $\mathbb{R}^{N-r}$ ), and so we can apply standard calculus to find its critical points. To that end, we define a (linear) map  $B \in \mathbb{R}^{N \times (N-r)}$  that has rank  $N - r$ . We also insist that  $B^T B = I_{N-r}$ . Such a  $B$  exists, as it is simply the first  $N - r$  columns of a change of basis matrix from the standard basis to an orthonormal basis for  $\mathbf{N}(G)$ . We have

$$\phi(x) = H(Bx + p^{(1)}),$$

where  $p^{(1)}$  is a fixed vector satisfying  $Gp^{(1)} = \begin{pmatrix} 1 \\ c \end{pmatrix}$ .

Observe that for any  $x \in \mathbb{R}^{N-r}$ ,  $Bx \in \mathbf{N}(G)$  so that  $G(Bx + p^{(1)}) = Gp^{(1)} = \begin{pmatrix} 1 \\ c \end{pmatrix}$  and so  $Bx + p^{(1)}$  is feasible. Moreover,  $B$  is a bijection from  $\mathbb{R}^{N-r}$  to the set of feasible solutions.<sup>7</sup> Importantly,  $\phi$  is now *unconstrained*, and so any saddle point (and so any maximum or minimum) must satisfy:

$$\nabla_x \phi(x) = 0$$

**Gradient Decomposition** Any critical point of  $H$  yields a critical point of  $\phi$ , that is, if  $p = p^{(0)} + p^{(1)}$  is a critical point of  $H$  then  $x = B^T p^{(0)}$  is a critical point of  $\phi$ . Consider any critical point  $p$ , then we can uniquely decompose the gradient as:

$$\nabla_p H(p) = g_0 + g_1 \text{ in which } g_0 \in \mathbf{N}(G) \text{ and } g_1 \in \mathbf{N}(G)^\perp.$$

---

<sup>6</sup>One can form the Lagrangian for non-linear constraints, but to derive it we need to use fancier math like the implicit function theorem. We only need linear constraints for our applications.

<sup>7</sup>For contradiction, suppose  $p, q$  are distinct feasible solutions then,  $p \neq q$  but  $B^T p = B^T q$  but we can write  $p = p^{(0)} + p^{(1)}$  and  $q = q^{(0)} + p^{(1)}$  from the above. However,  $B^T p = B^T q$  implies that  $B^T p^{(0)} = B^T q^{(0)}$ . In turn since  $B$  is a bijection on  $\mathbf{N}(G)$  this implies that  $p^{(0)} = q^{(0)}$ .

We claim  $g_0 = B\nabla\phi(B^T p)$  or equivalently  $B^T g = \nabla_x\phi(B^T p)$ . From direct calculation,  $\nabla_x\phi(x) = \nabla_x H(Bx + p^{(1)}) = B^T \nabla_p H(p^{(0)} + p^{(1)}) = B^T \nabla_p H(p) = B^T g_0$ , where the last equality is due to  $g_1 \in N(G)^\perp$ . A critical point of  $H$  satisfying the constraints must not change along any direction that satisfies the constraints, which is to say that we must have  $g_0 = 0$ . Very roughly, one can have the intuition that if  $p$  were a maximum (or minimum), then if  $g_0$  were non-zero there would be a way to strictly increase (or decrease) the function in a neighbor around  $p$ —contradicting  $p$  being a maximum (minimum).

**Lagrangian** Since  $g_1 \in N(G)^\perp = R(G^T)$  (see the fundamental theorem of linear algebra), we can find a  $\eta(p)$  such that  $g_1 = -G^T \eta(p)$ , which motivates the following functional form:

$$\mathcal{L}(p, \eta(p)) = H(p) + \langle \eta(p), Gp - c \rangle$$

By the definition of  $\eta(p)$ , we have:

$$\nabla_p \mathcal{L}(p, \eta(p)) = g_0 + g_1 + G^T \eta(p) = g_0.$$

That is, for any critical point  $p$  of the original function (which corresponds to  $g_0 = 0$ ) we can select  $\eta(p)$  so that it is a critical point of  $\mathcal{L}(p, \eta)$ . Informally, the multipliers combines the rows of  $G$  to cancel  $g_1$ , the component of the gradient in the direction of the constraints. This establishes that any critical point of the original constrained function is also a critical point of the Lagrangian.

## References

- [1] Jaynes, Edwin T, *Probability theory: The logic of science*, Cambridge University Press, 2003.

# CS229 Supplemental Lecture notes

John Duchi

## 1 General loss functions

Building off of our interpretations of supervised learning as (1) choosing a representation for our problem, (2) choosing a loss function, and (3) minimizing the loss, let us consider a slightly more general formulation for supervised learning. In the supervised learning settings we have considered thus far, we have input data  $x \in \mathbb{R}^n$  and targets  $y$  from a space  $\mathcal{Y}$ . In linear regression, this corresponded to  $y \in \mathbb{R}$ , that is,  $\mathcal{Y} = \mathbb{R}$ , for logistic regression and other binary classification problems, we had  $y \in \mathcal{Y} = \{-1, 1\}$ , and for multiclass classification we had  $y \in \mathcal{Y} = \{1, 2, \dots, k\}$  for some number  $k$  of classes.

For each of these problems, we made predictions based on  $\theta^T x$  for some vector  $\theta$ , and we constructed a loss function  $\mathsf{L} : \mathbb{R} \times \mathcal{Y} \rightarrow \mathbb{R}$ , where  $\mathsf{L}(\theta^T x, y)$  measures the loss we suffer for predicting  $\theta^T x$ . For logistic regression, we use the logistic loss

$$\mathsf{L}(z, y) = \log(1 + e^{-yz}) \quad \text{or} \quad \mathsf{L}(\theta^T x, y) = \log(1 + e^{-y\theta^T x}).$$

For linear regression we use the squared error

$$\mathsf{L}(z, y) = \frac{1}{2}(z - y)^2 \quad \text{or} \quad \mathsf{L}(\theta^T x, y) = \frac{1}{2}(\theta^T x - y)^2.$$

For multiclass classification, we had a slight variant, where we let  $\Theta = [\theta_1 \dots \theta_k]$  for  $\theta_i \in \mathbb{R}^n$ , and used the loss  $\mathsf{L} : \mathbb{R}^k \times \{1, \dots, k\} \rightarrow \mathbb{R}$

$$\mathsf{L}(z, y) = \log \left( \sum_{i=1}^k \exp(z_i - z_y) \right) \quad \text{or} \quad \mathsf{L}(\Theta^T x, y) = \log \left( \sum_{i=1}^k \exp(x^T(\theta_i - \theta_y)) \right),$$

the idea being that we wish to have  $\theta_y^T x > \theta_i^T x$  for all  $i \neq y$ . Given a training set of pairs  $\{x^{(i)}, y^{(i)}\}$ , choose  $\theta$  by minimizing the empirical risk

$$J(\theta) = \frac{1}{m} \sum_{i=1}^m \mathsf{L}(\theta^T x^{(i)}, y^{(i)}). \tag{1}$$

## 2 The representer theorem

Let us consider a slight variant of choosing  $\theta$  to minimize the risk (1). In many situations—for reasons that we will study more later in the class—it is useful to add *regularization* to the risk  $J$ . We add regularization for many reasons: often, it makes problem (1) easier to solve numerically, and also it can allow the  $\theta$  we get out of minimizing the risk (1) able to generalize better to unseen data. Generally, regularization is taken to be of the form  $r(\theta) = \|\theta\|$  or  $r(\theta) = \|\theta\|^2$  for some norm  $\|\cdot\|$  on  $\mathbb{R}^n$ . The most common choice is so-called  $\ell_2$ -regularization, in which we choose

$$r(\theta) = \frac{\lambda}{2} \|\theta\|_2^2,$$

where we recall that  $\|\theta\|_2 = \sqrt{\theta^T \theta}$  is the Euclidean norm, or length, of the vector  $\theta$ . This gives rise to the *regularized risk*, which is

$$J_\lambda(\theta) = \frac{1}{m} \sum_{i=1}^m \mathsf{L}(\theta^T x^{(i)}, y^{(i)}) + \frac{\lambda}{2} \|\theta\|_2^2. \quad (2)$$

Let us consider the structure of any  $\theta$  that minimizes the risk (2). We assume—as we often do—that for each fixed target value  $y \in \mathcal{Y}$ , the function  $\mathsf{L}(z, y)$  is convex in  $z$ . (This is the case for linear regression and binary and multiclass logistic regression, as well as a number of other losses we will consider.) It turns out that under these assumptions, we may always write the solutions to the problem (2) as a *linear combination* of the input variables  $x^{(i)}$ . More precisely, we have the following theorem, known as the *representer theorem*.

**Theorem 2.1.** *Suppose in the definition of the regularized risk (2) that  $\lambda \geq 0$ . Then there is a minimizer of the regularized risk (2) that can be written*

$$\theta = \sum_{i=1}^m \alpha_i x^{(i)}$$

*for some real-valued weights  $\alpha_i$ .*

**Proof** For intuition, we give a proof of the result in the case that  $\mathsf{L}(z, y)$ , when viewed as a function of  $z$ , is differentiable and  $\lambda > 0$ . In Appendix A,

we present a more general statement of the theorem as well as a rigorous proof.

Let  $L'(z, y) = \frac{\partial}{\partial z} L(z, y)$  denote the derivative of the loss with respect to  $z$ . Then by the chain rule, we have the gradient identity

$$\nabla_{\theta} L(\theta^T x, y) = L'(\theta^T x, y)x \quad \text{and} \quad \nabla_{\theta} \frac{1}{2} \|\theta\|_2^2 = \theta,$$

where  $\nabla_{\theta}$  denotes taking a gradient with respect to  $\theta$ . As the risk must have 0 gradient at all stationary points (including the minimizer), we can write

$$\nabla J_{\lambda}(\theta) = \frac{1}{m} \sum_{i=1}^m L'(\theta^T x^{(i)}, y^{(i)})x^{(i)} + \lambda\theta = \vec{0}.$$

In particular, letting  $w_i = L'(\theta^T x^{(i)}, y^{(i)})$ , as  $L'(\theta^T x^{(i)}, y^{(i)})$  is a scalar (which depends on  $\theta$ , but no matter what  $\theta$  is,  $w_i$  is still a real number), we have

$$\theta = -\frac{1}{\lambda} \sum_{i=1}^n w_i x^{(i)}.$$

Set  $\alpha_i = -\frac{w_i}{\lambda}$  to get the result. □

### 3 Nonlinear features and kernels

Based on the representer theorem, Theorem 2.1, we see that we can always write the vector  $\theta$  as a linear combination of the data  $\{x^{(i)}\}_{i=1}^m$ . Importantly, this means we can *always* make predictions

$$\theta^T x = x^T \theta = \sum_{i=1}^m \alpha_i x^T x^{(i)}.$$

That is, in *any* learning algorithm, we may can replace all appearances of  $\theta^T x$  with  $\sum_{i=1}^m \alpha_i x^{(i)T} x$ , and then minimize directly over  $\alpha \in \mathbb{R}^m$ .

Let us consider this idea in somewhat more generality. 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 regression 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  $\phi$  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 a learning algorithm 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  $\langle \phi(x), \phi(z) \rangle$ . Specifically, given a feature mapping  $\phi$ , 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  $\phi$ . Let us write this out more carefully. We saw by the representer theorem (Theorem 2.1) that we can write  $\theta = \sum_{i=1}^m \alpha_i \phi(x^{(i)})$  for some weights  $\alpha_i$ . Then we can write the (regularized) risk

$$\begin{aligned} J_\lambda(\theta) &= J_\lambda(\alpha) \\ &= \frac{1}{m} \sum_{i=1}^m \mathcal{L}\left(\phi(x^{(i)})^T \sum_{j=1}^m \alpha_j \phi(x^{(j)}), y^{(i)}\right) + \frac{\lambda}{2} \left\| \sum_{i=1}^m \alpha_i \phi(x^{(i)}) \right\|_2^2 \\ &= \frac{1}{m} \sum_{i=1}^m \mathcal{L}\left(\sum_{j=1}^m \alpha_j \phi(x^{(i)})^T \phi(x^{(j)}), y^{(i)}\right) + \frac{\lambda}{2} \sum_{i=1}^m \sum_{j=1}^m \alpha_i \alpha_j \phi(x^{(i)})^T \phi(x^{(j)}) \\ &= \frac{1}{m} \sum_{i=1}^m \mathcal{L}\left(\sum_{j=1}^m \alpha_j K(x^{(i)}, x^{(j)}), y^{(i)}\right) + \frac{\lambda}{2} \sum_{i,j} \alpha_i \alpha_j K(x^{(i)}, x^{(j)}). \end{aligned}$$

That is, we can write the entire loss function to be minimized in terms of the kernel matrix

$$K = [K(x^{(i)}, x^{(j)})]_{i,j=1}^m \in \mathbb{R}^{m \times m}.$$

Now, given  $\phi$ , 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 learn in the high dimensional feature space space given by  $\phi$ , but without ever having to explicitly find or represent vectors  $\phi(x)$ . As a few examples, some kernels (corresponding to infinite-dimensional vectors  $\phi$ ) include

$$K(x, z) = \exp\left(-\frac{1}{2\tau^2} \|x - z\|_2^2\right),$$

known as the Gaussian or Radial Basis Function (RBF) kernel and applicable to data in any dimension, or the min-kernel (applicable when  $x \in \mathbb{R}$ , defined by

$$K(x, z) = \min\{x, z\}.$$

See also the lecture notes on Support Vector Machines (SVMs) for more on these kernel machines.

## 4 Stochastic gradient descent for kernelized machine learning

If we let  $K \in \mathbb{R}^{m \times m}$  denote the kernel matrix, and for shorthand define the vectors

$$K^{(i)} = \begin{bmatrix} K(x^{(i)}, x^{(1)}) \\ K(x^{(i)}, x^{(2)}) \\ \vdots \\ K(x^{(i)}, x^{(m)}) \end{bmatrix},$$

so that  $K = [K^{(1)} \ K^{(2)} \ \dots \ K^{(m)}]$ , then we may write the regularized risk in a concise form as

$$J_\lambda(\alpha) = \frac{1}{m} \sum_{i=1}^m \mathsf{L}(K^{(i)T} \alpha, y^{(i)}) + \frac{\lambda}{2} \alpha^T K \alpha.$$

Now, let us consider taking a stochastic gradient of the above risk  $J_\lambda$ . That is, we wish to construct a (simple to compute) random vector whose expectation is  $\nabla J_\lambda(\alpha)$ , which does not have too much variance. To do so, we first compute the gradient of  $J_\lambda(\alpha)$  on its own. We calculate the gradient of individual loss terms by noting that

$$\nabla_\alpha \mathcal{L}(K^{(i)T} \alpha, y^{(i)}) = \mathcal{L}'(K^{(i)T} \alpha, y^{(i)}) K^{(i)},$$

while

$$\nabla_\alpha \left[ \frac{\lambda}{2} \alpha^T K \alpha \right] = \lambda K \alpha = \lambda \sum_{i=1}^m K^{(i)} \alpha_i.$$

Thus, we have

$$\nabla_\alpha J_\lambda(\alpha) = \frac{1}{m} \sum_{i=1}^m \mathcal{L}'(K^{(i)T} \alpha, y^{(i)}) K^{(i)} + \lambda \sum_{i=1}^m K^{(i)} \alpha_i.$$

Thus, if we choose a random index  $i \in \{1, \dots, m\}$ , we have that

$$\mathcal{L}'(K^{(i)T} \alpha, y^{(i)}) K^{(i)} + m\lambda K^{(i)} \alpha_i$$

is a stochastic gradient for  $J_\lambda(\alpha)$ . This gives us a stochastic gradient procedure for Kernel supervised learning problems, shown in Figure 1. One minor

**Input:** A loss function  $\mathcal{L}$ , kernel matrix  $K = [K^{(1)} \dots K^{(m)}]$ , and labels  $\{y^{(i)}\}_{i=1}^m$ , and sequence of positive stepsizes  $\eta_1, \eta_2, \eta_3, \dots$

**Iterate** for  $t = 1, 2, \dots$

- (i) Choose index  $i \in \{1, \dots, m\}$  uniformly at random
- (ii) Update

$$\alpha := \alpha - \eta_t \left[ \mathcal{L}'(K^{(i)T} \alpha, y^{(i)}) K^{(i)} + m\lambda K^{(i)} \alpha_i \right].$$

Figure 1: Stochastic gradient descent for kernel supervised learning problems.

remark is in order regarding Algorithm 1: because we multiply the  $\lambda K^{(i)} \alpha_i$  term by  $m$  to keep the gradient unbiased, it is important that  $\lambda > 0$  not be too large, as the algorithm can be a bit unstable otherwise. In addition, a common choice of stepsize is to use  $\eta_t = 1/\sqrt{t}$ , or a constant multiple thereof.

## 5 Support vector machines

Now we discuss (one approach) to Support Vector Machines (SVM)s, which apply to binary classification problems with labels  $y \in \{-1, 1\}$ , and a particular choice of loss function  $L$ . In particular, for the support vector machine, we use the margin-based loss function

$$L(z, y) = [1 - yz]_+ = \max\{0, 1 - yz\}. \quad (3)$$

So, in a sense, SVMs are nothing but a special case of the general theoretical results we have described above. In particular, we have the empirical regularized risk

$$J_\lambda(\alpha) = \frac{1}{m} \sum_{i=1}^m [1 - y^{(i)} K^{(i)T} \alpha]_+ + \frac{\lambda}{2} \alpha^T K \alpha,$$

where the matrix  $K = [K^{(1)} \dots K^{(m)}]$  is defined by  $K_{ij} = K(x^{(i)}, x^{(j)})$ .

In the lecture notes, you can see another way of deriving the support vector machine and a description of why we actually call them support vector machines.

## 6 An example

In this section, we consider a particular example kernel, known as the Gaussian or Radial Basis Function (RBF) kernel. This kernel is defined by

$$K(x, z) = \exp\left(-\frac{1}{2\tau^2} \|x - z\|_2^2\right), \quad (4)$$

where  $\tau > 0$  is a parameter controlling the *bandwidth* of the kernel. Intuitively, for  $\tau$  very small, we will have  $K(x, z) \approx 0$  unless  $x \approx z$ , that is,  $x$  and  $z$  are very close, in which case we have  $K(x, z) \approx 1$ . However, for  $\tau$  large, then we have a much smoother kernel function  $K$ . The feature function  $\phi$  for this kernel is infinite dimensional.<sup>1</sup> That said, it is possible to gain some

---

<sup>1</sup>If you have seen characteristic functions or Fourier transforms, then you might recognize the RBF kernel as the Fourier transform of the Gaussian distribution with mean zero and variance  $\tau^2$ . That is, in  $\mathbb{R}^n$ , let  $W \sim N(0, \tau^2 I_{n \times n})$ , so that  $W$  has density

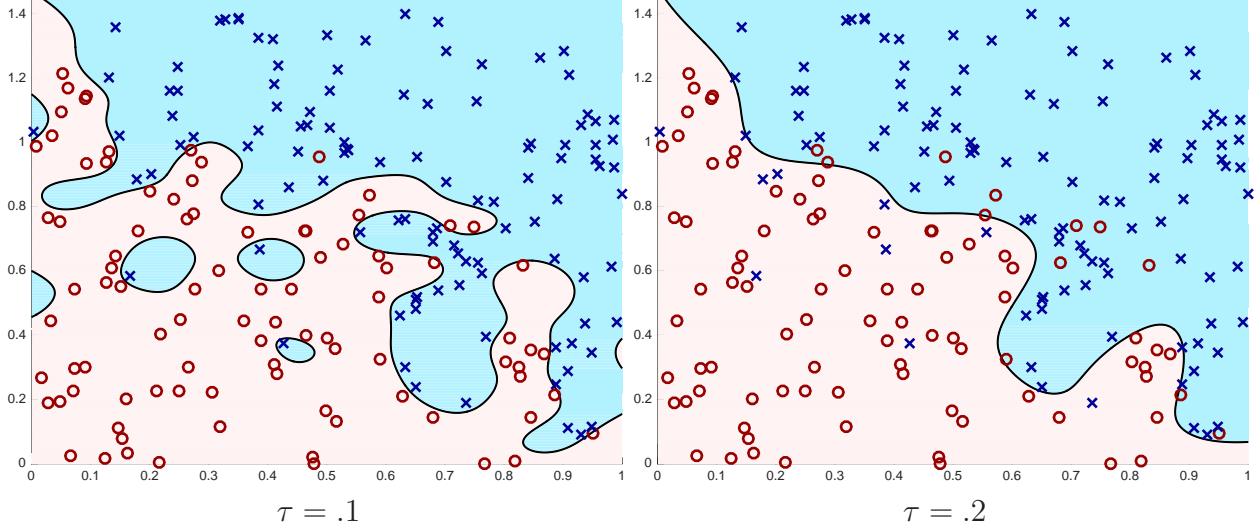


Figure 2: Small bandwidths  $\tau$  for Gaussian kernel  $K(x, z) = \exp\left(-\frac{1}{2\tau^2} \|x - z\|_2^2\right)$ .

intuition for the kernel by considering the classifications it makes on a new example  $x$ : we have prediction

$$\sum_{i=1}^m K(x^{(i)}, x)\alpha_i = \sum_{i=1}^m \exp\left(-\frac{1}{2\tau^2} \|x^{(i)} - x\|_2^2\right) \alpha_i,$$

---

$p(w) = \frac{1}{(2\pi\tau^2)^{n/2}} \exp\left(-\frac{\|w\|_2^2}{2\tau^2}\right)$ . Let  $i = \sqrt{-1}$  be the imaginary unit, then for any vector  $v$  we have

$$\begin{aligned} \mathbb{E}[\exp(iv^T W)] &= \int \exp(iv^T w)p(w)dw = \int \frac{1}{(2\pi\tau^2)^{n/2}} \exp\left(iv^T w - \frac{1}{2\tau^2} \|w\|_2^2\right) dw \\ &= \exp\left(-\frac{1}{2\tau^2} \|v\|_2^2\right). \end{aligned}$$

Thus, if we define the “vector” (really, function)  $\phi(x, w) = e^{ix^T w}$  and let  $a^*$  be the complex conjugate of  $a \in \mathbb{C}$ , then we have

$$\mathbb{E}[\phi(x, W)\phi(z, W)^*] = \mathbb{E}[e^{ix^T W} e^{-iz^T W}] = \mathbb{E}[\exp(iW^T(x - z))] = \exp\left(-\frac{1}{2\tau^2} \|x - z\|_2^2\right).$$

In particular, we see that  $K(x, z)$  is the inner product in a space of functions that are integrable against  $p(w)$ .

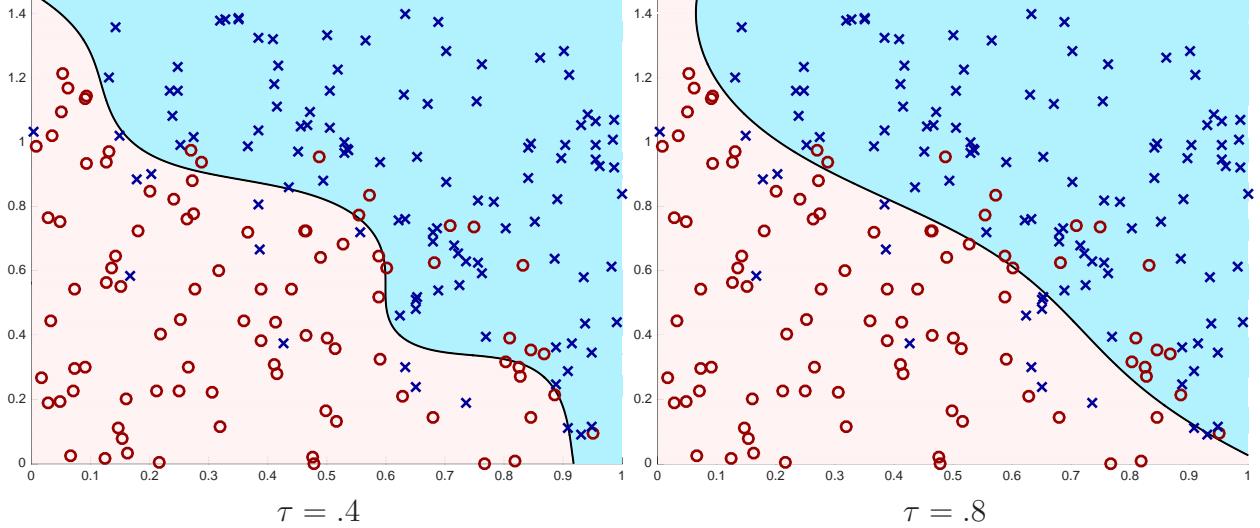


Figure 3: Medium bandwidths  $\tau$  for Gaussian kernel  $K(x, z) = \exp(-\frac{1}{2\tau^2} \|x - z\|_2^2)$ .

so that this becomes something like a weighting depending on *how close*  $x$  is to each  $x^{(i)}$ —that is, the contribution of weight  $\alpha_i$  is multiplied by the similarity of  $x$  to  $x^{(i)}$  as determined by the kernel function.

In Figures 2, 3, and 4, we show the results of training 6 different kernel classifiers by minimizing

$$J_\lambda(\alpha) = \sum_{i=1}^m \left[ 1 - y^{(i)} K^{(i)T} \alpha \right]_+ + \frac{\lambda}{2} \alpha^T K \alpha,$$

with  $m = 200$  and  $\lambda = 1/m$ , for different values of  $\tau$  in the kernel (4). We plot the training data (positive examples as blue x's and negative examples as red o's) as well as the decision surface of the resulting classifier. That is, we plot the lines defined by

$$\left\{ x \in \mathbb{R}^2 : \sum_{i=1}^m K(x, x^{(i)}) \alpha_i = 0 \right\},$$

giving the regions where the learned classifier makes a prediction  $\sum_{i=1}^m K(x, x^{(i)}) \alpha_i > 0$  and  $\sum_{i=1}^m K(x, x^{(i)}) \alpha_i < 0$ . From the figure, we see that for large  $\tau$ , we have

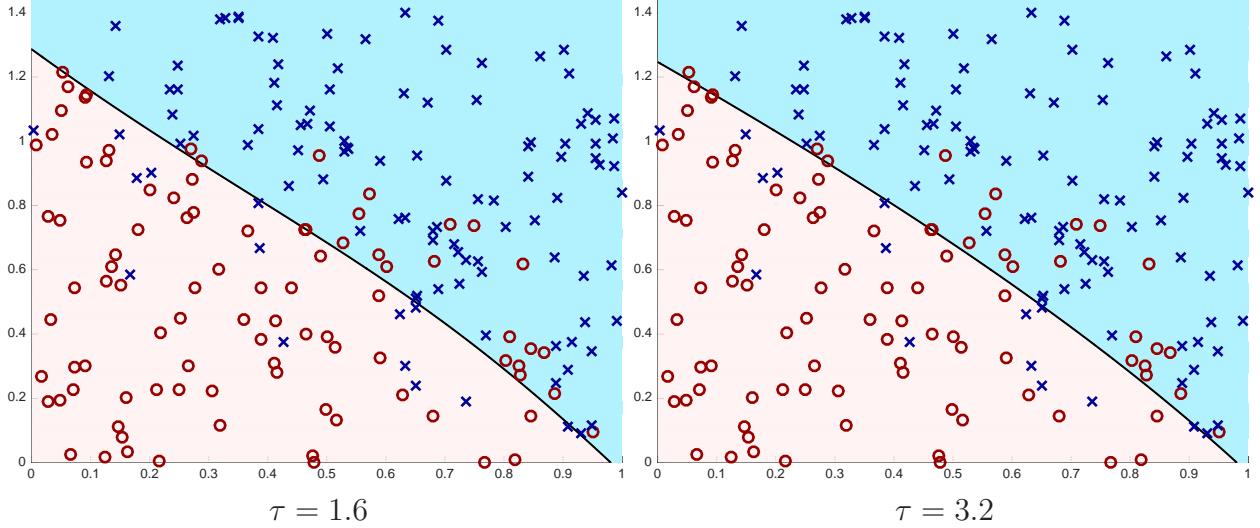


Figure 4: Large bandwidths  $\tau$  for Gaussian kernel  $K(x, z) = \exp(-\frac{1}{2\tau^2} \|x - z\|_2^2)$ .

a very simple classifier: it is nearly linear, while for  $\tau = .1$ , the classifier has substantial variability and is highly non-linear. For reference, in Figure 5, we plot the optimal classifier along with the training data; the optimal classifier minimizes the misclassification error given infinite training data.

## A A more general representer theorem

In this section, we present a more general version of the representer theorem along with a rigorous proof. Let  $r : \mathbb{R} \rightarrow \mathbb{R}$  be any non-decreasing function of its argument, and consider the regularized risk

$$J_r(\theta) = \frac{1}{m} \sum_{i=1}^m \mathsf{L}(x^{(i)T} \theta, y^{(i)}) + r(\|\theta\|_2). \quad (5)$$

Often, we take  $r(t) = \frac{\lambda}{2} t^2$ , which corresponds to the common choice of  $\ell_2$ -regularization, but the next theorem makes clear that this is not necessary for the representer theorem. Indeed, we could simply take  $r(t) = 0$  for all  $t$ , and the theorem still holds.

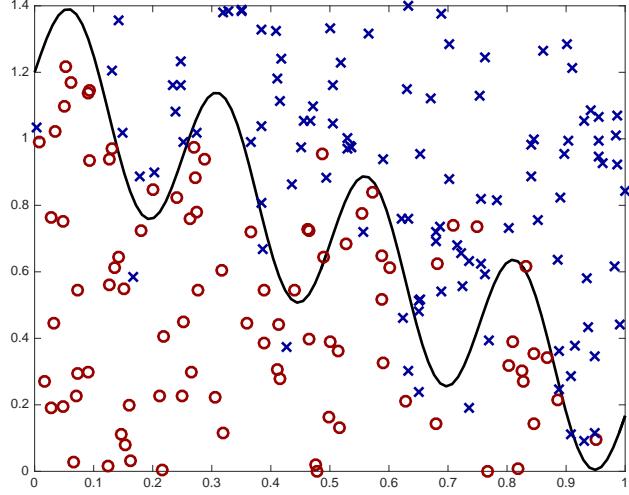


Figure 5: Optimal classifier along with training data.

**Theorem A.1** (Representer theorem in  $\mathbb{R}^n$ ). *Let  $\theta \in \mathbb{R}^n$  be any vector. Then there exists some  $\alpha \in \mathbb{R}^m$  and  $\theta^{(\alpha)} = \sum_{i=1}^m \alpha_i x^{(i)}$  such that*

$$J_r(\theta^{(\alpha)}) \leq J_r(\theta).$$

In particular, there is no loss of generality in always assuming we can write the optimization problem to minimize  $J(\theta)$  by only considering  $\theta$  in the span of the data.

**Proof** Our proof relies on some machinery from linear algebra, which allows us to keep it concise, but feel free to ask questions if it is too concise.

The vectors  $\{x^{(i)}\}_{i=1}^m$  are in  $\mathbb{R}^n$ , and as a consequence there is some subspace  $V$  of  $\mathbb{R}^n$  such that

$$V = \left\{ \sum_{i=1}^m \beta_i x^{(i)} : \beta_i \in \mathbb{R} \right\}.$$

Then  $V$  has an orthonormal basis  $\{v_1, \dots, v_{n_0}\}$  for vectors  $v_i \in \mathbb{R}^n$ , where the size (dimension) of the basis is  $n_0 \leq n$ . Thus we can write  $V = \{\sum_{i=1}^{n_0} b_i v_i : b_i \in \mathbb{R}\}$ , where we recall that orthonormality means that the vectors  $v_i$  satisfy  $\|v_i\|_2 = 1$  and  $v_i^T v_j = 0$  for  $i \neq j$ . There is also an orthogonal subspace  $V^\perp = \{u \in \mathbb{R}^n : u^T v = 0 \text{ for all } v \in V\}$ , which has an orthonormal

basis of size  $n_\perp = n - n_0 \geq 0$ , which we write as  $\{u_1, \dots, u_{n_\perp}\} \subset \mathbb{R}^n$ . By construction they satisfy  $u_i^T v_j = 0$  for all  $i, j$ .

Because  $\theta \in \mathbb{R}^n$ , we know that we can write it uniquely as

$$\theta = \sum_{i=1}^{n_0} \nu_i v_i + \sum_{i=1}^{n_\perp} \mu_i u_i, \quad \text{where } \nu_i \in \mathbb{R} \text{ and } \mu_i \in \mathbb{R},$$

and the values  $\mu, \nu$  are unique. Now, we know that by definition of the space  $V$  as the span of  $\{x^{(i)}\}_{i=1}^m$ , there exists  $\alpha \in \mathbb{R}^m$  such that

$$\sum_{i=1}^{n_0} \nu_i v_i = \sum_{i=1}^m \alpha_i x^{(i)},$$

so that we have

$$\theta = \sum_{i=1}^m \alpha_i x^{(i)} + \sum_{i=1}^{n_\perp} \mu_i u_i.$$

Define  $\theta^{(\alpha)} = \sum_{i=1}^m \alpha_i x^{(i)}$ . Now, for any data point  $x^{(j)}$ , we have

$$u_i^T x^{(j)} = 0 \quad \text{for all } i = 1, \dots, n_\perp,$$

so that  $u_i^T \theta^{(\alpha)} = 0$ . As a consequence, we have

$$\|\theta\|_2^2 = \left\| \theta^{(\alpha)} + \sum_{i=1}^{n_\perp} \mu_i u_i \right\|_2^2 = \left\| \theta^{(\alpha)} \right\|_2^2 + 2 \underbrace{\sum_{i=1}^{n_\perp} \mu_i u_i^T \theta^{(\alpha)}}_{=0} + \left\| \sum_{i=1}^{n_\perp} \mu_i u_i \right\|_2^2 \geq \left\| \theta^{(\alpha)} \right\|_2^2, \quad (6a)$$

and we also have

$$\theta^{(\alpha)T} x^{(i)} = \theta^T x^{(i)} \quad (6b)$$

for all points  $x^{(i)}$ .

That is, by using  $\|\theta\|_2 \geq \|\theta^{(\alpha)}\|_2$  and equality (6b), we have

$$\begin{aligned} J_r(\theta) &= \frac{1}{m} \sum_{i=1}^m \mathcal{L}(\theta^T x^{(i)}, y^{(i)}) + r(\|\theta\|_2) \stackrel{(6b)}{=} \frac{1}{m} \sum_{i=1}^m \mathcal{L}(\theta^{(\alpha)T} x^{(i)}, y^{(i)}) + r(\|\theta\|_2) \\ &\stackrel{(6a)}{\geq} \frac{1}{m} \sum_{i=1}^m \mathcal{L}(\theta^{(\alpha)T} x^{(i)}, y^{(i)}) + r(\|\theta^{(\alpha)}\|_2) \\ &= J_r(\theta^{(\alpha)}). \end{aligned}$$

This is the desired result.  $\square$

# CS 229, Summer 2020

## Problem Set 0: Linear Algebra, Multivariable Calculus, and Probability

---

**Due Wednesday, June 29 at 11:59 pm on Gradescope.**

**Notes:**

- (1) These questions require thought, but do not require long answers. Please be as concise as possible.
- (2) If you have a question about this homework, we encourage you to post your question on our Piazza forum at <https://piazza.com/stanford/summer2020/cs229>.
- (3) If you missed the first lecture or are unfamiliar with the collaboration or honor code policy, please read the policy before you start.
- (4) This specific homework is ***not graded***, but we encourage you to solve each of the problems to brush up on your linear algebra and probability. Some of them may even be useful for subsequent problem sets. It also serves as your introduction to using Gradescope for submissions. We strongly suggest you use LaTex to submit your psets (not only is it helpful for this class, but it is a good skill to learn). However, if you are scanning your document by cellphone, please use a scanning app such as CamScanner. There will not be any late days allowed for this particular assignment.

**Honor code:** We strongly encourage students to form study groups. Students may discuss and work on homework problems in groups. However, each student must write down the solutions independently, and without referring to written notes from the joint session. That being said, if students are submitting in a pair, they act as one unit - they may share resources (such as notes) with each other and write the solutions together. Note that both of the two students should fully understand all the answers in their submission, even though only one of them needs to write up a solution to a question. In other words, each student must understand the solution well enough in order to reconstruct it by him/herself. In addition, each student should write on the problem set the set of people with whom s/he collaborated. Further, because we occasionally reuse problem set questions from previous years, we expect students not to copy, refer to, or look at the solutions in preparing their answers. It is an honor code violation to intentionally refer to a previous year's solutions.

**1. [0 points] Gradients and Hessians**

Recall that a matrix  $A \in \mathbb{R}^{n \times n}$  is *symmetric* if  $A^T = A$ , that is,  $A_{ij} = A_{ji}$  for all  $i, j$ . Also recall the gradient  $\nabla f(x)$  of a function  $f : \mathbb{R}^n \rightarrow \mathbb{R}$ , which is the  $n$ -vector of partial derivatives

$$\nabla f(x) = \begin{bmatrix} \frac{\partial}{\partial x_1} f(x) \\ \vdots \\ \frac{\partial}{\partial x_n} f(x) \end{bmatrix} \quad \text{where } x = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}.$$

The hessian  $\nabla^2 f(x)$  of a function  $f : \mathbb{R}^n \rightarrow \mathbb{R}$  is the  $n \times n$  symmetric matrix of twice partial derivatives,

$$\nabla^2 f(x) = \begin{bmatrix} \frac{\partial^2}{\partial x_1^2} f(x) & \frac{\partial^2}{\partial x_1 \partial x_2} f(x) & \cdots & \frac{\partial^2}{\partial x_1 \partial x_n} f(x) \\ \frac{\partial^2}{\partial x_2 \partial x_1} f(x) & \frac{\partial^2}{\partial x_2^2} f(x) & \cdots & \frac{\partial^2}{\partial x_2 \partial x_n} f(x) \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2}{\partial x_n \partial x_1} f(x) & \frac{\partial^2}{\partial x_n \partial x_2} f(x) & \cdots & \frac{\partial^2}{\partial x_n^2} f(x) \end{bmatrix}.$$

- (a) Let  $f(x) = \frac{1}{2}x^T Ax + b^T x$ , where  $A$  is a symmetric matrix and  $b \in \mathbb{R}^n$  is a vector. What is  $\nabla f(x)$ ?
- (b) Let  $f(x) = g(h(x))$ , where  $g : \mathbb{R} \rightarrow \mathbb{R}$  is differentiable and  $h : \mathbb{R}^n \rightarrow \mathbb{R}$  is differentiable. What is  $\nabla f(x)$ ?
- (c) Let  $f(x) = \frac{1}{2}x^T Ax + b^T x$ , where  $A$  is symmetric and  $b \in \mathbb{R}^n$  is a vector. What is  $\nabla^2 f(x)$ ?
- (d) Let  $f(x) = g(a^T x)$ , where  $g : \mathbb{R} \rightarrow \mathbb{R}$  is continuously differentiable and  $a \in \mathbb{R}^n$  is a vector. What are  $\nabla f(x)$  and  $\nabla^2 f(x)$ ? (*Hint:* your expression for  $\nabla^2 f(x)$  may have as few as 11 symbols, including ' and parentheses.)

**2. [0 points] Positive definite matrices**

A matrix  $A \in \mathbb{R}^{n \times n}$  is *positive semi-definite* (PSD), denoted  $A \succeq 0$ , if  $A = A^T$  and  $x^T A x \geq 0$  for all  $x \in \mathbb{R}^n$ . A matrix  $A$  is *positive definite*, denoted  $A \succ 0$ , if  $A = A^T$  and  $x^T A x > 0$  for all  $x \neq 0$ , that is, all non-zero vectors  $x$ . The simplest example of a positive definite matrix is the identity  $I$  (the diagonal matrix with 1s on the diagonal and 0s elsewhere), which satisfies  $x^T I x = \|x\|_2^2 = \sum_{i=1}^n x_i^2$ .

- (a) Let  $z \in \mathbb{R}^n$  be an  $n$ -vector. Show that  $A = zz^T$  is positive semidefinite.
- (b) Let  $z \in \mathbb{R}^n$  be a *non-zero*  $n$ -vector. Let  $A = zz^T$ . What is the null-space of  $A$ ? What is the rank of  $A$ ?
- (c) Let  $A \in \mathbb{R}^{n \times n}$  be positive semidefinite and  $B \in \mathbb{R}^{m \times n}$  be arbitrary, where  $m, n \in \mathbb{N}$ . Is  $BAB^T$  PSD? If so, prove it. If not, give a counterexample with explicit  $A, B$ .

**3. [0 points] Eigenvectors, eigenvalues, and the spectral theorem**

The eigenvalues of an  $n \times n$  matrix  $A \in \mathbb{R}^{n \times n}$  are the roots of the characteristic polynomial  $p_A(\lambda) = \det(\lambda I - A)$ , which may (in general) be complex. They are also defined as the values  $\lambda \in \mathbb{C}$  for which there exists a vector  $x \in \mathbb{C}^n$  such that  $Ax = \lambda x$ . We call such a pair  $(x, \lambda)$  an *eigenvector, eigenvalue* pair. In this question, we use the notation  $\text{diag}(\lambda_1, \dots, \lambda_n)$  to denote the diagonal matrix with diagonal entries  $\lambda_1, \dots, \lambda_n$ , that is,

$$\text{diag}(\lambda_1, \dots, \lambda_n) = \begin{bmatrix} \lambda_1 & 0 & 0 & \cdots & 0 \\ 0 & \lambda_2 & 0 & \cdots & 0 \\ 0 & 0 & \lambda_3 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & \lambda_n \end{bmatrix}.$$

- (a) Suppose that the matrix  $A \in \mathbb{R}^{n \times n}$  is diagonalizable, that is,  $A = T\Lambda T^{-1}$  for an invertible matrix  $T \in \mathbb{R}^{n \times n}$ , where  $\Lambda = \text{diag}(\lambda_1, \dots, \lambda_n)$  is diagonal. Use the notation  $t^{(i)}$  for the columns of  $T$ , so that  $T = [t^{(1)} \ \dots \ t^{(n)}]$ , where  $t^{(i)} \in \mathbb{R}^n$ . Show that  $At^{(i)} = \lambda_i t^{(i)}$ , so that the eigenvalues/eigenvector pairs of  $A$  are  $(t^{(i)}, \lambda_i)$ .

A matrix  $U \in \mathbb{R}^{n \times n}$  is orthogonal if  $U^T U = I$ . The spectral theorem, perhaps one of the most important theorems in linear algebra, states that if  $A \in \mathbb{R}^{n \times n}$  is symmetric, that is,  $A = A^T$ , then  $A$  is *diagonalizable by a real orthogonal matrix*. That is, there are a diagonal matrix  $\Lambda \in \mathbb{R}^{n \times n}$  and orthogonal matrix  $U \in \mathbb{R}^{n \times n}$  such that  $U^T A U = \Lambda$ , or, equivalently,

$$A = U\Lambda U^T.$$

Let  $\lambda_i = \lambda_i(A)$  denote the  $i$ th eigenvalue of  $A$ .

- (b) Let  $A$  be symmetric. Show that if  $U = [u^{(1)} \ \dots \ u^{(n)}]$  is orthogonal, where  $u^{(i)} \in \mathbb{R}^n$  and  $A = U\Lambda U^T$ , then  $u^{(i)}$  is an eigenvector of  $A$  and  $Au^{(i)} = \lambda_i u^{(i)}$ , where  $\Lambda = \text{diag}(\lambda_1, \dots, \lambda_n)$ .
- (c) Show that if  $A$  is PSD, then  $\lambda_i(A) \geq 0$  for each  $i$ .

**4. [0 points] Probability and multivariate Gaussians**

Suppose  $X = (X_1, \dots, X_n)$  is sampled from a multivariate Gaussian distribution with mean  $\mu$  in  $\mathbb{R}^n$  and covariance  $\Sigma$  in  $S_+^n$  (i.e.  $\Sigma$  is positive semidefinite). This is commonly also written as  $X \sim \mathcal{N}(\mu, \Sigma)$ .

- (a) Describe the random variable  $Y = X_1 + X_2 + \dots + X_n$ . What is the mean and variance? Is this a well known distribution, and if so, which?
- (b) Now, further suppose that  $\Sigma$  is invertible. Find  $\mathbb{E}[X^T \Sigma^{-1} X]$ . (Hint: use the property of trace that  $x^T A x = \text{tr}(x^T A x)$ ).

# CS 229, Summer 2020

## Problem Set #1

---

**Due Monday, July 13 at 11:59 pm on Gradescope.**

**Notes:** (1) These questions require thought, but do not require long answers. Please be as concise as possible. (2) If you have a question about this homework, we encourage you to post your question on our Piazza forum, at <https://piazza.com/stanford/summer2020/cs229>. (3) This quarter, Summer 2020, students may submit in pairs. If you do so, make sure both names are attached to the Gradescope submission. However, students are not allowed to work with the same partner on more than one assignment. If you missed the first lecture or are unfamiliar with the collaboration or honor code policy, please read the policy on the course website before starting work. (4) For the coding problems, you may not use any libraries except those defined in the provided `environment.yml` file. In particular, ML-specific libraries such as scikit-learn are not permitted. (5) To account for late days, the due date is Monday, July 13 at 11:59 pm. If you submit after Monday, July 13 at 11:59 pm, you will begin consuming your late days. If you wish to submit on time, submit before Monday, July 13 at 11:59 pm.

All students must submit an electronic PDF version of the written questions. We highly recommend typesetting your solutions via L<sup>A</sup>T<sub>E</sub>X, and we will award one bonus point for typeset submissions. All students must also submit a zip file of their source code to Gradescope, which should be created using the `make_zip.py` script. You should make sure to (1) restrict yourself to only using libraries included in the `environment.yml` file, and (2) make sure your code runs without errors. Your submission may be evaluated by the auto-grader using a private test set, or used for verifying the outputs reported in the writeup.

### 1. [40 points] Linear Classifiers (logistic regression and GDA)

In this problem, we cover two probabilistic linear classifiers we have covered in class so far. First, a discriminative linear classifier: logistic regression. Second, a generative linear classifier: Gaussian discriminant analysis (GDA). Both the algorithms find a linear decision boundary that separates the data into two classes, but make different assumptions. Our goal in this problem is to get a deeper understanding of the similarities and differences (and, strengths and weaknesses) of these two algorithms.

For this problem, we will consider two datasets, along with starter codes provided in the following files:

- `src/linearclass/ds1_{train,valid}.csv`
- `src/linearclass/ds2_{train,valid}.csv`
- `src/linearclass/logreg.py`
- `src/linearclass/gda.py`

Each file contains  $n$  examples, one example  $(x^{(i)}, y^{(i)})$  per row. In particular, the  $i$ -th row contains columns  $x_1^{(i)} \in \mathbb{R}$ ,  $x_2^{(i)} \in \mathbb{R}$ , and  $y^{(i)} \in \{0, 1\}$ . In the subproblems that follow, we will investigate using logistic regression and Gaussian discriminant analysis (GDA) to perform binary classification on these two datasets.

(a) [10 points]

In lecture we saw the average empirical loss for logistic regression:

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

where  $y^{(i)} \in \{0, 1\}$ ,  $h_\theta(x) = g(\theta^T x)$  and  $g(z) = 1/(1 + e^{-z})$ .

Find the Hessian  $H$  of this function, and show that for any vector  $z$ , it holds true that

$$z^T H z \geq 0.$$

**Hint:** You may want to start by showing that  $\sum_i \sum_j z_i x_i x_j z_j = (z^T z)^2 \geq 0$ . Recall also that  $g'(z) = g(z)(1 - g(z))$ .

**Remark:** This is one of the standard ways of showing that the matrix  $H$  is positive semi-definite, written " $H \succeq 0$ ." This implies that  $J$  is convex, and has no local minima other than the global one. If you have some other way of showing  $H \succeq 0$ , you're also welcome to use your method instead of the one above.

(b) [5 points] **Coding problem.** Follow the instructions in `src/linearclass/logreg.py` to train a logistic regression classifier using Newton's Method. Starting with  $\theta = \vec{0}$ , run Newton's Method until the updates to  $\theta$  are small: Specifically, train until the first iteration  $k$  such that  $\|\theta_k - \theta_{k-1}\|_1 < \epsilon$ , where  $\epsilon = 1 \times 10^{-5}$ . Make sure to write your model's predicted probabilities on the validation set to the file specified in the code.

Include a plot of the **validation data** with  $x_1$  on the horizontal axis and  $x_2$  on the vertical axis. To visualize the two classes, use a different symbol for examples  $x^{(i)}$  with  $y^{(i)} = 0$  than for those with  $y^{(i)} = 1$ . On the same figure, plot the decision boundary found by logistic regression (i.e., line corresponding to  $p(y|x) = 0.5$ ).

- (c) [5 points] Recall that in GDA we model the joint distribution of  $(x, y)$  by the following equations:

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

where  $\phi$ ,  $\mu_0$ ,  $\mu_1$ , and  $\Sigma$  are the parameters of our model.

Suppose we have already fit  $\phi$ ,  $\mu_0$ ,  $\mu_1$ , and  $\Sigma$ , and now want to predict  $y$  given a new point  $x$ . To show that GDA results in a classifier that has a linear decision boundary, show the posterior distribution can be written as

$$p(y=1 | x; \phi, \mu_0, \mu_1, \Sigma) = \frac{1}{1 + \exp(-(\theta^T x + \theta_0))},$$

where  $\theta \in \mathbb{R}^d$  and  $\theta_0 \in \mathbb{R}$  are appropriate functions of  $\phi$ ,  $\Sigma$ ,  $\mu_0$ , and  $\mu_1$ .

- (d) [7 points] Given the dataset, we claim that the maximum likelihood estimates of the parameters are given by

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

The log-likelihood of the data is

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

By maximizing  $\ell$  with respect to the four parameters, prove that the maximum likelihood estimates of  $\phi$ ,  $\mu_0$ ,  $\mu_1$ , and  $\Sigma$  are indeed as given in the formulas above. (You may assume that there is at least one positive and one negative example, so that the denominators in the definitions of  $\mu_0$  and  $\mu_1$  above are non-zero.)

- (e) [5 points] **Coding problem.** In `src/linearclass/gda.py`, fill in the code to calculate  $\phi$ ,  $\mu_0$ ,  $\mu_1$ , and  $\Sigma$ , use these parameters to derive  $\theta$ , and use the resulting GDA model to make predictions on the validation set. Make sure to write your model's predictions on the validation set to the file specified in the code.

Include a plot of the **validation data** with  $x_1$  on the horizontal axis and  $x_2$  on the vertical axis. To visualize the two classes, use a different symbol for examples  $x^{(i)}$  with  $y^{(i)} = 0$  than for those with  $y^{(i)} = 1$ . On the same figure, plot the decision boundary found by GDA (i.e., line corresponding to  $p(y|x) = 0.5$ ).

- (f) [2 points] For Dataset 1, compare the validation set plots obtained in part (b) and part (e) from logistic regression and GDA respectively, and briefly comment on your observation in a couple of lines.
- (g) [5 points] Repeat the steps in part (b) and part (e) for Dataset 2. Create similar plots on the **validation set** of Dataset 2 and include those plots in your writeup.  
On which dataset does GDA seem to perform worse than logistic regression? Why might this be the case?
- (h) [1 points] For the dataset where GDA performed worse in parts (f) and (g), can you find a transformation of the  $x^{(i)}$ 's such that GDA performs significantly better? What might this transformation be?

## 2. [30 points] Incomplete, Positive-Only Labels

In this problem we will consider training binary classifiers in situations where we do not have full access to the labels. In particular, we consider a scenario, which is not too infrequent in real life, where we have labels only for a subset of the positive examples. All the negative examples and the rest of the positive examples are unlabelled.

We formalize the scenario as follows. Let  $\{(x^{(i)}, t^{(i)})\}_{i=1}^n$  be a standard dataset of i.i.d distributed examples. Here  $x^{(i)}$ 's are the inputs/features and  $t^{(i)}$  are the labels. Now consider the situation where  $t^{(i)}$ 's are not observed by us. Instead, we only observe the labels of some of the positive examples. Concretely, we assume that we observe  $y^{(i)}$ 's that are generated by

$$\begin{aligned}\forall x, \quad p(y^{(i)} = 1 | t^{(i)} = 1, x^{(i)} = x) &= \alpha, \\ \forall x, \quad p(y^{(i)} = 0 | t^{(i)} = 1, x^{(i)} = x) &= 1 - \alpha \\ \forall x, \quad p(y^{(i)} = 1 | t^{(i)} = 0, x^{(i)} = x) &= 0, \\ \forall x, \quad p(y^{(i)} = 0 | t^{(i)} = 0, x^{(i)} = x) &= 1\end{aligned}$$

where  $\alpha \in (0, 1)$  is some unknown scalar. In other words, if the unobserved “true” label  $t^{(i)}$  is 1, then with  $\alpha$  chance we observe a label  $y^{(i)} = 1$ . On the other hand, if the unobserved “true” label  $t^{(i)} = 0$ , then we always observe the label  $y^{(i)} = 0$ .

Our final goal in the problem is to construct a binary classifier  $h$  of the true label  $t$ , with only access to the partial label  $y$ . In other words, we want to construct  $h$  such that  $h(x^{(i)}) \approx p(t^{(i)} = 1 | x^{(i)})$  as closely as possible, using only  $x$  and  $y$ .

*Real world example:* Suppose we maintain a database of proteins which are involved in transmitting signals across membranes. Every example added to the database is involved in a signaling process, but there are many proteins involved in cross-membrane signaling which are missing from the database. It would be useful to train a classifier to identify proteins that should be added to the database. In our notation, each example  $x^{(i)}$  corresponds to a protein,  $y^{(i)} = 1$  if the protein is in the database and 0 otherwise, and  $t^{(i)} = 1$  if the protein is involved in a cross-membrane signaling process and thus should be added to the database, and 0 otherwise.

For the rest of the question, we will use the dataset and starter code provided in the following files:

- `src/posonly/{train,valid,test}.csv`
- `src/posonly/posonly.py`

Each file contains the following columns:  $x_1$ ,  $x_2$ ,  $y$ , and  $t$ . As in Problem 1, there is one example per row. The  $y^{(i)}$ 's are generated from the process defined above with some unknown  $\alpha$ .

### (a) [5 points] Coding problem: ideal (fully observed) case

First we will consider the hypothetical (and uninteresting) case, where we have access to the true  $t$ -labels for training. In `src/posonly/posonly.py`, write a logistic regression classifier that uses  $x_1$  and  $x_2$  as input features, and train it using the  $t$ -labels. We will ignore the  $y$ -labels for this part. Output the trained model's predictions on the `test` set to the file specified in the code.

Create a plot to visualize the test set with  $x_1$  on the horizontal axis and  $x_2$  on the vertical axis. Use different symbols for examples  $x^{(i)}$  with true label  $t^{(i)} = 1$  than those with  $t^{(i)} = 0$ . On the same figure, plot the decision boundary obtained by your model (i.e., line

corresponding to model's predicted probability = 0.5) in red color. Include this plot in your writeup.

(b) [5 points] **Coding problem: The naive method on partial labels**

We now consider the case where the  $t$ -labels are unavailable, so you only have access to the  $y$ -labels at training time. Extend your code in `src/posonly/posonly.py` to re-train the classifier (still using  $x_1$  and  $x_2$  as input features), but using the  $y$ -labels only. Output the predictions on the **test set** to the appropriate file (as described in the code comments).

Create a plot to visualize the test set with  $x_1$  on the horizontal axis and  $x_2$  on the vertical axis. Use different symbols for examples  $x^{(i)}$  with true label  $t^{(i)} = 1$  (even though we only used the  $y^{(i)}$  labels for training, use the true  $t^{(i)}$  labels for plotting) than those with  $t^{(i)} = 0$ . On the same figure, plot the decision boundary obtained by your model (i.e., line corresponding to model's predicted probability = 0.5) in red color. Include this plot in your writeup.

Note that the algorithm should learn a function  $h(\cdot)$  that approximately predicts the probability  $p(y^{(i)} = 1 | x^{(i)})$ . Also note that we expect it to perform poorly on predicting the probability of interest, namely  $p(t^{(i)} = 1 | x^{(i)})$ .

In the following sub-questions we will attempt to solve the problem with only partial observations. That is, we only have access to  $\{(x^{(i)}, y^{(i)})\}_{i=1}^n$ , and will try to predict  $p(t^{(i)} = 1 | x^{(i)})$ .

(c) [5 points] **Warm-up with Bayes rule**

Show that under our assumptions, for any  $i$ ,

$$p(t^{(i)} = 1 | y^{(i)} = 1, x^{(i)}) = 1 \quad (1)$$

That is, observing a positive partial label  $y^{(i)} = 1$  tells us for sure the hidden true label is 1. Use Bayes rule to derive this (an informal explanation will not earn credit).

(d) [5 points] Show that for any example, the probability that true label  $t^{(i)}$  is positive is  $1/\alpha$  times the probability that the partial label is positive. That is, show that

$$p(t^{(i)} = 1 | x^{(i)}) = \frac{1}{\alpha} \cdot p(y^{(i)} = 1 | x^{(i)}) \quad (2)$$

Note that the equation above suggests that if we know the value of  $\alpha$ , then we can convert a function  $h(\cdot)$  that approximately predicts the probability  $h(x^{(i)}) \approx p(y^{(i)} = 1 | x^{(i)})$  into a function that approximately predicts  $p(t^{(i)} = 1 | x^{(i)})$  by multiplying the factor  $1/\alpha$ .

(e) [5 points] **Estimating  $\alpha$**

The solution to estimate  $p(t^{(i)} | x^{(i)})$  outlined in the previous sub-question requires the knowledge of  $\alpha$  which we don't have. Now we will design a way to estimate  $\alpha$  based on the function  $h(\cdot)$  that approximately predicts  $p(y^{(i)} = 1 | x^{(i)})$  (which we obtained in part b).

To simplify the analysis, let's assume that we have magically obtained a function  $h(x)$  that perfectly predicts the value of  $p(y^{(i)} = 1 | x^{(i)})$ , that is,  $h(x^{(i)}) = p(y^{(i)} = 1 | x^{(i)})$ .

We make the crucial assumption that  $p(t^{(i)} = 1 | x^{(i)}) \in \{0, 1\}$ . This assumption means that the process of generating the “true” label  $t^{(i)}$  is a noise-free process. This assumption is not very unreasonable to make. Note, we are NOT assuming that the observed label  $y^{(i)}$  is noise-free, which would be an unreasonable assumption!

Now we will show that:

$$\alpha = \mathbb{E}[h(x^{(i)}) \mid y^{(i)} = 1] \quad (3)$$

To show this, prove that  $h(x^{(i)}) = \alpha$  when  $y^{(i)} = 1$ , and  $h(x^{(i)}) = 0$  when  $y^{(i)} = 0$ .

The above result motivates the following algorithm to estimate  $\alpha$  by estimating the RHS of the equation above using samples: Let  $V_+$  be the set of labeled (and hence positive) examples in the validation set  $V$ , given by  $V_+ = \{x^{(i)} \in V \mid y^{(i)} = 1\}$ .

Then we use

$$\alpha \approx \frac{1}{|V_+|} \sum_{x^{(i)} \in V_+} h(x^{(i)}).$$

to estimate  $\alpha$ . (You will be asked to implement this algorithm in the next sub-question. For this sub-question, you only need to show equation (3). Moreover, this sub-question may be slightly harder than other sub-questions.)

(f) [5 points] **Coding problem.**

Using the validation set, estimate the constant  $\alpha$  by averaging your classifier's predictions over all labeled examples in the validation set:<sup>1</sup>

$$\alpha \approx \frac{1}{|V_+|} \sum_{x^{(i)} \in V_+} h(x^{(i)}).$$

Add code in `src/posonly/posonly.py` to rescale your predictions  $h(y^{(i)} = 1 \mid x^{(i)})$  of the classifier that is obtained from part b, using the equation (2) obtained in part (d) and using the estimated value for  $\alpha$ .

Finally, create a plot to visualize the test set with  $x_1$  on the horizontal axis and  $x_2$  on the vertical axis. Use different symbols for examples  $x^{(i)}$  with true label  $t^{(i)} = 1$  (even though we only used the  $y^{(i)}$  labels for training, use the true  $t^{(i)}$  labels for plotting) than those with  $t^{(i)} = 0$ . On the same figure, plot the decision boundary obtained by your model (i.e., line corresponding to model's **adjusted** predicted probability = 0.5) in red color. Include this plot in your writeup.

**Remark:** We saw that the true probability  $p(t \mid x)$  was only a constant factor away from  $p(y \mid x)$ . This means, if our task is to only rank examples (*i.e.* sort them) in a particular order (e.g, sort the proteins in order of being most likely to be involved in transmitting signals across membranes), then in fact we do not even need to estimate  $\alpha$ . The rank based on  $p(y \mid x)$  will agree with the rank based on  $p(t \mid x)$ .

---

<sup>1</sup>There is a reason to use the validation set, instead of the training set, to estimate the  $\alpha$ . However, for the purpose of this question, we sweep the subtlety here under the rug, and you don't need to understand the difference between the two for this question.

### 3. [25 points] Poisson Regression

In this question we will construct another kind of a commonly used GLM, which is called Poisson Regression. In a GLM, the choice of the exponential family distribution is based on the kind of problem at hand. If we are solving a classification problem, then we use an exponential family distribution with support over discrete classes (such as Bernoulli, or Categorical). Similarly, if the output is real valued, we can use Gaussian or Laplace (both are in the exponential family). Sometimes the desired output is to predict counts, for e.g., predicting the number of emails expected in a day, or the number of customers expected to enter a store in the next hour, etc. based on input features (also called covariates). You may recall that a probability distribution with support over integers (i.e. counts) is the Poisson distribution, and it also happens to be in the exponential family.

In the following sub-problems, we will start by showing that the Poisson distribution is in the exponential family, derive the functional form of the hypothesis, derive the update rules for training models, and finally using the provided dataset train a real model and make predictions on the test set.

- (a) [5 points] Consider the Poisson distribution parameterized by  $\lambda$ :

$$p(y; \lambda) = \frac{e^{-\lambda} \lambda^y}{y!}.$$

(Here  $y$  has positive integer values and  $y!$  is the factorial of  $y$ . ) Show that the Poisson distribution is in the exponential family, and clearly state the values for  $b(y)$ ,  $\eta$ ,  $T(y)$ , and  $a(\eta)$ .

- (b) [3 points] Consider performing regression using a GLM model with a Poisson response variable. What is the canonical response function for the family? (You may use the fact that a Poisson random variable with parameter  $\lambda$  has mean  $\lambda$ .)
- (c) [7 points] For a training set  $\{(x^{(i)}, y^{(i)}); i = 1, \dots, n\}$ , let the log-likelihood of an example be  $\log p(y^{(i)}|x^{(i)}; \theta)$ . By taking the derivative of the log-likelihood with respect to  $\theta_j$ , derive the stochastic gradient ascent update rule for learning using a GLM model with Poisson responses  $y$  and the canonical response function.

- (d) [10 points] **Coding problem**

Consider a website that wants to predict its daily traffic. The website owners have collected a dataset of past traffic to their website, along with some features which they think are useful in predicting the number of visitors per day. The dataset is split into train/valid sets and the starter code is provided in the following files:

- `src/poisson/{train,valid}.csv`
- `src/poisson/poisson.py`

We will apply Poisson regression to model the number of visitors per day. Note that applying Poisson regression in particular assumes that the data follows a Poisson distribution whose natural parameter is a linear combination of the input features (*i.e.*,  $\eta = \theta^T x$ ). In `src/poisson/poisson.py`, implement Poisson regression for this dataset and use *full batch gradient ascent* to maximize the log-likelihood of  $\theta$ . For the stopping criterion, check if the change in parameters has a norm smaller than a small value such as  $10^{-5}$ .

Using the trained model, predict the expected counts for the **validation set**, and create a scatter plot between the true counts vs predicted counts (on the validation set). In the

scatter plot, let x-axis be the true count and y-axis be the corresponding predicted expected count. Note that the true counts are integers while the expected counts are generally real values.

#### 4. [15 points] Convexity of Generalized Linear Models

In this question we will explore and show some nice properties of Generalized Linear Models, specifically those related to its use of Exponential Family distributions to model the output.

Most commonly, GLMs are trained by using the negative log-likelihood (NLL) as the loss function. This is mathematically equivalent to Maximum Likelihood Estimation (*i.e.*, maximizing the log-likelihood is equivalent to minimizing the negative log-likelihood). In this problem, our goal is to show that the NLL loss of a GLM is a *convex* function w.r.t the model parameters. As a reminder, this is convenient because a convex function is one for which any local minimum is also a global minimum, and there is extensive research on how to optimize various types of convex functions efficiently with various algorithms such as gradient descent or stochastic gradient descent.

To recap, an exponential family distribution is one whose probability density can be represented

$$p(y; \eta) = b(y) \exp(\eta^T T(y) - a(\eta)),$$

where  $\eta$  is the *natural parameter* of the distribution. Moreover, in a Generalized Linear Model,  $\eta$  is modeled as  $\theta^T x$ , where  $x \in \mathbb{R}^d$  are the input features of the example, and  $\theta \in \mathbb{R}^d$  are learnable parameters. In order to show that the NLL loss is convex for GLMs, we break down the process into sub-parts, and approach them one at a time. Our approach is to show that the second derivative (*i.e.*, Hessian) of the loss w.r.t the model parameters is Positive Semi-Definite (PSD) at all values of the model parameters. We will also show some nice properties of Exponential Family distributions as intermediate steps.

For the sake of convenience we restrict ourselves to the case where  $\eta$  is a scalar. Assume  $p(Y|X; \theta) \sim \text{ExponentialFamily}(\eta)$ , where  $\eta \in \mathbb{R}$  is a scalar, and  $T(y) = y$ . This makes the exponential family representation take the form

$$p(y; \eta) = b(y) \exp(\eta y - a(\eta)).$$

- (a) [5 points] Derive an expression for the mean of the distribution. Show that  $\mathbb{E}[Y; \eta] = \frac{\partial}{\partial \eta} a(\eta)$  (note that  $\mathbb{E}[Y; \eta] = \mathbb{E}[Y|X; \theta]$  since  $\eta = \theta^T x$ ). In other words, show that the mean of an exponential family distribution is the first derivative of the log-partition function with respect to the natural parameter.

**Hint:** Start with observing that  $\frac{\partial}{\partial \eta} \int p(y; \eta) dy = \int \frac{\partial}{\partial \eta} p(y; \eta) dy$ .

- (b) [5 points] Next, derive an expression for the variance of the distribution. In particular, show that  $\text{Var}(Y; \eta) = \frac{\partial^2}{\partial \eta^2} a(\eta)$  (again, note that  $\text{Var}(Y; \eta) = \text{Var}(Y|X; \theta)$ ). In other words, show that the variance of an exponential family distribution is the second derivative of the log-partition function w.r.t. the natural parameter.

**Hint:** Building upon the result in the previous sub-problem can simplify the derivation.

- (c) [5 points] Finally, write out the loss function  $\ell(\theta)$ , the NLL of the distribution, as a function of  $\theta$ . Then, calculate the Hessian of the loss w.r.t  $\theta$ , and show that it is always PSD. This concludes the proof that NLL loss of GLM is convex.

**Hint 1:** Use the chain rule of calculus along with the results of the previous parts to simplify your derivations.

**Hint 2:** Recall that variance of any probability distribution is non-negative.

**Remark:** The main takeaways from this problem are:

- Any GLM model is convex in its model parameters.

- The exponential family of probability distributions are mathematically nice. Whereas calculating mean and variance of distributions in general involves integrals (hard), surprisingly we can calculate them using derivatives (easy) for exponential family.

### 5. [25 points] Linear regression: linear in what?

In the first two lectures, you have seen how to fit a linear function of the data for the regression problem. In this question, we will see how linear regression can be used to fit non-linear functions of the data using feature maps. We will also explore some of its limitations, for which future lectures will discuss fixes.

#### (a) [5 points] Learning degree-3 polynomials of the input

Suppose we have a dataset  $\{(x^{(i)}, y^{(i)})\}_{i=1}^n$  where  $x^{(i)}, y^{(i)} \in \mathbb{R}$ . We would like to fit a third degree polynomial  $h_\theta(x) = \theta_3x^3 + \theta_2x^2 + \theta_1x^1 + \theta_0$  to the dataset. The key observation here is that the function  $h_\theta(x)$  is still linear in the unknown parameter  $\theta$ , even though it's not linear in the input  $x$ . This allows us to convert the problem into a linear regression problem as follows.

Let  $\phi : \mathbb{R} \rightarrow \mathbb{R}^4$  be a function that transforms the original input  $x$  to a 4-dimensional vector defined as

$$\phi(x) = \begin{bmatrix} 1 \\ x \\ x^2 \\ x^3 \end{bmatrix} \in \mathbb{R}^4 \quad (4)$$

Let  $\hat{x} \in \mathbb{R}^4$  be a shorthand for  $\phi(x)$ , and let  $\hat{x}^{(i)} \triangleq \phi(x^{(i)})$  be the transformed input in the training dataset. We construct a new dataset  $\{(\phi(x^{(i)}), y^{(i)})\}_{i=1}^n = \{(\hat{x}^{(i)}, y^{(i)})\}_{i=1}^n$  by replacing the original inputs  $x^{(i)}$ 's by  $\hat{x}^{(i)}$ 's. We see that fitting  $h_\theta(x) = \theta_3x^3 + \theta_2x^2 + \theta_1x^1 + \theta_0$  to the old dataset is equivalent to fitting a linear function  $h_\theta(\hat{x}) = \theta_3\hat{x}_3 + \theta_2\hat{x}_2 + \theta_1\hat{x}_1 + \theta_0$  to the new dataset because

$$h_\theta(x) = \theta_3x^3 + \theta_2x^2 + \theta_1x^1 + \theta_0 = \theta_3\phi(x)_3 + \theta_2\phi(x)_2 + \theta_1\phi(x)_1 + \theta_0 = \theta^T\hat{x} \quad (5)$$

In other words, we can use linear regression on the new dataset to find parameters  $\theta_0, \dots, \theta_3$ . Please write down 1) the objective function  $J(\theta)$  of the linear regression problem on the new dataset  $\{(\hat{x}^{(i)}, y^{(i)})\}_{i=1}^n$  and 2) the update rule of the batch gradient descent algorithm for linear regression on the dataset  $\{(\hat{x}^{(i)}, y^{(i)})\}_{i=1}^n$ .

*Terminology:* In machine learning,  $\phi$  is often called the feature map which maps the original input  $x$  to a new set of variables. To distinguish between these two sets of variables, we will call  $x$  the input **attributes**, and call  $\phi(x)$  the **features**. (Unfortunately, different authors use different terms to describe these two things. In this course, we will do our best to follow the above convention consistently.)

#### (b) [5 points] Coding question: degree-3 polynomial regression

For this sub-question question, we will use the dataset provided in the following files:

```
src/featuremaps/{train,valid,test}.csv
```

Each file contains two columns:  $x$  and  $y$ . In the terminology described in the introduction,  $x$  is the attribute (in this case one dimensional) and  $y$  is the output label.

Using the formulation of the previous sub-question, implement linear regression with **normal equations** using the feature map of degree-3 polynomials. Use the starter code provided in `src/featuremaps/featuremap.py` to implement the algorithm.

Create a scatter plot of the training data, and plot the learnt hypothesis as a smooth curve over it. Submit the plot in the writeup as the solution for this problem.

*Remark:* Suppose  $\hat{X}$  is the design matrix of the transformed dataset. You may sometimes encounter a non-invertible matrix  $\hat{X}^T \hat{X}$ . For a numerically stable code implementation, always use `np.linalg.solve` to obtain the parameters directly, rather than explicitly calculating the inverse and then multiplying it with  $\hat{X}^T y$ .

(c) [5 points] **Coding question: degree- $k$  polynomial regression**

Now we extend the idea above to degree- $k$  polynomials by considering  $\phi : \mathbb{R} \rightarrow \mathbb{R}^{k+1}$  to be

$$\phi(x) = \begin{bmatrix} 1 \\ x \\ x^2 \\ \vdots \\ x^k \end{bmatrix} \in \mathbb{R}^{k+1} \quad (6)$$

Follow the same procedure as the previous sub-question, and implement the algorithm with  $k = 3, 5, 10, 20$ . Create a similar plot as in the previous sub-question, and include the hypothesis curves for each value of  $k$  with a different color. Include a legend in the plot to indicate which color is for which value of  $k$ .

Submit the plot in the writeup as the solution for this sub-problem. Observe how the fitting of the training dataset changes as  $k$  increases. Briefly comment on your observations in the plot.

(d) [5 points] **Coding question: other feature maps**

You may have observed that it requires a relatively high degree  $k$  to fit the given training data, and this is because the dataset cannot be explained (i.e., approximated) very well by low-degree polynomials. By visualizing the data, you may have realized that  $y$  can be approximated well by a sine wave. In fact, we generated the data by sampling from  $y = \sin(x) + \xi$ , where  $\xi$  is noise with Gaussian distribution. Please update the feature map  $\phi$  to include a sine transformation as follows:

$$\phi(x) = \begin{bmatrix} 1 \\ x \\ x^2 \\ \vdots \\ x^k \\ \sin(x) \end{bmatrix} \in \mathbb{R}^{k+2} \quad (7)$$

With the updated feature map, train different models for values of  $k = 0, 1, 2, 3, 5, 10, 20$ , and plot the resulting hypothesis curves over the data as before.

Submit the plot as a solution to this sub-problem. Compare the fitted models with the previous sub-question, and briefly comment about noticeable differences in the fit with this feature map.

(e) [5 points] **Overfitting with expressive models and small data**

For the rest of the problem, we will consider a small dataset (a random subset of the dataset you have been using so far) with much fewer examples, provided in the following file:

`src/featuremaps/small.csv`

We will be exploring what happens when the number of features start becoming bigger than the number of examples in the training set. Run your algorithm on this small dataset using the following feature map

$$\phi(x) = \begin{bmatrix} 1 \\ x \\ x^2 \\ \vdots \\ x^k \end{bmatrix} \in \mathbb{R}^{k+1} \quad (8)$$

with  $k = 1, 2, 5, 10, 20$ .

Create a plot of the various hypothesis curves (just like previous sub-questions). Observe how the fitting of the training dataset changes as  $k$  increases. Submit the plot in the writeup and comment on what you observe.

**Remark:** The phenomenon you observe where the models start to fit the training dataset very well, but suddenly “goes wild” is due to what is called *overfitting*. The intuition to have for now is that, when the amount of data you have is small relative to the expressive capacity of the family of possible models (that is, the hypothesis class, which, in this case, is the family of all degree  $k$  polynomials), it results in overfitting.

Loosely speaking, the set of hypothesis function is “very flexible” and can be easily forced to pass through all your data points especially in unnatural ways. In other words, the model explains the noises in the training dataset, which shouldn’t be explained in the first place. This hurts the predictive power of the model on test examples. We will describe overfitting in more detail in future lectures when we cover learning theory and bias-variance tradeoffs.

# CS 229, Summer 2020

## Problem Set #2

---

**Due Monday, July 27 at 11:59 pm on Gradescope.**

**Notes:** (1) These questions require thought, but do not require long answers. Please be as concise as possible. (2) If you have a question about this homework, we encourage you to post your question on our Piazza forum, at <https://piazza.com/stanford/summer2020/cs229>. (3) This quarter, Summer 2020, students may submit in pairs. If you do so, make sure both names are attached to the Gradescope submission. However, students are not allowed to work with the same partner on more than one assignment. If you missed the first lecture or are unfamiliar with the collaboration or honor code policy, please read the policy on the course website before starting work. (4) For the coding problems, you may not use any libraries except those defined in the provided `environment.yml` file. In particular, ML-specific libraries such as scikit-learn are not permitted. (5) To account for late days, the due date is Monday, July 27 at 11:59 pm. If you submit after Monday, July 27 at 11:59 pm, you will begin consuming your late days. If you wish to submit on time, submit before Monday, July 27 at 11:59 pm.

All students must submit an electronic PDF version of the written questions. We highly recommend typesetting your solutions via L<sup>A</sup>T<sub>E</sub>X, and we will award one bonus point for typeset submissions. All students must also submit a zip file of their source code to Gradescope, which should be created using the `make_zip.py` script. You should make sure to (1) restrict yourself to only using libraries included in the `environment.yml` file, and (2) make sure your code runs without errors. Your submission may be evaluated by the auto-grader using a private test set, or used for verifying the outputs reported in the writeup.

**1. [15 points] Logistic Regression: Training stability**

In this problem, we will be delving deeper into the workings of logistic regression. The goal of this problem is to help you develop your skills debugging machine learning algorithms (which can be very different from debugging software in general).

We have provided an implementation of logistic regression in `src/stability/stability.py`, and two labeled datasets  $A$  and  $B$  in `src/stability/ds1_a.csv` and `src/stability/ds1_b.csv`.

Please do not modify the code for the logistic regression training algorithm for this problem. First, run the given logistic regression code to train two different models on  $A$  and  $B$ . You can run the code by simply executing `python stability.py` in the `src/stability` directory.

- (a) [2 points] What is the most notable difference in training the logistic regression model on datasets  $A$  and  $B$ ?

- (b) [5 points] Investigate why the training procedure behaves unexpectedly on dataset  $B$ , but not on  $A$ . Provide hard evidence (in the form of math, code, plots, etc.) to corroborate your hypothesis for the misbehavior. Remember, you should address why your explanation does *not* apply to  $A$ .

**Hint:** The issue is not a numerical rounding or over/underflow error.

- (c) [5 points] For each of these possible modifications, state whether or not it would lead to the provided training algorithm converging on datasets such as  $B$ . Justify your answers.

- i. Using a different constant learning rate.
- ii. Decreasing the learning rate over time (e.g. scaling the initial learning rate by  $1/t^2$ , where  $t$  is the number of gradient descent iterations thus far).
- iii. Linear scaling of the input features.
- iv. Adding a regularization term  $\|\theta\|_2^2$  to the loss function.
- v. Adding zero-mean Gaussian noise to the training data or labels.

- (d) [3 points] Are support vector machines vulnerable to datasets like  $B$ ? Why or why not? Give an informal justification.

## 2. [22 points] Spam classification

In this problem, we will use the naive Bayes algorithm and an SVM to build a spam classifier.

In recent years, spam on electronic media has been a growing concern. Here, we'll build a classifier to distinguish between real messages, and spam messages. For this class, we will be building a classifier to detect SMS spam messages. We will be using an SMS spam dataset developed by Tiago A. Almedia and José María Gómez Hidalgo which is publicly available on <http://www.dt.fee.unicamp.br/~tiago/smsspamcollection><sup>1</sup>

We have split this dataset into training and testing sets and have included them in this assignment as `src/spam/spam_train.tsv` and `src/spam/spam_test.tsv`. See `src/spam/spam_readme.txt` for more details about this dataset. Please refrain from redistributing these dataset files. The goal of this assignment is to build a classifier from scratch that can tell the difference the spam and non-spam messages using the text of the SMS message.

- (a) [5 points] Implement code for processing the the spam messages into numpy arrays that can be fed into machine learning models. Do this by completing the `get_words`, `create_dictionary`, and `transform_text` functions within our provided `src/spam.py`. Do note the corresponding comments for each function for instructions on what specific processing is required. The provided code will then run your functions and save the resulting dictionary into `spam_dictionary` and a sample of the resulting training matrix into `spam_sample_train_matrix`.

In your writeup, report the vocabulary size after the pre-processing step. You do not need to include any other output for this subquestion.

- (b) [10 points] In this question you are going to implement a naive Bayes classifier for spam classification with **multinomial event model** and Laplace smoothing.

Code your implementation by completing the `fit_naive_bayes_model` and `predict_from_naive_bayes_model` functions in `src/spam/spam.py`.

Now `src/spam/spam.py` should be able to train a Naive Bayes model, compute your prediction accuracy and then save your resulting predictions to `spam_naive_bayes_predictions`. In your writeup, report the accuracy of the trained model on the `test set`.

**Remark.** If you implement naive Bayes the straightforward way, you will find that the computed  $p(x|y) = \prod_i p(x_i|y)$  often equals zero. This is because  $p(x|y)$ , which is the product of many numbers less than one, is a very small number. The standard computer representation of real numbers cannot handle numbers that are too small, and instead rounds them off to zero. (This is called “underflow.”) You’ll have to find a way to compute Naive Bayes’ predicted class labels without explicitly representing very small numbers such as  $p(x|y)$ . [**Hint:** Think about using logarithms.]

- (c) [5 points] Intuitively, some tokens may be particularly indicative of an SMS being in a particular class. We can try to get an informal sense of how indicative token  $i$  is for the SPAM class by looking at:

$$\log \frac{p(x_j = i | y = 1)}{p(x_j = i | y = 0)} = \log \left( \frac{P(\text{token } i | \text{email is SPAM})}{P(\text{token } i | \text{email is NOTSPAM})} \right).$$

Complete the `get_top_five_naive_bayes_words` function within the provided code using the above formula in order to obtain the 5 most indicative tokens.

---

<sup>1</sup>Almeida, T.A., Gómez Hidalgo, J.M., Yamakami, A. Contributions to the Study of SMS Spam Filtering: New Collection and Results. Proceedings of the 2011 ACM Symposium on Document Engineering (DOCENG'11), Mountain View, CA, USA, 2011.

Report the top five words in your writeup.

- (d) [2 points] Support vector machines (SVMs) are an alternative machine learning model that we discussed in class. We have provided you an SVM implementation (using a radial basis function (RBF) kernel) within `src/spam/svm.py` (You should not need to modify that code).

One important part of training an SVM parameterized by an RBF kernel (a.k.a Gaussian kernel) is choosing an appropriate kernel radius parameter.

Complete the `compute_best_svm_radius` by writing code to compute the best SVM radius which maximizes accuracy on the validation dataset. Report the best kernel radius you obtained in the writeup.

### 3. [18 points] Constructing kernels

In class, we saw that by choosing a kernel  $K(x, z) = \phi(x)^T \phi(z)$ , we can implicitly map data to a high dimensional space, and have a learning algorithm (e.g SVM or logistic regression) work in that space. One way to generate kernels is to explicitly define the mapping  $\phi$  to a higher dimensional space, and then work out the corresponding  $K$ .

However in this question we are interested in direct construction of kernels. I.e., suppose we have a function  $K(x, z)$  that we think gives an appropriate similarity measure for our learning problem, and we are considering plugging  $K$  into the SVM as the kernel function. However for  $K(x, z)$  to be a valid kernel, it must correspond to an inner product in some higher dimensional space resulting from some feature mapping  $\phi$ . Mercer's theorem tells us that  $K(x, z)$  is a (Mercer) kernel if and only if for any finite set  $\{x^{(1)}, \dots, x^{(n)}\}$ , the square matrix  $K \in \mathbb{R}^{n \times n}$  whose entries are given by  $K_{ij} = K(x^{(i)}, x^{(j)})$  is symmetric and positive semidefinite. You can find more details about Mercer's theorem in the notes, though the description above is sufficient for this problem. In this question we are interested to see which operations preserve the validity of kernels.

Let  $K_1, K_2$  be kernels over  $\mathbb{R}^d \times \mathbb{R}^d$ , let  $a \in \mathbb{R}^+$  be a positive real number, let  $f : \mathbb{R}^d \mapsto \mathbb{R}$  be a real-valued function, let  $\phi : \mathbb{R}^d \rightarrow \mathbb{R}^p$  be a function mapping from  $\mathbb{R}^d$  to  $\mathbb{R}^p$ , let  $K_3$  be a kernel over  $\mathbb{R}^p \times \mathbb{R}^p$ , and let  $p(x)$  a polynomial over  $x$  with *positive* coefficients.

For each of the functions  $K$  below, state whether it is necessarily a kernel. If you think it is, prove it; if you think it isn't, give a counter-example.

- (a) [1 points]  $K(x, z) = K_1(x, z) + K_2(x, z)$
- (b) [1 points]  $K(x, z) = K_1(x, z) - K_2(x, z)$
- (c) [1 points]  $K(x, z) = aK_1(x, z)$
- (d) [1 points]  $K(x, z) = -aK_1(x, z)$
- (e) [5 points]  $K(x, z) = K_1(x, z)K_2(x, z)$
- (f) [3 points]  $K(x, z) = f(x)f(z)$
- (g) [3 points]  $K(x, z) = K_3(\phi(x), \phi(z))$
- (h) [3 points]  $K(x, z) = p(K_1(x, z))$

[**Hint:** For part (e), the answer is that  $K$  *is* indeed a kernel. You still have to prove it, though. (This one may be harder than the rest.) This result may also be useful for another part of the problem.]

#### 4. [15 points] Kernelizing the Perceptron

Let there be a binary classification problem with  $y \in \{0, 1\}$ . The perceptron uses hypotheses of the form  $h_\theta(x) = g(\theta^T x)$ , where  $g(z) = \text{sign}(z) = 1$  if  $z \geq 0$ , 0 otherwise. In this problem we will consider a stochastic gradient descent-like implementation of the perceptron algorithm where each update to the parameters  $\theta$  is made using only one training example. However, unlike stochastic gradient descent, the perceptron algorithm will only make one pass through the entire training set. The update rule for this version of the perceptron algorithm is given by

$$\theta^{(i+1)} := \theta^{(i)} + \alpha(y^{(i+1)} - h_{\theta^{(i)}}(x^{(i+1)}))x^{(i+1)}$$

where  $\theta^{(i)}$  is the value of the parameters after the algorithm has seen the first  $i$  training examples. Prior to seeing any training examples,  $\theta^{(0)}$  is initialized to  $\vec{0}$ .

- (a) [3 points] Let  $K$  be a kernel corresponding to some very high-dimensional feature mapping  $\phi$ . Suppose  $\phi$  is so high-dimensional (say,  $\infty$ -dimensional) that it's infeasible to ever represent  $\phi(x)$  explicitly. Describe how you would apply the “kernel trick” to the perceptron to make it work in the high-dimensional feature space  $\phi$ , but without ever explicitly computing  $\phi(x)$ .

[Note: You don't have to worry about the intercept term. If you like, think of  $\phi$  as having the property that  $\phi_0(x) = 1$  so that this is taken care of.] Your description should specify:

- i. [1 points] How you will (implicitly) represent the high-dimensional parameter vector  $\theta^{(i)}$ , including how the initial value  $\theta^{(0)} = 0$  is represented (note that  $\theta^{(i)}$  is now a vector whose dimension is the same as the feature vectors  $\phi(x)$ );
- ii. [1 points] How you will efficiently make a prediction on a new input  $x^{(i+1)}$ . I.e., how you will compute  $h_{\theta^{(i)}}(x^{(i+1)}) = g(\theta^{(i)T} \phi(x^{(i+1)}))$ , using your representation of  $\theta^{(i)}$ ; and
- iii. [1 points] How you will modify the update rule given above to perform an update to  $\theta$  on a new training example  $(x^{(i+1)}, y^{(i+1)})$ ; i.e., using the update rule corresponding to the feature mapping  $\phi$ :

$$\theta^{(i+1)} := \theta^{(i)} + \alpha(y^{(i+1)} - h_{\theta^{(i)}}(x^{(i+1)}))\phi(x^{(i+1)})$$

- (b) [10 points] Implement your approach by completing the `initial_state`, `predict`, and `update_state` methods of `src/perceptron/perceptron.py`.

We provide three functions to be used as kernel, a dot-product kernel defined as:

$$K(x, z) = x^\top z, \tag{1}$$

a radial basis function (RBF) kernel, defined as:

$$K(x, z) = \exp\left(-\frac{\|x - z\|_2^2}{2\sigma^2}\right), \tag{2}$$

and finally the following function:

$$K(x, z) = \begin{cases} -1 & x = z \\ 0 & x \neq z \end{cases} \tag{3}$$

Note that the last function is not a kernel function (since its corresponding matrix is not a PSD matrix). However, we are still interested to see what happens when

the kernel is invalid. Run `src/perceptron/perceptron.py` to train kernelized perceptrons on `src/perceptron/train.csv`. The code will then test the perceptron on `src/perceptron/test.csv` and save the resulting predictions in the `src/perceptron/` folder. Plots will also be saved in `src/perceptron/`.

Include the three plots (corresponding to each of the kernels) in your writeup, and indicate which plot belongs to which function.

- (c) [2 points] One of the choices in Q4b completely fails, one works a bit, and one works well in classifying the points. Discuss the performance of different choices and why do they fail or perform well?

### 5. [30 points] Neural Networks: MNIST image classification

In this problem, you will implement a simple neural network to classify grayscale images of handwritten digits (0 - 9) from the MNIST dataset. The dataset contains 60,000 training images and 10,000 testing images of handwritten digits, 0 - 9. Each image is 28×28 pixels in size, and is generally represented as a flat vector of 784 numbers. It also includes labels for each example, a number indicating the actual digit (0 - 9) handwritten in that image. A sample of a few such images are shown below.



The data and starter code for this problem can be found in

- `src/mnist/nn.py`
- `src/mnist/images_train.csv`
- `src/mnist/labels_train.csv`
- `src/mnist/images_test.csv`
- `src/mnist/labels_test.csv`

The starter code splits the set of 60,000 training images and labels into a set of 50,000 examples as the training set, and 10,000 examples for dev set.

To start, you will implement a neural network with a single hidden layer and cross entropy loss, and train it with the provided data set. Use the sigmoid function as activation for the hidden layer, and softmax function for the output layer. Recall that for a single example  $(x, y)$ , the cross entropy loss is:

$$CE(y, \hat{y}) = - \sum_{k=1}^K y_k \log \hat{y}_k,$$

where  $\hat{y} \in \mathbb{R}^K$  is the vector of softmax outputs from the model for the training example  $x$ , and  $y \in \mathbb{R}^K$  is the ground-truth vector for the training example  $x$  such that  $y = [0, \dots, 0, 1, 0, \dots, 0]^\top$  contains a single 1 at the position of the correct class (also called a “one-hot” representation).

For clarity, we provide the forward propagation equations below for the neural network with a single hidden layer. We have labeled data  $(x^{(i)}, y^{(i)})_{i=1}^n$ , where  $x^{(i)} \in \mathbb{R}^d$ , and  $y^{(i)} \in \mathbb{R}^K$  is a

one-hot vector as described above. Let  $h$  be the number of hidden units in the neural network, so that weight matrices  $W^{[1]} \in \mathbb{R}^{d \times h}$  and  $W^{[2]} \in \mathbb{R}^{h \times K}$ . We also have biases  $b^{[1]} \in \mathbb{R}^h$  and  $b^{[2]} \in \mathbb{R}^K$ . The forward propagation equations for a single input  $x^{(i)}$  then are:

$$\begin{aligned} a^{(i)} &= \sigma\left(W^{[1]}\top x^{(i)} + b^{[1]}\right) \in \mathbb{R}^h \\ z^{(i)} &= W^{[2]}\top a^{(i)} + b^{[2]} \in \mathbb{R}^K \\ \hat{y}^{(i)} &= \text{softmax}(z^{(i)}) \in \mathbb{R}^K \end{aligned}$$

where  $\sigma$  is the sigmoid function.

For  $n$  training examples, we average the cross entropy loss over the  $n$  examples.

$$J(W^{[1]}, W^{[2]}, b^{[1]}, b^{[2]}) = \frac{1}{n} \sum_{i=1}^n CE(y^{(i)}, \hat{y}^{(i)}) = -\frac{1}{n} \sum_{i=1}^n \sum_{k=1}^K y_k^{(i)} \log \hat{y}_k^{(i)}.$$

The starter code already converts labels into one hot representations for you.

Instead of batch gradient descent or stochastic gradient descent, the common practice is to use mini-batch gradient descent for deep learning tasks. In this case, the cost function is defined as follows:

$$J_{MB} = \frac{1}{B} \sum_{i=1}^B CE(y^{(i)}, \hat{y}^{(i)})$$

where  $B$  is the batch size, i.e. the number of training example in each mini-batch.

(a) [5 points]

For a single input example  $x^{(i)}$  with one-hot label vector  $y^{(i)}$ , show that

$$\nabla_{z^{(i)}} \text{CE}(y^{(i)}, \hat{y}^{(i)}) = \hat{y}^{(i)} - y^{(i)} \in \mathbb{R}^K$$

where  $z^{(i)} \in \mathbb{R}^K$  is the input to the softmax function, i.e.

$$\hat{y}^{(i)} = \text{softmax}(z^{(i)})$$

(Note: in deep learning,  $z^{(i)}$  is sometimes referred to as the "logits".)

**Hint:** To simplify your answer, it might be convenient to denote the true label of  $x^{(i)}$  as  $l \in \{1, \dots, K\}$ . Hence  $l$  is the index such that that  $y^{(i)} = [0, \dots, 0, 1, 0, \dots, 0]^\top$  contains a single 1 at the  $l$ -th position. You may also wish to compute  $\frac{\partial \text{CE}(y^{(i)}, \hat{y}^{(i)})}{\partial z_j^{(i)}}$  for  $j \neq l$  and  $j = l$  separately.

(b) [15 points]

Implement both forward-propagation and back-propagation for the above loss function  $J_{MB} = \frac{1}{B} \sum_{i=1}^B \text{CE}(y^{(i)}, \hat{y}^{(i)})$ . Initialize the weights of the network by sampling values from a standard normal distribution. Initialize the bias/intercept term to 0. Set the number of hidden units to be 300, and learning rate to be 5. Set  $B = 1,000$  (mini batch size). This means that we train with 1,000 examples in each iteration. Therefore, for each epoch,

we need 50 iterations to cover the entire training data. The images are pre-shuffled. So you don't need to randomly sample the data, and can just create mini-batches sequentially.

Train the model with mini-batch gradient descent as described above. Run the training for 30 epochs. At the end of each epoch, calculate the value of loss function averaged over the entire training set, and plot it (y-axis) against the number of epochs (x-axis). In the same image, plot the value of the loss function averaged over the dev set, and plot it against the number of epochs.

Similarly, in a new image, plot the accuracy (on y-axis) over the training set, measured as the fraction of correctly classified examples, versus the number of epochs (x-axis). In the same image, also plot the accuracy over the dev set versus number of epochs.

**Submit the two plots (one for loss vs epoch, another for accuracy vs epoch) in your writeup.**

Also, at the end of 30 epochs, save the learnt parameters (i.e all the weights and biases) into a file, so that next time you can directly initialize the parameters with these values from the file, rather than re-training all over. You do NOT need to submit these parameters.

**Hint:** Be sure to vectorize your code as much as possible! Training can be very slow otherwise.

- (c) [7 points] Now add a regularization term to your cross entropy loss. The loss function will become

$$J_{MB} = \left( \frac{1}{B} \sum_{i=1}^B CE(y^{(i)}, \hat{y}^{(i)}) \right) + \lambda \left( \|W^{[1]}\|^2 + \|W^{[2]}\|^2 \right)$$

Be careful not to regularize the bias/intercept term. Set  $\lambda$  to be 0.0001. Implement the regularized version and plot the same figures as part (a). Be careful NOT to include the regularization term to measure the loss value for plotting (i.e., regularization should only be used for gradient calculation for the purpose of training).

**Submit the two new plots obtained with regularized training (i.e loss (without regularization term) vs epoch, and accuracy vs epoch) in your writeup.**

**Compare the plots obtained from the regularized model with the plots obtained from the non-regularized model, and summarize your observations in a couple of sentences.**

As in the previous part, save the learnt parameters (weights and biases) into a different file so that we can initialize from them next time.

- (d) [3 points] All this while you should have stayed away from the test data completely. Now that you have convinced yourself that the model is working as expected (i.e, the observations you made in the previous part matches what you learnt in class about regularization), it is finally time to measure the model performance on the test set. Once we measure the test set performance, we report it (whatever value it may be), and NOT go back and refine the model any further.

Initialize your model from the parameters saved in part (a) (i.e, the non-regularized model), and evaluate the model performance on the test data. Repeat this using the parameters saved in part (b) (i.e, the regularized model).

Report your test accuracy for both regularized model and non-regularized model. Briefly (in one sentence) explain why this outcome makes sense" You should have accuracy close to 0.92870 without regularization, and 0.96760 with regularization. Note: these accuracies assume you implement the code with the matrix dimensions as specified in the comments,

which is not the same way as specified in your code. Even if you do not precisely these numbers, you should observe good accuracy and better test accuracy with regularization.

### 6. [20 points] Bayesian Interpretation of Regularization

**Background:** In Bayesian statistics, almost every quantity is a random variable, which can either be observed or unobserved. For instance, parameters  $\theta$  are generally unobserved random variables, and data  $x$  and  $y$  are observed random variables. The joint distribution of all the random variables is also called the *model* (e.g.,  $p(x, y, \theta)$ ). Every unknown quantity can be estimated by conditioning the model on all the observed quantities. Such a conditional distribution over the unobserved random variables, conditioned on the observed random variables, is called the *posterior distribution*. For instance  $p(\theta|x, y)$  is the posterior distribution in the machine learning context. A consequence of this approach is that we are required to endow our model parameters, i.e.,  $p(\theta)$ , with a *prior distribution*. The prior probabilities are to be assigned *before* we see the data—they capture our prior beliefs of what the model parameters might be before observing any evidence.

In the purest Bayesian interpretation, we are required to keep the entire posterior distribution over the parameters all the way until prediction, to come up with the *posterior predictive distribution*, and the final prediction will be the expected value of the posterior predictive distribution. However in most situations, this is computationally very expensive, and we settle for a compromise that is *less pure* (in the Bayesian sense).

The compromise is to estimate a point value of the parameters (instead of the full distribution) which is the mode of the posterior distribution. Estimating the mode of the posterior distribution is also called *maximum a posteriori estimation* (MAP). That is,

$$\theta_{\text{MAP}} = \arg \max_{\theta} p(\theta|x, y).$$

Compare this to the *maximum likelihood estimation* (MLE) we have seen previously:

$$\theta_{\text{MLE}} = \arg \max_{\theta} p(y|x, \theta).$$

In this problem, we explore the connection between MAP estimation, and common regularization techniques that are applied with MLE estimation. In particular, you will show how the choice of prior distribution over  $\theta$  (e.g., Gaussian or Laplace prior) is equivalent to different kinds of regularization (e.g.,  $L_2$ , or  $L_1$  regularization). You will also explore how regularization strengths affect generalization in part (d).

- (a) [3 points] Show that  $\theta_{\text{MAP}} = \arg \max_{\theta} p(y|x, \theta)p(\theta)$  if we assume that  $p(\theta) = p(\theta|x)$ . The assumption that  $p(\theta) = p(\theta|x)$  will be valid for models such as linear regression where the input  $x$  are not explicitly modeled by  $\theta$ . (Note that this means  $x$  and  $\theta$  are marginally independent, but not conditionally independent when  $y$  is given.)
- (b) [5 points] Recall that  $L_2$  regularization penalizes the  $L_2$  norm of the parameters while minimizing the loss (i.e., negative log likelihood in case of probabilistic models). Now we will show that MAP estimation with a zero-mean Gaussian prior over  $\theta$ , specifically  $\theta \sim \mathcal{N}(0, \eta^2 I)$ , is equivalent to applying  $L_2$  regularization with MLE estimation. Specifically, show that for some scalar  $\lambda$ ,

$$\theta_{\text{MAP}} = \arg \min_{\theta} -\log p(y|x, \theta) + \lambda \|\theta\|_2^2. \quad (4)$$

Also, what is the value of  $\lambda$ ?

- (c) [7 points] Now consider a specific instance, a linear regression model given by  $y = \theta^T x + \epsilon$  where  $\epsilon \sim \mathcal{N}(0, \sigma^2)$ . Assume that the random noise  $\epsilon^{(i)}$  is independent for every training

example  $x^{(i)}$ . Like before, assume a Gaussian prior on this model such that  $\theta \sim \mathcal{N}(0, \eta^2 I)$ . For notation, let  $X$  be the design matrix of all the training example inputs where each row vector is one example input, and  $\vec{y}$  be the column vector of all the example outputs.

Come up with a closed form expression for  $\theta_{\text{MAP}}$ .

- (d) [5 points] Next, consider the Laplace distribution, whose density is given by

$$f_L(z|\mu, b) = \frac{1}{2b} \exp\left(-\frac{|z - \mu|}{b}\right).$$

As before, consider a linear regression model given by  $y = x^T \theta + \epsilon$  where  $\epsilon \sim \mathcal{N}(0, \sigma^2)$ . Assume a Laplace prior on this model, where each parameter  $\theta_i$  is marginally independent, and is distributed as  $\theta_i \sim \mathcal{L}(0, b)$ .

Show that  $\theta_{\text{MAP}}$  in this case is equivalent to the solution of linear regression with  $L_1$  regularization, whose loss is specified as

$$J(\theta) = \|X\theta - \vec{y}\|_2^2 + \gamma \|\theta\|_1$$

Also, what is the value of  $\gamma$ ?

**Note:** A closed form solution for linear regression problem with  $L_1$  regularization does not exist. To optimize this, we use gradient descent with a random initialization and solve it numerically.

**Remark:** Linear regression with  $L_2$  regularization is also commonly called *Ridge regression*, and when  $L_1$  regularization is employed, is commonly called *Lasso regression*. These regularizations can be applied to any Generalized Linear models just as above (by replacing  $\log p(y|x, \theta)$  with the appropriate family likelihood). Regularization techniques of the above type are also called *weight decay*, and *shrinkage*. The Gaussian and Laplace priors encourage the parameter values to be closer to their mean (*i.e.*, zero), which results in the shrinkage effect.

**Remark:** Lasso regression (*i.e.*,  $L_1$  regularization) is known to result in sparse parameters, where most of the parameter values are zero, with only some of them non-zero.

# CS 229, Summer 2020

## Problem Set #3

---

**Due Monday, August 10 at 11:59 pm on Gradescope.**

**Notes:** (1) These questions require thought, but do not require long answers. Please be as concise as possible. (2) If you have a question about this homework, we encourage you to post your question on our Piazza forum, at <https://piazza.com/stanford/summer2020/cs229>. (3) This quarter, Summer 2020, students may submit in pairs. If you do so, make sure both names are attached to the Gradescope submission. However, students are not allowed to work with the same partner on more than one assignment. If you missed the first lecture or are unfamiliar with the collaboration or honor code policy, please read the policy on the course website before starting work. (4) For the coding problems, you may not use any libraries except those defined in the provided `environment.yml` file. In particular, ML-specific libraries such as scikit-learn are not permitted. (5) To account for late days, the due date is Monday, August 10 at 11:59 pm. If you submit after Monday, August 10 at 11:59 pm, you will begin consuming your late days. If you wish to submit on time, submit before Monday, August 10 at 11:59 pm.

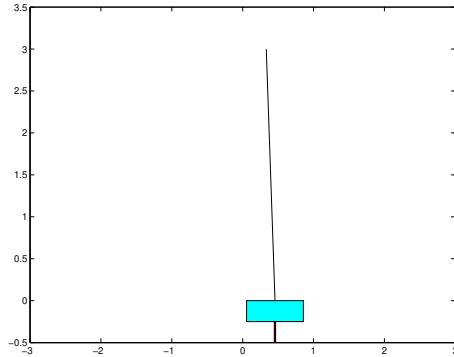
All students must submit an electronic PDF version of the written questions. We highly recommend typesetting your solutions via L<sup>A</sup>T<sub>E</sub>X, and we will award one bonus point for typeset submissions. All students must also submit a zip file of their source code to Gradescope, which should be created using the `make_zip.py` script. You should make sure to (1) restrict yourself to only using libraries included in the `environment.yml` file, and (2) make sure your code runs without errors. Your submission may be evaluated by the auto-grader using a private test set, or used for verifying the outputs reported in the writeup.

### 1. [25 points] Reinforcement Learning: The inverted pendulum

In this problem, you will apply reinforcement learning to automatically design a policy for a difficult control task, without ever using any explicit knowledge of the dynamics of the underlying system.

The problem we will consider is the inverted pendulum or the pole-balancing problem.<sup>1</sup>

Consider the figure shown. A thin pole is connected via a free hinge to a cart, which can move laterally on a smooth table surface. The controller is said to have failed if either the angle of the pole deviates by more than a certain amount from the vertical position (i.e., if the pole falls over), or if the cart's position goes out of bounds (i.e., if it falls off the end of the table). Our objective is to develop a controller to balance the pole with these constraints, by appropriately having the cart accelerate left and right.



We have written a simple simulator for this problem. The simulation proceeds in discrete time cycles (steps). The state of the cart and pole at any time is completely characterized by 4 parameters: the cart position  $x$ , the cart velocity  $\dot{x}$ , the angle of the pole  $\theta$  measured as its deviation from the vertical position, and the angular velocity of the pole  $\dot{\theta}$ . Since it would be simpler to consider reinforcement learning in a discrete state space, we have approximated the state space by a discretization that maps a state vector  $(x, \dot{x}, \theta, \dot{\theta})$  into a number from 0 to `NUM_STATES-1`. Your learning algorithm will need to deal only with this discretized representation of the states.

At every time step, the controller must choose one of two actions - push (accelerate) the cart right, or push the cart left. (To keep the problem simple, there is no *do-nothing* action.) These are represented as actions 0 and 1 respectively in the code. When the action choice is made, the simulator updates the state parameters according to the underlying dynamics, and provides a new discretized state.

We will assume that the reward  $R(s)$  is a function of the current state only. When the pole angle goes beyond a certain limit or when the cart goes too far out, a negative reward is given, and the system is reinitialized randomly. At all other times, the reward is zero. Your program must learn to balance the pole using only the state transitions and rewards observed.

The files for this problem are in `src/cartpole/` directory. Most of the the code has already been written for you, and you need to make changes only to `cartpole.py` in the places specified. This file can be run to show a display and to plot a learning curve at the end. Read the comments at the top of the file for more details on the working of the simulation.

---

<sup>1</sup>The dynamics are adapted from <http://www-anw.cs.umass.edu/rll/domains.html>

To solve the inverted pendulum problem, you will estimate a model (i.e., transition probabilities and rewards) for the underlying MDP, solve Bellman's equations for this estimated MDP to obtain a value function, and act greedily with respect to this value function.

Briefly, you will maintain a current model of the MDP and a current estimate of the value function. Initially, each state has estimated reward zero, and the estimated transition probabilities are uniform (equally likely to end up in any other state).

During the simulation, you must choose actions at each time step according to some current policy. As the program goes along taking actions, it will gather observations on transitions and rewards, which it can use to get a better estimate of the MDP model. Since it is inefficient to update the whole estimated MDP after every observation, we will store the state transitions and reward observations each time, and update the model and value function/policy only periodically. Thus, you must maintain counts of the total number of times the transition from state  $s_i$  to state  $s_j$  using action  $a$  has been observed (similarly for the rewards). Note that the rewards at any state are deterministic, but the state transitions are not because of the discretization of the state space (several different but close configurations may map onto the same discretized state).

Each time a failure occurs (such as if the pole falls over), you should re-estimate the transition probabilities and rewards as the average of the observed values (if any). Your program must then use value iteration to solve Bellman's equations on the estimated MDP, to get the value function and new optimal policy for the new model. For value iteration, use a convergence criterion that checks if the maximum absolute change in the value function on an iteration exceeds some specified tolerance.

Finally, assume that the whole learning procedure has converged once several consecutive attempts (defined by the parameter `NO_LEARNING_THRESHOLD`) to solve Bellman's equation all converge in the first iteration. Intuitively, this indicates that the estimated model has stopped changing significantly.

The code outline for this problem is already in `cartpole.py`, and you need to write code fragments only at the places specified in the file. There are several details (convergence criteria etc.) that are also explained inside the code. Use a discount factor of  $\gamma = 0.995$ .

Implement the reinforcement learning algorithm as specified, and run it.

- How many trials (how many times did the pole fall over or the cart fall off) did it take before the algorithm converged? Hint: if your solution is correct, on the plot the red line indicating smoothed log num steps to failure should start to flatten out at about 60 iterations.
- Plot a learning curve showing the number of time-steps for which the pole was balanced on each trial. Python starter code already includes the code to plot. Include it in your submission.
- Find the line of code that says `np.random.seed`, and rerun the code with the seed set to 1, 2, and 3. What do you observe? What does this imply about the algorithm?

## 2. [15 points] KL divergence and Maximum Likelihood

The Kullback-Leibler (KL) divergence is a measure of how much one probability distribution is different from a second one. It is a concept that originated in Information Theory, but has made its way into several other fields, including Statistics, Machine Learning, Information Geometry, and many more. In Machine Learning, the KL divergence plays a crucial role, connecting various concepts that might otherwise seem unrelated.

In this problem, we will introduce KL divergence over discrete distributions, practice some simple manipulations, and see its connection to Maximum Likelihood Estimation.

The *KL divergence* between two discrete-valued distributions  $P(X), Q(X)$  over the outcome space  $\mathcal{X}$  is defined as follows<sup>2</sup>:

$$D_{KL}(P\|Q) = \sum_{x \in \mathcal{X}} P(x) \log \frac{P(x)}{Q(x)}$$

For notational convenience, we assume  $P(x) > 0, \forall x$ . (One other standard thing to do is to adopt the convention that “ $0 \log 0 = 0$ .”) Sometimes, we also write the KL divergence more explicitly as  $D_{KL}(P||Q) = D_{KL}(P(X)||Q(X))$ .

### *Background on Information Theory*

Before we dive deeper, we give a brief (optional) Information Theoretic background on KL divergence. While this introduction is not necessary to answer the assignment question, it may help you better understand and appreciate why we study KL divergence, and how Information Theory can be relevant to Machine Learning.

We start with the *entropy*  $H(P)$  of a probability distribution  $P(X)$ , which is defined as

$$H(P) = - \sum_{x \in \mathcal{X}} P(x) \log P(x).$$

Intuitively, entropy measures how dispersed a probability distribution is. For example, a uniform distribution is considered to have very high entropy (i.e. a lot of uncertainty), whereas a distribution that assigns all its mass on a single point is considered to have zero entropy (i.e. no uncertainty). Notably, it can be shown that among continuous distributions over  $\mathbb{R}$ , the Gaussian distribution  $\mathcal{N}(\mu, \sigma^2)$  has the highest entropy (highest uncertainty) among all possible distributions that have the given mean  $\mu$  and variance  $\sigma^2$ .

To further solidify our intuition, we present motivation from communication theory. Suppose we want to communicate from a source to a destination, and our messages are always (a sequence of) discrete symbols over space  $\mathcal{X}$  (for example,  $\mathcal{X}$  could be letters  $\{a, b, \dots, z\}$ ). We want to construct an encoding scheme for our symbols in the form of sequences of binary bits that are transmitted over the channel. Further, suppose that in the long run the frequency of occurrence of symbols follow a probability distribution  $P(X)$ . This means, in the long run, the fraction of times the symbol  $x$  gets transmitted is  $P(x)$ .

A common desire is to construct an encoding scheme such that the average number of bits per symbol transmitted remains as small as possible. Intuitively, this means we want very frequent symbols to be assigned to a bit pattern having a small number of bits. Likewise, because we are

---

<sup>2</sup>If  $P$  and  $Q$  are densities for continuous-valued random variables, then the sum is replaced by an integral, and everything stated in this problem works fine as well. But for the sake of simplicity, in this problem we'll just work with this form of KL divergence for probability mass functions/discrete-valued distributions.

interested in reducing the average number of bits per symbol in the long term, it is tolerable for infrequent words to be assigned to bit patterns having a large number of bits, since their low frequency has little effect on the long term average. The encoding scheme can be as complex as we desire, for example, a single bit could possibly represent a long sequence of multiple symbols (if that specific pattern of symbols is very common). The entropy of a probability distribution  $P(X)$  is its optimal bit rate, i.e., the lowest average bits per message that can possibly be achieved if the symbols  $x \in \mathcal{X}$  occur according to  $P(X)$ . It does not specifically tell us *how* to construct that optimal encoding scheme. It only tells us that no encoding can possibly give us a lower long term bits per message than  $H(P)$ .

To see a concrete example, suppose our messages have a vocabulary of  $K = 32$  symbols, and each symbol has an equal probability of transmission in the long term (i.e, uniform probability distribution). An encoding scheme that would work well for this scenario would be to have  $\log_2 K$  bits per symbol, and assign each symbol some unique combination of the  $\log_2 K$  bits. In fact, it turns out that this is the most efficient encoding one can come up with for the uniform distribution scenario.

It may have occurred to you by now that the long term average number of bits per message depends only on the frequency of occurrence of symbols. The encoding scheme of scenario A can in theory be reused in scenario B with a different set of symbols (assume equal vocabulary size for simplicity), with the same long term efficiency, as long as the symbols of scenario B follow the same probability distribution as the symbols of scenario A. It might also have occurred to you, that reusing the encoding scheme designed to be optimal for scenario A, for messages in scenario B having a *different probability* of symbols, will always be suboptimal for scenario B. To be clear, we do not need know *what* the specific optimal schemes are in either scenarios. As long as we know the distributions of their symbols, we can say that the optimal scheme designed for scenario A will be suboptimal for scenario B if the distributions are different.

Concretely, if we reuse the optimal scheme designed for a scenario having symbol distribution  $Q(X)$ , into a scenario that has symbol distribution  $P(X)$ , the long term average number of bits per symbol achieved is called the *cross entropy*, denoted by  $H(P, Q)$ :

$$H(P, Q) = - \sum_{x \in \mathcal{X}} P(x) \log Q(x).$$

To recap, the entropy  $H(P)$  is the best possible long term average bits per message (optimal) that can be achieved under a symbol distribution  $P(X)$  by using an encoding scheme (possibly unknown) specifically designed for  $P(X)$ . The cross entropy  $H(P, Q)$  is the long term average bits per message (suboptimal) that results under a symbol distribution  $P(X)$ , by reusing an encoding scheme (possibly unknown) designed to be optimal for a scenario with symbol distribution  $Q(X)$ .

Now, KL divergence is the penalty we pay, as measured in average number of bits, for using the optimal scheme for  $Q(X)$ , under the scenario where symbols are actually distributed as  $P(X)$ . It is straightforward to see this

$$\begin{aligned} D_{KL}(P \| Q) &= \sum_{x \in \mathcal{X}} P(x) \log \frac{P(x)}{Q(x)} \\ &= \sum_{x \in \mathcal{X}} P(x) \log P(x) - \sum_{x \in \mathcal{X}} P(x) \log Q(x) \\ &= H(P, Q) - H(P). \quad (\text{difference in average number of bits.}) \end{aligned}$$

If the cross entropy between  $P$  and  $Q$  is  $H(P)$  (and hence  $D_{KL}(P||Q) = 0$ ) then it necessarily means  $P = Q$ . In Machine Learning, it is a common task to find a distribution  $Q$  that is “close” to another distribution  $P$ . To achieve this, it is common to use  $D_{KL}(Q||P)$  as the loss function to be optimized. As we will see in this question below, Maximum Likelihood Estimation, which is a commonly used optimization objective, turns out to be equivalent to minimizing the KL divergence between the training data (i.e. the empirical distribution over the data) and the model.

Now, we get back to showing some simple properties of KL divergence.

- (a) [5 points] **Nonnegativity.**

Prove the following:

$$\forall P, Q. \quad D_{KL}(P||Q) \geq 0$$

and

$$D_{KL}(P||Q) = 0 \quad \text{if and only if} \quad P = Q.$$

[Hint: You may use the following result, called **Jensen’s inequality**. If  $f$  is a convex function, and  $X$  is a random variable, then  $E[f(X)] \geq f(E[X])$ . Moreover, if  $f$  is strictly convex ( $f$  is convex if its Hessian satisfies  $H \geq 0$ ; it is *strictly* convex if  $H > 0$ ; for instance  $f(x) = -\log x$  is strictly convex), then  $E[f(X)] = f(E[X])$  implies that  $X = E[X]$  with probability 1; i.e.,  $X$  is actually a constant.]

- (b) [5 points] **Chain rule for KL divergence.**

The KL divergence between 2 conditional distributions  $P(X|Y), Q(X|Y)$  is defined as follows:

$$D_{KL}(P(X|Y)||Q(X|Y)) = \sum_y P(y) \left( \sum_x P(x|y) \log \frac{P(x|y)}{Q(x|y)} \right)$$

This can be thought of as the expected KL divergence between the corresponding conditional distributions on  $x$  (that is, between  $P(X|Y = y)$  and  $Q(X|Y = y)$ ), where the expectation is taken over the random  $y$ .

Prove the following chain rule for KL divergence:

$$D_{KL}(P(X, Y)||Q(X, Y)) = D_{KL}(P(X)||Q(X)) + D_{KL}(P(Y|X)||Q(Y|X)).$$

- (c) [5 points] **KL and maximum likelihood.**

Consider a density estimation problem, and suppose we are given a training set  $\{x^{(i)}; i = 1, \dots, n\}$ . Let the empirical distribution be  $\hat{P}(x) = \frac{1}{n} \sum_{i=1}^n \mathbf{1}\{x^{(i)} = x\}$ . ( $\hat{P}$  is just the uniform distribution over the training set; i.e., sampling from the empirical distribution is the same as picking a random example from the training set.)

Suppose we have some family of distributions  $P_\theta$  parameterized by  $\theta$ . (If you like, think of  $P_\theta(x)$  as an alternative notation for  $P(x; \theta)$ .) Prove that finding the maximum likelihood estimate for the parameter  $\theta$  is equivalent to finding  $P_\theta$  with minimal KL divergence from  $\hat{P}$ . I.e. prove:

$$\arg \min_{\theta} D_{KL}(\hat{P}||P_\theta) = \arg \max_{\theta} \sum_{i=1}^n \log P_\theta(x^{(i)})$$

**Remark.** Consider the relationship between parts (b-c) and multi-variate Bernoulli Naive Bayes parameter estimation. In the Naive Bayes model we assumed  $P_\theta$  is of the following form:  $P_\theta(x, y) = p(y) \prod_{i=1}^d p(x_i|y)$ . By the chain rule for KL divergence, we therefore have:

$$D_{KL}(\hat{P} \| P_\theta) = D_{KL}(\hat{P}(y) \| p(y)) + \sum_{i=1}^d D_{KL}(\hat{P}(x_i|y) \| p(x_i|y)).$$

This shows that finding the maximum likelihood/minimum KL-divergence estimate of the parameters decomposes into  $2n + 1$  independent optimization problems: One for the class priors  $p(y)$ , and one for each of the conditional distributions  $p(x_i|y)$  for each feature  $x_i$  given each of the two possible labels for  $y$ . Specifically, finding the maximum likelihood estimates for each of these problems individually results in also maximizing the likelihood of the joint distribution. (If you know what Bayesian networks are, a similar remark applies to parameter estimation for them.)

### 3. [20 points] K-means for compression

In this problem, we will apply the K-means algorithm to lossy image compression, by reducing the number of colors used in an image.

We will be using the files `src/k_means/peppers-small.tiff` and `src/k_means/peppers-large.tiff`.

The `peppers-large.tiff` file contains a 512x512 image of peppers represented in 24-bit color. This means that, for each of the 262144 pixels in the image, there are three 8-bit numbers (each ranging from 0 to 255) that represent the red, green, and blue intensity values for that pixel. The straightforward representation of this image therefore takes about  $262144 \times 3 = 786432$  bytes (a byte being 8 bits). To compress the image, we will use K-means to reduce the image to  $k = 16$  colors. More specifically, each pixel in the image is considered a point in the three-dimensional  $(r, g, b)$ -space. To compress the image, we will cluster these points in color-space into 16 clusters, and replace each pixel with the closest cluster centroid.

Follow the instructions below. Be warned that some of these operations can take a while (several minutes even on a fast computer)!

- (a) [15 points] **[Coding Problem] K-Means Compression Implementation.** First let us look at our data. From the `src/k_means/` directory, open an interactive Python prompt, and type

```
from matplotlib.image import imread; import matplotlib.pyplot as plt;
```

and run `A = imread('peppers-large.tiff')`. Now, `A` is a “three dimensional matrix,” and `A[:, :, 0]`, `A[:, :, 1]` and `A[:, :, 2]` are 512x512 arrays that respectively contain the red, green, and blue values for each pixel. Enter `plt.imshow(A); plt.show()` to display the image.

Since the large image has 262,144 pixels and would take a while to cluster, we will instead run vector quantization on a smaller image. Repeat (a) with `peppers-small.tiff`.

Next we will implement image compression in the file `src/k_means/k_means.py` which has some starter code. Treating each pixel’s  $(r, g, b)$  values as an element of  $\mathbb{R}^3$ , implement K-means with 16 clusters on the pixel data from this smaller image, iterating (preferably) to convergence, but in no case for less than 30 iterations. For initialization, set each cluster centroid to the  $(r, g, b)$ -values of a randomly chosen pixel in the image.

Take the image of `peppers-large.tiff`, and replace each pixel’s  $(r, g, b)$  values with the value of the closest cluster centroid from the set of centroids computed with `peppers-small.tiff`. Visually compare it to the original image to verify that your implementation is reasonable. **Include in your write-up a copy of this compressed image alongside the original image.**

- (b) [5 points] **Compression Factor.**

If we represent the image with these reduced (16) colors, by (approximately) what factor have we compressed the image?

#### 4. [35 points] Semi-supervised EM

Expectation Maximization (EM) is a classical algorithm for unsupervised learning (*i.e.*, learning with hidden or latent variables). In this problem we will explore one of the ways in which EM algorithm can be adapted to the semi-supervised setting, where we have some labeled examples along with unlabeled examples.

In the standard unsupervised setting, we have  $n \in \mathbb{N}$  unlabeled examples  $\{x^{(1)}, \dots, x^{(n)}\}$ . We wish to learn the parameters of  $p(x, z; \theta)$  from the data, but  $z^{(i)}$ 's are not observed. The classical EM algorithm is designed for this very purpose, where we maximize the intractable  $p(x; \theta)$  indirectly by iteratively performing the E-step and M-step, each time maximizing a tractable lower bound of  $p(x; \theta)$ . Our objective can be concretely written as:

$$\begin{aligned}\ell_{\text{unsup}}(\theta) &= \sum_{i=1}^n \log p(x^{(i)}; \theta) \\ &= \sum_{i=1}^n \log \sum_{z^{(i)}} p(x^{(i)}, z^{(i)}; \theta)\end{aligned}$$

Now, we will attempt to construct an extension of EM to the semi-supervised setting. Let us suppose we have an *additional*  $\tilde{n} \in \mathbb{N}$  labeled examples  $\{(\tilde{x}^{(1)}, \tilde{z}^{(1)}), \dots, (\tilde{x}^{(\tilde{n})}, \tilde{z}^{(\tilde{n})})\}$  where both  $x$  and  $z$  are observed. We want to simultaneously maximize the marginal likelihood of the parameters using the unlabeled examples, and full likelihood of the parameters using the labeled examples, by optimizing their weighted sum (with some hyperparameter  $\alpha$ ). More concretely, our semi-supervised objective  $\ell_{\text{semi-sup}}(\theta)$  can be written as:

$$\begin{aligned}\ell_{\text{sup}}(\theta) &= \sum_{i=1}^{\tilde{n}} \log p(\tilde{x}^{(i)}, \tilde{z}^{(i)}; \theta) \\ \ell_{\text{semi-sup}}(\theta) &= \ell_{\text{unsup}}(\theta) + \alpha \ell_{\text{sup}}(\theta)\end{aligned}$$

We can derive the EM steps for the semi-supervised setting using the same approach and steps as before. You are *strongly encouraged* to show to yourself (no need to include in the write-up) that we end up with:

#### E-step (semi-supervised)

For each  $i \in \{1, \dots, n\}$ , set

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

#### M-step (semi-supervised)

$$\theta^{(t+1)} := \arg \max_{\theta} \left[ \sum_{i=1}^n \left( \sum_{z^{(i)}} Q_i^{(t)}(z^{(i)}) \log \frac{p(x^{(i)}, z^{(i)}; \theta)}{Q_i^{(t)}(z^{(i)})} \right) + \alpha \left( \sum_{i=1}^{\tilde{n}} \log p(\tilde{x}^{(i)}, \tilde{z}^{(i)}; \theta) \right) \right]$$

- (a) [5 points] **Convergence.** First we will show that this algorithm eventually converges. In order to prove this, it is sufficient to show that our semi-supervised objective  $\ell_{\text{semi-sup}}(\theta)$  monotonically increases with each iteration of E and M step. Specifically, let  $\theta^{(t)}$  be the parameters obtained at the end of  $t$  EM-steps. Show that  $\ell_{\text{semi-sup}}(\theta^{(t+1)}) \geq \ell_{\text{semi-sup}}(\theta^{(t)})$ .

### Semi-supervised GMM

Now we will revisit the Gaussian Mixture Model (GMM), to apply our semi-supervised EM algorithm. Let us consider a scenario where data is generated from  $k \in \mathbb{N}$  Gaussian distributions, with unknown means  $\mu_j \in \mathbb{R}^d$  and covariances  $\Sigma_j \in \mathbb{S}_+^d$  where  $j \in \{1, \dots, k\}$ . We have  $n$  data points  $x^{(i)} \in \mathbb{R}^d, i \in \{1, \dots, n\}$ , and each data point has a corresponding latent (hidden/unknown) variable  $z^{(i)} \in \{1, \dots, k\}$  indicating which distribution  $x^{(i)}$  belongs to. Specifically,  $z^{(i)} \sim \text{Multinomial}(\phi)$ , such that  $\sum_{j=1}^k \phi_j = 1$  and  $\phi_j \geq 0$  for all  $j$ , and  $x^{(i)}|z^{(i)} \sim \mathcal{N}(\mu_{z^{(i)}}, \Sigma_{z^{(i)}})$  i.i.d. So,  $\mu$ ,  $\Sigma$ , and  $\phi$  are the model parameters.

We also have additional  $\tilde{n}$  data points  $\tilde{x}^{(i)} \in \mathbb{R}^d, i \in \{1, \dots, \tilde{n}\}$ , and an associated *observed* variable  $\tilde{z}^{(i)} \in \{1, \dots, k\}$  indicating the distribution  $\tilde{x}^{(i)}$  belongs to. Note that  $\tilde{z}^{(i)}$  are known constants (in contrast to  $z^{(i)}$  which are unknown *random* variables). As before, we assume  $\tilde{x}^{(i)}|\tilde{z}^{(i)} \sim \mathcal{N}(\mu_{\tilde{z}^{(i)}}, \Sigma_{\tilde{z}^{(i)}})$  i.i.d.

In summary we have  $n + \tilde{n}$  examples, of which  $n$  are unlabeled data points  $x$ 's with unobserved  $z$ 's, and  $\tilde{n}$  are labeled data points  $\tilde{x}^{(i)}$  with corresponding observed labels  $\tilde{z}^{(i)}$ . The traditional EM algorithm is designed to take only the  $n$  unlabeled examples as input, and learn the model parameters  $\mu$ ,  $\Sigma$ , and  $\phi$ .

Our task now will be to apply the semi-supervised EM algorithm to GMMs in order to also leverage the additional  $\tilde{n}$  labeled examples, and come up with semi-supervised E-step and M-step update rules specific to GMMs. Whenever required, you can cite the lecture notes for derivations and steps.

- (b) [5 points] **Semi-supervised E-Step.** Clearly state which are all the latent variables that need to be re-estimated in the E-step. Derive the E-step to re-estimate all the stated latent variables. Your final E-step expression must only involve  $x, z, \mu, \Sigma, \phi$  and universal constants.
- (c) [10 points] **Semi-supervised M-Step.** Clearly state which are all the parameters that need to be re-estimated in the M-step. Derive the M-step to re-estimate all the stated parameters. Specifically, derive closed form expressions for the parameter update rules for  $\mu^{(t+1)}, \Sigma^{(t+1)}$  and  $\phi^{(t+1)}$  based on the semi-supervised objective.
- (d) [5 points] **Classical (Unsupervised) EM Implementation.** For this sub-question, we are only going to consider the  $n$  unlabelled examples. Follow the instructions in `src/semi_supervised_em/gmm.py` to implement the traditional EM algorithm, and run it on the unlabelled data-set until convergence.

Run three trials and use the provided plotting function to construct a scatter plot of the resulting assignments to clusters (one plot for each trial). Your plot should indicate cluster assignments with colors they got assigned to (*i.e.*, the cluster which had the highest probability in the final E-step).

**Submit the three plots obtained above in your write-up.**

- (e) [7 points] **Semi-supervised EM Implementation.** Now we will consider both the labelled and unlabelled examples (a total of  $n + \tilde{n}$ ), with 5 labelled examples per cluster. We have provided starter code for splitting the dataset into matrices `x` and `x_tilde` of unlabelled and labelled examples respectively. Add to your code in `src/semi_supervised_em/gmm.py` to implement the modified EM algorithm, and run it on the dataset until convergence.

Create a plot for each trial, as done in the previous sub-question.

**Submit the three plots obtained above in your write-up.**

(f) [3 points] **Comparison of Unsupervised and Semi-supervised EM.** Briefly describe the differences you saw in unsupervised *vs.* semi-supervised EM for each of the following:

- i. Number of iterations taken to converge.
- ii. Stability (*i.e.*, how much did assignments change with different random initializations?)
- iii. Overall quality of assignments.

**Note:** The dataset was sampled from a mixture of three low-variance Gaussian distributions, and a fourth, high-variance Gaussian distribution. This should be useful in determining the overall quality of the assignments that were found by the two algorithms.

### 5. [10 points] PCA

In class, we showed that PCA finds the “variance maximizing” directions onto which to project the data. In this problem, we find another interpretation of PCA.

Suppose we are given a set of points  $\{x^{(1)}, \dots, x^{(n)}\}$ . Let us assume that we have as usual preprocessed the data to have zero-mean and unit variance in each coordinate. For a given unit-length vector  $u$ , let  $f_u(x)$  be the projection of point  $x$  onto the direction given by  $u$ . I.e., if  $\mathcal{V} = \{\alpha u : \alpha \in \mathbb{R}\}$ , then

$$f_u(x) = \arg \min_{v \in \mathcal{V}} \|x - v\|^2.$$

Show that the unit-length vector  $u$  that minimizes the mean squared error between projected points and original points corresponds to the first principal component for the data. I.e., show that

$$\arg \min_{u: u^T u = 1} \sum_{i=1}^n \|x^{(i)} - f_u(x^{(i)})\|_2^2.$$

gives the first principal component.

**Remark.** If we are asked to find a  $k$ -dimensional subspace onto which to project the data so as to minimize the sum of squares distance between the original data and their projections, then we should choose the  $k$ -dimensional subspace spanned by the first  $k$  principal components of the data. This problem shows that this result holds for the case of  $k = 1$ .

### 6. [20 points] Independent components analysis

While studying Independent Component Analysis (ICA) in class, we made an informal argument about why Gaussian distributed sources will not work. We also mentioned that any other distribution (except Gaussian) for the sources will work for ICA, and hence used the logistic distribution instead. In this problem, we will go deeper into understanding why Gaussian distributed sources are a problem. We will also derive ICA with the Laplace distribution, and apply it to the cocktail party problem.

Reintroducing notation, let  $s \in \mathbb{R}^d$  be source data that is generated from  $d$  independent sources. Let  $x \in \mathbb{R}^d$  be observed data such that  $x = As$ , where  $A \in \mathbb{R}^{d \times d}$  is called the *mixing matrix*. We assume  $A$  is invertible, and  $W = A^{-1}$  is called the *unmixing matrix*. So,  $s = Wx$ . The goal of ICA is to estimate  $W$ . Similar to the notes, we denote  $w_j^T$  to be the  $j^{th}$  row of  $W$ . Note that this implies that the  $j^{th}$  source can be reconstructed with  $w_j$  and  $x$ , since  $s_j = w_j^T x$ . We are given a training set  $\{x^{(1)}, \dots, x^{(n)}\}$  for the following sub-questions. Let us denote the entire training set by the design matrix  $X \in \mathbb{R}^{n \times d}$  where each example corresponds to a row in the matrix.

#### (a) [5 points] Gaussian source

For this sub-question, we assume sources are distributed according to a standard normal distribution, i.e  $s_j \sim \mathcal{N}(0, 1)$ ,  $j = \{1, \dots, d\}$ . The log-likelihood of our unmixing matrix, as described in the notes, is

$$\ell(W) = \sum_{i=1}^n \left( \log |W| + \sum_{j=1}^d \log g'(w_j^T x^{(i)}) \right),$$

where  $g$  is the cumulative distribution function, and  $g'$  is the probability density function of the source distribution (in this sub-question it is a standard normal distribution). Whereas in the notes we derive an update rule to train  $W$  iteratively, for the cause of Gaussian distributed sources, we can analytically reason about the resulting  $W$ .

Try to derive a closed form expression for  $W$  in terms of  $X$  when  $g$  is the standard normal CDF. Deduce the relation between  $W$  and  $X$  in the simplest terms, and highlight the ambiguity (in terms of rotational invariance) in computing  $W$ .

#### (b) [10 points] Laplace source.

For this sub-question, we assume sources are distributed according to a standard Laplace distribution, i.e  $s_i \sim \mathcal{L}(0, 1)$ . The Laplace distribution  $\mathcal{L}(0, 1)$  has PDF  $f_{\mathcal{L}}(s) = \frac{1}{2} \exp(-|s|)$ . With this assumption, derive the update rule for a single example in the form

$$W := W + \alpha(\dots).$$

#### (c) [5 points] Cocktail Party Problem

For this question you will implement the Bell and Sejnowski ICA algorithm, but assuming a Laplace source (as derived in part-b), instead of the Logistic distribution covered in class. The file `src/ica/mix.dat` contains the input data which consists of a matrix with 5 columns, with each column corresponding to one of the mixed signals  $x_i$ . The code for this question can be found in `src/ica/ica.py`.

Implement the `update_W` and `unmix` functions in `src/ica/ica.py`.

You can then run `ica.py` in order to split the mixed audio into its components. The mixed audio tracks are written to `mixed_i.wav` in the output folder. The split audio tracks are written to `split_i.wav` in the output folder.

To make sure your code is correct, you should listen to the resulting unmixed sources. (Some overlap or noise in the sources may be present, but the different sources should be pretty clearly separated.)

**Submit the full unmixing matrix  $W$  ( $5 \times 5$ ) that you obtained, by including the `W.txt` the code outputs along with your code.**

If your implementation is correct, your output `split_0.wav` should sound similar to the file `correct_split_0.wav` included with the source code.

Note: In our implementation, we **anneal** the learning rate  $\alpha$  (slowly decreased it over time) to speed up learning. In addition to using the variable learning rate to speed up convergence, one thing that we also do is choose a random permutation of the training data, and running stochastic gradient ascent visiting the training data in that order (each of the specified learning rates was then used for one full pass through the data).