Machine Learning

Stanford, California

Contents

Acknowledgments

This work is taken from the lecture notes for the course *Machine Learning* at Stanford University, CS 229 (`cs229.stanford.edu`). The contributors to the content of this work are Andrew Ng and Christopher Ré—this collection is simply a typesetting of existing lecture notes with minor modifications and additions of working Julia implementations. We would like to thank the original authors for their contribution. In addition, we wish to thank Mykel Kochenderfer and Tim Wheeler for their contribution to the Tufte-Algorithms LaTeX template, based off of *Algorithms for Optimization*.[1]

[1] M. J. Kochenderfer and T. A. Wheeler, *Algorithms for Optimization*. MIT Press, 2019.

Robert J. Moss
Stanford, Calif.
May 4, 2021

Ancillary material is available on the template's webpage:
`https://github.com/sisl/textbook_template`

Part I: 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:

From CS229 Fall 2020, Tengyu Ma, Andrew Ng, Moses Charikar, & Christopher Ré, Stanford University.

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

Table 1. Housing prices in Portland, OR.

We can plot this data:

Figure 1. Housing prices in Portland, OR.

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 $\{(x^{(i)}, y^{(i)});\ i = 1, \ldots, n\}$—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:

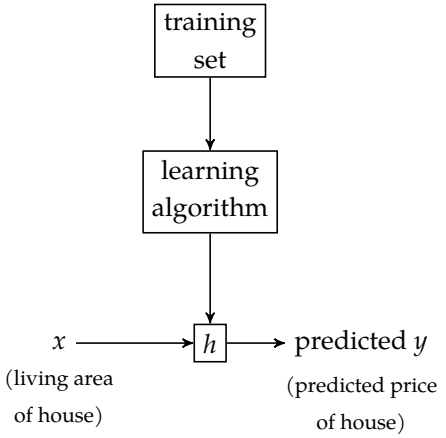


Figure 2. Hypothesis diagram.

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**[2] 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.

[2] The term *regression* was originally coined due to ''regressing'' to the mean (Francis Galton, 1886).

1 Linear Regression

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

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

Table 1.1. Housing prices with bedrooms in Portland, OR.

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.[1]

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:

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 \tag{1.1}$$

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

$$h(x) = \sum_{i=0}^{d} \theta_i x_i = \theta^\top x, \tag{1.2}$$

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

[1] 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.

Now, given a training set, how do we pick, or learn, the parameters θ? One reasonable method seems to be to make $h(x)$ close to y, at least for the training examples we have. To formalize this, we will define a function that measures, for each value of the θ's, how close the $h(x^{(i)})$'s are to the corresponding $y^{(i)}$'s. We define the **cost function**:

$$J(\theta) = \frac{1}{2}\sum_{i=1}^{n}\left(h_\theta(x^{(i)}) - y^{(i)}\right)^2. \tag{1.3}$$

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.

1.1 *Least mean squares (LMS) algorithm*

We want to choose θ so as to minimize $J(\theta)$. To do so, let's use a search algorithm that starts with some "initial guess" for θ, and that repeatedly changes θ to make $J(\theta)$ smaller, until hopefully we converge to a value of θ that minimizes $J(\theta)$. Specifically, let's consider the **gradient descent** algorithm, which starts with some initial θ, and repeatedly performs the update:[2]

$$\theta_j \leftarrow \theta_j - \alpha\frac{\partial}{\partial\theta_j}J(\theta) \tag{1.4}$$

[2] This update is simultaneously performed for all values of $j = 0,\ldots,d$.

Here, α is called the **learning rate**. This is a very natural algorithm that repeatedly takes a step in the direction of steepest decrease of J.

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

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

For a single training example, this gives the update rule:[3]

$$\theta_j \leftarrow \theta_j + \alpha\left(y^{(i)} - h_\theta(x^{(i)})\right)x_j^{(i)}. \tag{1.5}$$

[3] We use the notation "$a \leftarrow b$" 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 (something := is used). In other words, this operation overwrites a with the value of b. In contrast, we will write "$a = b$" when we are asserting a statement of fact, that the value of a is equal to the value of b.

The 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've 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
  for every j **do**
    $\theta_j \leftarrow \theta_j + \alpha\sum_{i=1}^{n}\left(y^{(i)} - h_\theta(x^{(i)})\right)x_j^{(i)}$
  end for
until convergence

Algorithm 1.1. Gradient descent.

By grouping the updates of the coordinates into an update of the vector θ, we can rewrite update algorithm 1.1 in a slightly more succinct way:

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

repeat

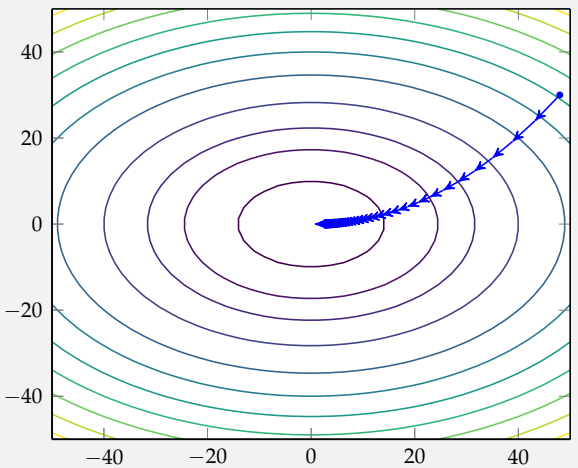
$$\theta \leftarrow \theta + \alpha \sum_{i=1}^{n} \left(y^{(i)} - h_\theta(x^{(i)}) \right) x^{(i)}$$

until convergence

Algorithm 1.2. Gradient descent vectorized.

gradient descent on the original cost function J. This method looks at every example in the entire training set on every step, and is called **batch gradient descent**. Note that, while gradient descent can be susceptible to local minima in general, the optimization problem we have posed here for linear regression has only one global, and no other local, optima; thus gradient descent always converges (assuming the learning rate α is not too large) to the global minimum. Indeed, J is a convex quadratic function.

Here is an example of gradient descent as it is run to minimize a quadratic function.



Example 1.1. Gradient descent on 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 arrows in the figure (joined by straight lines) mark the successive values of θ that gradient descent went through.

Example 1.2. Best fit line using batch gradient descent on Portland, Oregon housing prices.

When we run batch gradient descent to fit θ on our previous dataset, to learn to predict housing price as a function of living area. We obtain:

$$\theta_0 = 71.27 \qquad \text{(intercept)}$$
$$\theta_1 = 0.1345 \qquad \text{(slope)}$$

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 results in example 1.2 were obtained with batch gradient descent. There is an alternative to batch gradient descent that also works very well. Consider the following algorithm:

repeat
  for $i = 1$ to n **do**
    for every j **do**
      $$\theta_j \leftarrow \theta_j + \alpha \sum_{i=1}^{n} \left(y^{(i)} - h_\theta(x^{(i)}) \right) x_j^{(i)}$$
    end for
  end for
until convergence

Algorithm 1.3. Stochastic gradient descent.

By grouping the updates of the coordinates into an update of the vector θ, we can rewrite update in algorithm 1.3 in a slightly more succinct way:

$$\theta \leftarrow \theta + \alpha \left(y^{(i)} - h_\theta^{(i)} \right) x^{(i)} \tag{1.6}$$

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 θ ''close'' to the minimum much faster than batch gradient descent.[4] For these reasons, particularly when the training set is large, stochastic gradient descent is often preferred over batch gradient descent.

[4] Note, however, that it may never ''converge'' to the minimum, and the parameters θ will keep oscillating around the minimum of $J(\theta)$; but in practice most of the values near the minimum will be reasonably good approximations to the true minimum. By slowly letting the learning rate α decrease to zero as the algorithm runs, it is also possible to ensure that the parameters will converge to the global minimum rather than merely oscillate around the minimum.

1.2 *The normal equations*

Gradient descent gives one way of minimizing J. Let's discuss a second way of doing so, this time performing the minimization explicitly and without resorting to an iterative algorithm. In this method, we will minimize J by explicitly taking its derivatives with respect to the θ_j's, and setting them to zero. To enable us to

do this without having to write reams of algebra and pages full of matrices of derivatives, let's introduce some notation for doing calculus with matrices.

1.2.1 *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:

$$\nabla_A f(A) = \begin{bmatrix} \frac{\partial f}{\partial A_{11}} & \cdots & \frac{\partial f}{\partial A_{1d}} \\ \vdots & \ddots & \vdots \\ \frac{\partial f}{\partial A_{n1}} & \cdots & \frac{\partial f}{\partial A_{nd}} \end{bmatrix} \tag{1.7}$$

Thus, the gradient $\nabla_A f(A)$ is itself an n-by-d matrix, whose (i, j)-element is $\partial f / \partial A_{ij}$.

Example 1.3. Matrix derivative.

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:

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

1.2.2 *Least squares revisited*

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

$$\mathbf{X} = \begin{bmatrix} — (x^{(1)})^\top — \\ — (x^{(2)})^\top — \\ \vdots \\ — (x^{(n)})^\top — \end{bmatrix} \tag{1.8}$$

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

$$\mathbf{y} = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(n)} \end{bmatrix} \tag{1.9}$$

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

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

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

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

Finally, to minimize J, let's find its derivative with respect to θ. Hence:

$$\begin{aligned}
\nabla_\theta J(\theta) &= \nabla_\theta \frac{1}{2}(\mathbf{X}\theta - \mathbf{y})^\top (\mathbf{X}\theta - \mathbf{y}) \\
&= \frac{1}{2}\nabla_\theta \left((\mathbf{X}\theta)^\top \mathbf{X}\theta - (\mathbf{X}\theta)^\top \mathbf{y} - \mathbf{y}^\top (\mathbf{X}\theta) + \mathbf{y}^\top \mathbf{y} \right) \\
&= \frac{1}{2}\nabla_\theta \left(\theta^\top (\mathbf{X}^\top \mathbf{X})\theta - \mathbf{y}^\top (\mathbf{X}\theta) - \mathbf{y}^\top (\mathbf{X}\theta) \right) \qquad (a^\top b = b^\top a) \\
&= \frac{1}{2}\nabla_\theta \left(\theta^\top (\mathbf{X}^\top \mathbf{X})\theta - 2(\mathbf{X}^\top \mathbf{y})^\top \theta \right) \\
&= \frac{1}{2}\left(2\mathbf{X}^\top \mathbf{X}\theta - 2\mathbf{X}^\top \mathbf{y} \right) \quad (\nabla_x b^\top x = b \text{ and } \nabla_x x^\top A x = 2Ax \text{ for sym. } A) \\
&= \mathbf{X}^\top \mathbf{X}\theta - \mathbf{X}^\top \mathbf{y}
\end{aligned}$$

To minimize J, we set its derivatives to zero, and obtain the **normal equations**:

$$\mathbf{X}^\top \mathbf{X}\theta = \mathbf{X}^\top \mathbf{y} \tag{1.10}$$

Thus, the value of θ that minimizes $J(\theta)$ is given in closed form by the equation: [5]

$$\theta = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y} \tag{1.11}$$

[5] Note that in the this step, we are implicitly assuming that $\mathbf{X}^\top \mathbf{X}$ 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 $\mathbf{X}^\top \mathbf{X}$ 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 simplicty.

1.3 *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

$$y^{(i)} = \theta^\top x^{(i)} + \epsilon^{(i)}, \tag{1.12}$$

where $\epsilon^{(i)}$ is an error term that captures either unmodeled effects (such as if there are some features very pertinent to predicting housing price, but that we'd left out of the regression), or random noise. Let us further assume that the $\epsilon^{(i)}$ are distributed IID (independently and identically distributed) according to a Gaussian distribution (also called a Normal distribution) with mean zero and some variance σ^2. We can write this assumption as $\epsilon^{(i)} \sim \mathcal{N}(0, \sigma^2)$, i.e. the density of $\epsilon^{(i)}$ is given by

$$p(\epsilon^{(i)}) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(\epsilon^{(i)})^2}{2\sigma^2} \right). \tag{1.13}$$

This implies that

$$p(y^{(i)} \mid x^{(i)}; \theta) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(y^{(i)} - \theta^\top x^{(i)})^2}{2\sigma^2}\right). \tag{1.14}$$

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

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

$$L(\theta) = L(\theta; \mathbf{X}, \mathbf{y}) = p(\mathbf{y} \mid \mathbf{X}; \theta) \tag{1.15}$$

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 as

$$L(\theta) = \prod_{i=1}^{n} p(y^{(i)} \mid x^{(i)}; \theta) \tag{1.16}$$

$$= \prod_{i=1}^{n} \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(y^{(i)} - \theta^\top x^{(i)})^2}{2\sigma^2}\right). \tag{1.17}$$

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

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

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

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

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

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

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

To summarize. Under the previous probabilistic assumptions on the data, least-squares regression corresponds to finding the maximum likelihood estimate of θ. This is thus one set of assumptions under which least-squares regression can be justified as a very natural method that's just doing maximum likelihood estimation.[6]

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

[6] 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.

1.4 *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

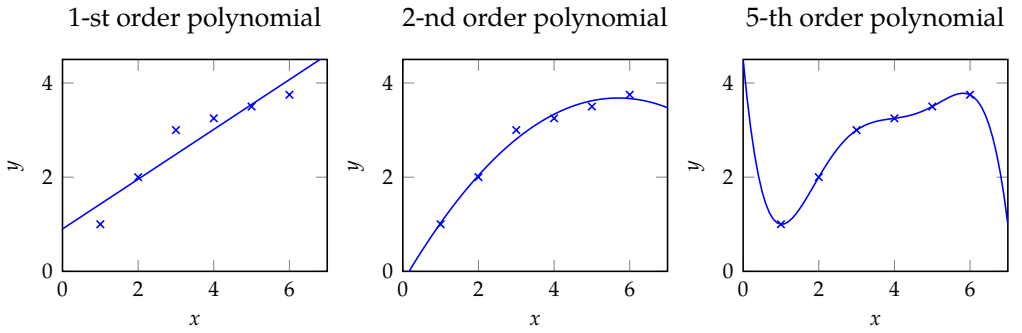


Figure 1.1. Polynomial regression with different k-order fits.

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**.[7]

[7] 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 figure 1.1, 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 briefly talk about the **locally weighted linear regression** (LWR) algorithm which, assuming there is sufficient training data, makes the choice of features less critical. This treatment will be brief, since you'll get a chance to explore some of the properties of the LWR algorithm yourself in the homework.

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

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

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

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

Here, the $w^{(i)}$'s are non-negative valued **weights**. Intuitively, if $w^{(i)}$ is large for a particular value of i, then in picking θ we'll try hard to make $(y^{(i)} - \theta^\top x^{(i)})^2$ small. If $w^{(i)}$ is small, then the $(y^{(i)} - \theta^\top x^{(i)})^2$ error term will be pretty much ignored in the fit.

A fairly standard choice for the weights is:[8]

$$w^{(i)} = \exp\left(-\frac{(x^{(i)} - x)^2}{2\tau^2}\right) \tag{1.18}$$

Note that the weights depend on the particular point x at which we're trying to evaluate x. Moreover, if $|x^{(i)} - x|$ is small, then $w^{(i)}$ is close to 1; and if $|x^{(i)} - x|$ is large, then $w^{(i)}$ is small. Hence, θ is chosen giving a much higher "weight" to the (errors on) training examples close to the query point x.[9] The parameter τ controls how quickly the weight of a training example falls off with distance of its $x^{(i)}$ from the query point x; τ is called the **bandwidth** parameter, and is also something that you'll get to experiment with in your homework.

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

[8] If x is vector-valued, the weights $w^{(i)}$ can be generalized to

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

or

$$\exp\left(-\frac{(x^{(i)} - x)^\top \Sigma^{-1} (x^{(i)} - x)}{2\tau^2}\right)$$

for appropriate choices of τ or Σ.

[9] 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.

2 *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. The class 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.

2.1 *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

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

where

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

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

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

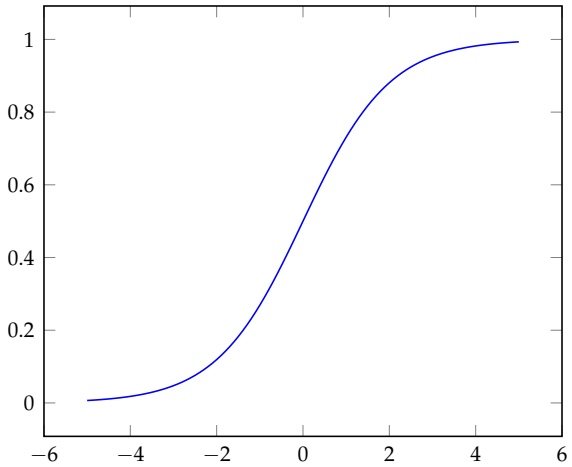


Figure 2.1. Sigmoid function (i.e. logistic).

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':

$$g'(z) = \frac{d}{dz}\frac{1}{1+e^{-z}} \tag{2.1}$$

$$= \frac{1}{(1+e^{-z})^2}(e^{-z}) \tag{2.2}$$

$$= \frac{1}{(1+e^{-z})} \cdot \left(1 - \frac{1}{(1+e^{-z})}\right) \tag{2.3}$$

$$= g(z)(1-g(z)) \tag{2.4}$$

So, given the logistic regression model, how do we fit θ for it? Following how we saw least squares regression could be derived as the maximum like-lihood 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

$$P(y = 1 \mid x; \theta) = h_\theta(x)$$
$$P(y = 0 \mid x; \theta) = 1 - h_\theta(x)$$

Note that this can be written more compactly as

$$p(y \mid x; \theta) = (h_\theta(x))^y (1 - h_\theta(x))^{1-y} \tag{2.5}$$

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

$$\begin{aligned} L(\theta) &= p(\mathbf{y} \mid X; \theta) && (2.6)\\ &= \prod_{i=1}^{n} p(y^{(i)} \mid x^{(i)}; \theta) && (2.7)\\ &= \prod_{i=1}^{n} \left(h_\theta(x^{(i)})\right)^{y^{(i)}} \left(1 - h_\theta(x^{(i)})\right)^{1-y^{(i)}} && (2.8) \end{aligned}$$

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

$$\begin{aligned} \ell(\theta) &= \log L(\theta) && (2.9)\\ &= \sum_{i=1}^{n} y^{(i)} \log h(x^{(i)}) + (1 - y^{(i)}) \log(1 - h(x^{(i)})) && (2.10) \end{aligned}$$

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:

$$\begin{aligned} \frac{\partial}{\partial \theta_j} \ell(\theta) &= \left(y \frac{1}{g(\theta^\top x)} - (1 - y) \frac{1}{1 - g(\theta^\top x)} \right) \frac{\partial}{\partial \theta_j} g(\theta^\top x) && (2.11)\\ &= \left(y \frac{1}{g(\theta^\top x)} - (1 - y) \frac{1}{1 - g(\theta^\top x)} \right) g(\theta^\top x)(1 - g(\theta^\top x)) \frac{\partial}{\partial \theta_j} \theta^\top x && (2.12)\\ &= \left(y(1 - g(\theta^\top x)) - (1 - y) g(\theta^\top x) \right) x_j && (2.13)\\ &= (y - h_\theta(x)) x_j && (2.14) \end{aligned}$$

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

$$\theta_j := \theta_j + \alpha \left(y^{(i)} - h_\theta(x^{(i)}) \right) x_j^{(i)} \tag{2.15}$$

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 $\theta^\top x^{(i)}$. 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.

2.2 *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:

$$g(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases} \tag{2.16}$$

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

$$\theta_j := \theta_j + \alpha \left(y^{(i)} - h_\theta(x^{(i)}) \right) x_j^{(i)} \tag{2.17}$$

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 predic- tions with meaningful probabilistic interpretations, or derive the perceptron as a maximum likelihood estimation algorithm.

2.3 *Another algorithm for maximizing $\ell(\theta)$*

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

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

$$\theta := \theta - \frac{f(\theta)}{f'(\theta)} \tag{2.18}$$

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

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

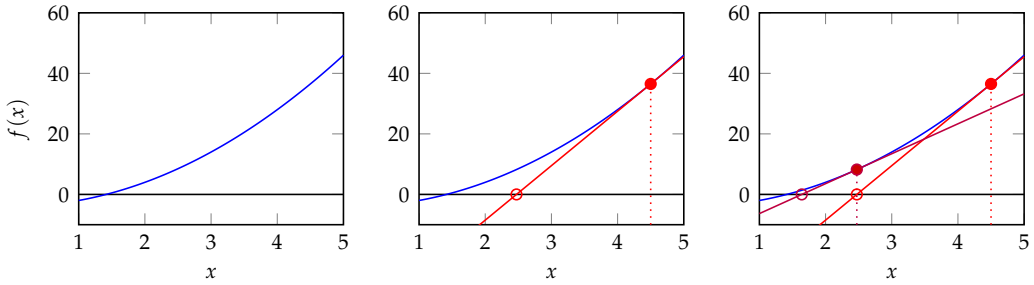


Figure 2.2. Newton's method for two steps.

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

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

$$\theta := \theta - \frac{\ell'(\theta)}{\ell''(\theta)}. \tag{2.19}$$

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

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

$$\theta := \theta - H^{-1}\nabla_\theta \ell(\theta). \tag{2.20}$$

Here, $\nabla_\theta \ell(\theta)$ is, as usual, the vector of partial derivatives of $\ell(\theta)$ with respect to the θ_i's; and H is an 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

$$Hij = \frac{\partial^2 \ell(\theta)}{\partial \theta_i \partial \theta_j}. \tag{2.21}$$

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**.

3 *Generalized Linear Models*

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

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.).

3.1 *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:

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

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

MLE w.r.t. η is concave $\rightarrow$ (neg. log-likelihood is convex)

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

We now show that the Bernoulli and the Gaussian distributions are examples of exponential family distributions. The Bernoulli distribution with mean ϕ, written $\text{Bernoulli}(\phi)$, specifies a distribution over $y \in \{0, 1\}$, so that $p(y = 1; \phi) =$

$\phi; p(y = 0; \phi) = 1 - \phi$. As we vary ϕ, we obtain Bernoulli distributions with different means. We now show that this class of Bernoulli distributions, ones obtained by varying ϕ, is in the exponential family; i.e., that there is a choice of T, a and b so that 3.1 becomes exactly the class of Bernoulli distributions.

We write the Bernoulli distribution as:

$$p(y; \phi) = \phi^y (1 - \phi)^{1-y} \tag{3.1}$$
$$= \exp(y \log \phi + (1 - y) \log(1 - \phi)) \tag{3.2}$$
$$= \exp\left(y \left(\log\left(\frac{\phi}{1 - \phi}\right)\right) + \log(1 - \phi)\right). \tag{3.3}$$

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

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

This shows that the Bernoulli distribution can be written in the form of 3.1, using an appropriate choice of T, a and b.

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

$$p(y; \mu) = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{1}{2}(y - \mu)^2\right) \tag{3.4}$$
$$= \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{1}{2}y^2\right) \cdot \exp\left(\mu y - \frac{1}{2}\mu^2\right) \tag{3.5}$$

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

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

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

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

3.2 *Constructing GLMs*

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

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

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

Inference is easy:

$$\mathbb{E}[y; \eta] = \frac{\partial}{\partial \eta} a(\eta)$$

(log partition of exp. family).

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 $h(x) = \mathbb{E}[y \mid x]$. (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 $h_\theta(x) = p(y = 1 \mid x; \theta) = 0 \cdot p(y = 0 \mid x; \theta) + 1 \cdot p(y = 1 \mid x; \theta) = \mathbb{E}[y \mid x; \theta]$.)

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

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.

3.2.1 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, μ may depend x.) So, we let the ExponentialFamily(η) 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

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

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

3.2.2 *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 $y \mid x; \theta \sim$ Bernoulli(ϕ), then $\mathbb{E}[y \mid x; \theta] = \phi$. So, following a similar derivation as the one for ordinary least squares, we get:

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

So, this gives us hypothesis functions of the form $h_\theta(x) = 1/(1 + e^{-\theta^\top x})$. 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 ($g(\eta) = \mathbb{E}[T(y); \eta]$) 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.[2]

[2] 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.

3.2.3 *Softmax Regression*

Let's look at one more example of a GLM. Consider a classification problem in which the response variable y can take on any one of k values, so $y \in \{1, 2, \ldots, k\}$.

For example, rather than classifying email into the two classes spam or not-spam—which would have been a binary classification problem— we might want to classify it into three classes, such as spam, personal mail, and work-related mail. The response variable is still discrete, but can now take on more than two values. We will thus model it as distributed according to a multinomial distribution.

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

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

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

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

Unlike our previous examples, here we do *not* have $T(y) = y$; also, $T(y)$ is now a $k-1$ dimensional vector, rather than a real number. We will write $(T(y))_i$ to denote the i-th element of the vector $T(y)$. We introduce one more very useful piece of notation. An indicator function $\mathbb{1}\{\cdot\}$ takes on a value of 1 if its argument is true, and 0 otherwise ($\mathbb{1}\{\texttt{True}\} = 1$, $\mathbb{1}\{\texttt{False}\} = 0$). For example, $\mathbb{1}\{2 = 3\} = 0$, and $\mathbb{1}\{3 = 5 - 2\} = 1$. So, we can also write the relationship between $T(y)$ and y as $(T(y))_i = \mathbb{1}\{y = i\}$. (Before you continue reading, please make sure you understand why this is true!) Further, we have that $\mathbb{E}[(T(y))_i] = P(y = i) = \phi_i$.

We are now ready to show that the multinomial is a member of the exponential family. We have:

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

where

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

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

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

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

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

$$e^{\eta_i} = \frac{\phi_i}{\phi_k} \tag{3.6}$$

$$\phi_k e^{\eta_i} = \phi_i \tag{3.7}$$

$$\phi_k\sum_{i=1}^{k} e^{\eta_i} = \sum_{i=1}^{k}\phi_i = 1 \tag{3.8}$$

This implies that $\phi_k = 1/\sum_{i=1}^{k} e^{\eta_i}$, which can be substituted back into equation (3.7) to give the response function

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

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

To complete our model, we use Assumption 3, given earlier, that the η_i's are linearly related to the x's. So, have $\eta_i = \theta_i^\top x$ (for $i = 1, \ldots, k-1$), where $\theta_1, \ldots, \theta_{k-1} \in \mathbb{R}^{d+1}$ are the parameters of our model. For notational convenience, we can also define $\theta_k = 0$, so that $\eta_k = \theta_k^\top x = 0$, as given previously. Hence, our model assumes that the conditional distribution of y given x is given by:

$$\begin{aligned} p(y = 1 \mid x; \theta) &= \phi_i && (3.9) \\ &= \frac{e^{\eta_i}}{\sum_{j=1}^{k} e^{\eta_j}} && (3.10) \\ &= \frac{e^{\theta_i^\top x}}{\sum_{j=1}^{k} e^{\theta_j^\top x}} && (3.11) \end{aligned}$$

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

Our hypothesis will output:

$$h_\theta(x) = \mathbb{E}\left[T(y) \mid x;\theta\right] \tag{3.12}$$

$$= \mathbb{E}\begin{bmatrix} \mathbb{1}\{y=1\} \\ \mathbb{1}\{y=2\} \\ \vdots \\ \mathbb{1}\{y=k-1\} \end{bmatrix} \mid x;\theta \tag{3.13}$$

$$= \begin{bmatrix} \phi_1 \\ \phi_2 \\ \vdots \\ \phi_{k-1} \end{bmatrix} \tag{3.14}$$

$$= \begin{bmatrix} \frac{\exp(\theta_1^\top x)}{\sum_{j=1}^k \exp(\theta_j^\top x)} \\ \frac{\exp(\theta_2^\top x)}{\sum_{j=1}^k \exp(\theta_j^\top x)} \\ \vdots \\ \frac{\exp(\theta_{k-1}^\top x)}{\sum_{j=1}^k \exp(\theta_j^\top x)} \end{bmatrix} \tag{3.15}$$

In other words, our hypothesis will output the estimated probability that $p(y = i \mid x;\theta)$, for every value of $i = 1,\ldots,k$. (Even though $h_\theta(x)$ as defined above is only $k-1$ dimensional, clearly $p(y = k \mid x;\theta)$ 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 $\{(x^{(i)}, y^{(i)}); i = 1,\ldots,n\}$ and would like to learn the parameters θ_i of this model, we would begin by writing down the log-likelihood

$$\ell(\theta) = \sum_{i=1}^n \log p(y^{(i)} \mid x^{(i)};\theta) \tag{3.16}$$

$$= \sum_{i=1}^n \log \prod_{l=1}^k \left(\frac{e^{\theta_l^\top x^{(i)}}}{\sum_{j=1}^k e^{\theta_j^\top x^{(i)}}} \right)^{\mathbb{1}\{y^{(i)}=l\}} \tag{3.17}$$

To obtain the second line above, we used the definition for $p(y \mid x;\theta)$ given in 3.11. We can now obtain the maximum likelihood estimate of the parameters by maximizing $\ell(\theta)$ in terms of θ, using a method such as gradient ascent or Newton's method.

Part II: Generative Learning Algorithms

From CS229 Spring 2021, Andrew Ng, Moses Charikar, & Christopher Ré, Stanford University.

So far, we've mainly been talking about learning algorithms that model $p(y \mid x; \theta)$, the conditional distribution of y given x. For instance, logistic regression modeled $p(y \mid x; \theta)$ as $h_\theta(x) = g(\theta^\top x)$ 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 $p(y \mid x)$ 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 $p(x \mid y)$ (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 $p(x \mid y = 0)$ models the distribution of dogs' features, and $p(x \mid y = 1)$ models the distribution of elephants' features.

After modeling $p(y)$ (called the **class priors**) and $p(x \mid y)$, our algorithm can then use Bayes rule to derive the posterior distribution on y given x:

$$p(y \mid x) = \frac{p(x \mid y)p(y)}{p(x)} \tag{3.18}$$

Here, the denominator is given by $p(x) = p(x \mid y = 1)p(y = 1) + p(x \mid y = 0)p(y = 0)$ (you should be able to verify that this is true from the standard prop-

erties of probabilities), and thus can also be expressed in terms of the quantities $p(x \mid y)$ and $p(y)$ that we've learned. Actually, if we're calculating $p(y \mid x)$ in order to make a prediction, then we don't actually need to calculate the denominator, since

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

4 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 $p(x \mid y)$ 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.

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:

$$p(x; \mu, \Sigma) = \frac{1}{(2\pi)^{d/2}|\Sigma|^{1/2}} \exp\left(-\frac{1}{2}(x-\mu)^\top \Sigma^{-1}(x-\mu)\right) \tag{4.1}$$

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

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

$$\mathbb{E}[X] = \int_x x p(x; \mu, \Sigma) dx = \mu \tag{4.2}$$

The **covariance** of a vector-valued random variable Z is defined as $\mathrm{Cov}(Z) = \mathbb{E}[(Z - E[Z])(Z - E[Z])^\top]$. This generalizes the notion of the variance of a real-valued random variable. The covariance can also be defined as $\mathrm{Cov}(Z) = \mathbb{E}[ZZ^\top] - (\mathbb{E}[Z])(\mathbb{E}[Z])^\top$. (You should be able to prove to yourself that these two definitions are equivalent.) If $X \sim \mathcal{N}(\mu, \Sigma)$, then

$$\mathrm{Cov}(X) = \Sigma. \tag{4.3}$$

Here are some examples of what the density of a Gaussian distribution looks like:

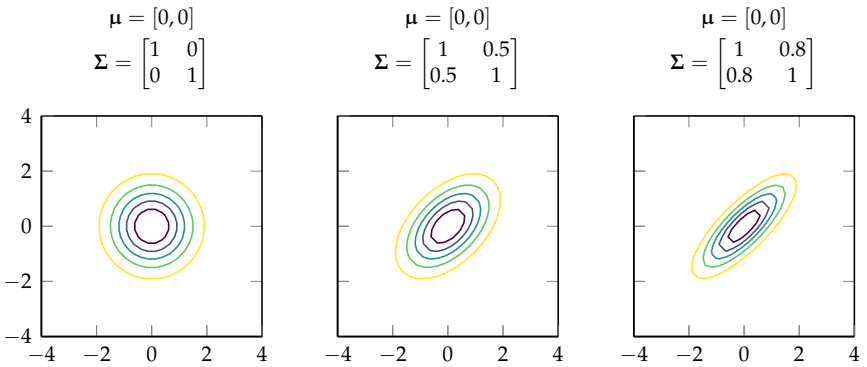
The left-most figure shows a Gaussian with mean zero (that is, the 2×1 zero-vector) and covariance matrix $\Sigma = I$ (the 2×2 identity matrix). A Gaussian with zero mean and identity covariance is also called the **standard normal distribution**. The middle figure shows the density of a Gaussian with zero mean and $\Sigma = 0.6I$; and in the rightmost figure shows one with, $\Sigma = 2I$. We see that as Σ becomes larger, the Gaussian becomes more ''spread-out,'' and as it becomes smaller, the distribution becomes more ''compressed.''

Let's look at some more examples.

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

$$\Sigma = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}; \quad \Sigma = \begin{bmatrix} 1 & 0.5 \\ 0.5 & 1 \end{bmatrix}; \quad \Sigma = \begin{bmatrix} 1 & 0.8 \\ 0.8 & 1 \end{bmatrix}. \tag{4.4}$$

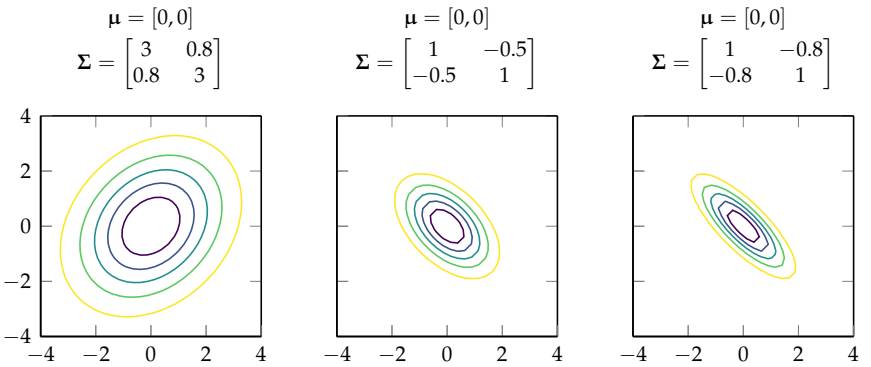
The leftmost figure shows the familiar standard normal distribution, and we see that as we increase the off-diagonal entry in Σ, the density becomes more ''compressed'' towards the 45° line (given by $x_1 = x_2$). We can see this more clearly when we look at the contours of the same three densities:



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

The plots above used, respectively,

$$\Sigma = \begin{bmatrix} 3 & 0.8 \\ 0.8 & 1 \end{bmatrix}; \quad \Sigma = \begin{bmatrix} 1 & -0.5 \\ -0.5 & 1 \end{bmatrix}; \quad \Sigma = \begin{bmatrix} 1 & -0.8 \\ -0.8 & 1 \end{bmatrix}. \tag{4.5}$$



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

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

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

$$\mu = \begin{bmatrix} 1 \\ 0 \end{bmatrix}; \quad \mu = \begin{bmatrix} -0.5 \\ 0 \end{bmatrix}; \quad \mu = \begin{bmatrix} -1 \\ -1.5 \end{bmatrix}. \tag{4.6}$$

4.1 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 $p(x \mid y)$ using a multivariate normal distribution. The model is:

$$y \sim \text{Bernoulli}(\phi) \tag{4.7}$$

$$x \mid y = 0 \sim \mathcal{N}(\mu_0, \Sigma) \tag{4.8}$$

$$x \mid y = 1 \sim \mathcal{N}(\mu_1, \Sigma) \tag{4.9}$$

Writing out the distributions, this is:

$$p(y) = \phi^y(1-\phi)^{1-y} \tag{4.10}$$

$$p(x \mid y=0) = \frac{1}{(2\pi)^{d/2}|\Sigma|^{1/2}} \exp\left(-\frac{1}{2}(x-\mu_0)^\top \Sigma^{-1}(x-\mu_0)\right) \tag{4.11}$$

$$p(x \mid y=1) = \frac{1}{(2\pi)^{d/2}|\Sigma|^{1/2}} \exp\left(-\frac{1}{2}(x-\mu_1)^\top \Sigma^{-1}(x-\mu_1)\right) \tag{4.12}$$

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

$$\ell(\phi, \mu_0, \mu_1, \Sigma) = \log \prod_{i=1}^{n} p(x^{(i)}, y^{(i)}; \phi, \mu_o, \mu_1, \Sigma) \tag{4.13}$$

$$= \log \prod_{i=1}^{n} p(x^{(i)} \mid y^{(i)}; \phi, \mu_o, \mu_1, \Sigma) p(y^{(i)}; \phi). \tag{4.14}$$

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

$$\phi = \frac{1}{n}\sum_{i=1}^{n} \mathbb{1}\{y^{(i)} = 1\} \tag{4.15}$$

$$\mu_0 = \frac{\sum_{i=1}^{n} \mathbb{1}\{y^{(i)} = 0\} x^{(i)}}{\sum_{i=1}^{n} \mathbb{1}\{y^{(i)} = 0\}} \tag{4.16}$$

$$\mu_1 = \frac{\sum_{i=1}^{n} \mathbb{1}\{y^{(i)} = 1\} x^{(i)}}{\sum_{i=1}^{n} \mathbb{1}\{y^{(i)} = 1\}} \tag{4.17}$$

$$\Sigma = \frac{1}{n}\sum_{i=1}^{n} (x^{(i)} - \mu_{y^{(i)}})(x^{(i)} - \mu_{y^{(i)}})^\top \tag{4.18}$$

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

Shown in the figure are the training set, as well as the contours of the two Gaussian distributions that have been fit to the data in each of the two classes. Note that the two Gaussians have contours that are the same shape and orientation, since they share a covariance matrix Σ, but they have different means μ_0 and μ_1. Also shown in the figure is the straight line giving the decision boundary at which $p(y = 1 \mid x) = 0.5$. 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 $y = 0$.

4.2 *Discussion: GDA and logistic regression*

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

$$p(y = 1 \mid x; \phi, \Sigma, \mu_0, \mu_1) = \frac{1}{1 + \exp(-\theta^\top x)}, \tag{4.19}$$

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

[1] 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.

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 $p(x \mid y)$ is multivariate Gaussian (with shared Σ), then $p(y \mid x)$ necessarily follows a logistic function. The converse, however, is not true; i.e., $p(y \mid x)$ being a logistic function does not imply $p(x \mid y)$ 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 $p(x \mid y)$ is indeed Gaussian (with shared Σ), then GDA is **asymptotically efficient**. Informally, this means that in the limit of very large training sets (large n), there is no algorithm that is strictly better than GDA (in terms of, say, how accurately they estimate $p(y \mid x)$). 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 be 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 $p(y \mid x)$ taking the form of a logistic function. For example, if $x \mid y = 0 \sim \text{Poisson}(\lambda_0)$, and $x \mid y = 1 \sim \text{Poisson}(\lambda_1)$, then $p(y \mid x)$ will be logistic. Logistic regression will also work well on Poisson data like this. But if we were to use GDA on such data—and fit Gaussian distributions to such non-Gaussian data—then the results will be less predictable, and GDA may (or may not) do well.

To summarize: GDA makes stronger modeling assumptions, and is more data efficient (i.e., requires less training data to learn ''well'') when the modeling

assumptions are correct or at least approximately correct. Logistic regression makes weaker assumptions, and is significantly more robust to deviations from modeling assumptions. Specifically, when the data is indeed non-Gaussian, then in the limit of large datasets, logistic regression will 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.)

5 *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

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

is used to represent an email that contains the words ''a'' and ''buy,'' but not ''aardvark,'' ''aardwolf'' or ''zygmurgy.''[1] 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 $p(x \mid y)$. 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

[1] 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

outcomes, then we'd end up with a $(2^{50000} - 1)$-dimensional parameter vector. This is clearly too many parameters.

To model $p(x \mid y)$, 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 $p(x_{2087} \mid y) = p(x_{2087} \mid y, x_{39831})$. (Note that this is not the same as saying that x_{2087} and x_{39831} are independent, which would have been written ''$p(x_{2087}) = p(x_{2087} \mid x_{39831})$''; rather, we are only assuming that x_{2087} and x_{39831} are conditionally independent given y.)

We now have:

$$
\begin{aligned}
& p(x_1, \ldots, x_{50000} \mid y) && (5.1)\\
&\quad = p(x_1 \mid y)p(x_2 \mid y, x_1)p(x_3 \mid y, x_1, x_2) \cdots p(x_{50000} \mid y, x_1, \ldots, x_{49999}) && (5.2)\\
&\quad = p(x_1 \mid y)p(x_2 \mid y)p(x_3 \mid y) \cdots p(x_{50000} \mid y) && (5.3)\\
&\quad = \prod_{j=1}^{d} p(x_j \mid y) && (5.4)
\end{aligned}
$$

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

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

$$
\mathcal{L}(\phi_y, \phi_{j|y=0}, \phi_{j|y=1}) = \prod_{i=1}^{n} p(x^{(i)}, y^{(i)}) \tag{5.5}
$$

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

$$\phi_{j|y=1} = \frac{\sum_{i=1}^{n} \mathbb{1}\{x_j^{(i)} = 1 \wedge y^{(i)} = 1\}}{\sum_{i=1}^{n} \mathbb{1}\{y^{(i)} = 1\}} \tag{5.6}$$

$$\phi_{j|y=0} = \frac{\sum_{i=1}^{n} \mathbb{1}\{x_j^{(i)} = 1 \wedge y^{(i)} = 0\}}{\sum_{i=1}^{n} \mathbb{1}\{y^{(i)} = 0\}} \tag{5.7}$$

$$\phi_y = \frac{\sum_{i=1}^{n} \mathbb{1}\{y^{(i)} = 1\}}{n} \tag{5.8}$$

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

$$p(y = 1 \mid x) = \frac{p(x \mid y = 1)p(y = 1)}{p(x)} \tag{5.9}$$

$$= \frac{\left(\prod_{j=1}^{d} p(x_j \mid y = 1)\right) p(y = 1)}{\left(\prod_{j=1}^{d} p(x_j \mid y = 1)\right) p(y = 1) + \left(\prod_{j=1}^{d} p(x_j \mid y = 0)\right) p(y = 0)}, \tag{5.10}$$

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 $\{1, 2, \ldots, k_j\}$ is straightforward. Here, we would simply model $p(x_j \mid y)$ 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 (ft^2)	< 400	$400 - 800$	$800 - 1200$	$1200 - 1600$	> 1600
x_i	1	2	3	4	5

Table 5.1. Discretized living area.

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 $p(x_j \mid y)$ 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.

5.1 *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.[2] 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

[2] NeurIPS is one of the top machine learning conferences. The deadline for submitting a paper is typically in May-June.

$$\phi_{35000|y=1} = \frac{\sum_{i=1}^{n} \mathbb{1}\{x_{35000}^{(i)} = 1 \wedge y^{(i)} = 1\}}{\sum_{i=1}^{n} \mathbb{1}\{y^{(i)} = 1\}} = 0 \tag{5.11}$$

$$\phi_{35000|y=0} = \frac{\sum_{i=1}^{n} \mathbb{1}\{x_{35000}^{(i)} = 1 \wedge y^{(i)} = 0\}}{\sum_{i=1}^{n} \mathbb{1}\{y^{(i)} = 0\}} = 0, \tag{5.12}$$

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 ''neurips''

is spam, it calculates the class posterior probabilities, and obtains

$$p(y = 1 \mid x) = \frac{\prod_{j=1}^{d} p(x_j \mid y = 1)p(y = 1)}{\prod_{j=1}^{d} p(x_j \mid y = 1)p(y = 1) + \prod_{j=1}^{d} p(x_j \mid y = 0)p(y = 0)} \tag{5.13}$$

$$= \frac{0}{0} \tag{5.14}$$

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

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

$$\phi_j = \frac{\sum_{i=1}^{n} \mathbb{1}\{z^{(i)} = j\}}{n}. \tag{5.15}$$

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

$$\phi_j = \frac{1 + \sum_{i=1}^{n} \mathbb{1}\{z^{(i)} = j\}}{k + n}. \tag{5.16}$$

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

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

$$\phi_{j|y=1} = \frac{1 + \sum_{i=1}^{n} \mathbb{1}\{x_j^{(i)} = 1 \wedge y^{(i)} = 1\}}{2 + \sum_{i=1}^{n} \mathbb{1}\{y^{(i)} = 1\}} \tag{5.17}$$

$$\phi_{j|y=0} = \frac{1 + \sum_{i=1}^{n} \mathbb{1}\{x_j^{(i)} = 1 \wedge y^{(i)} = 0\}}{2 + \sum_{i=1}^{n} \mathbb{1}\{y^{(i)} = 0\}} \tag{5.18}$$

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

5.2 *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 $p(x_j = 1 \mid y) = \phi_{j|y}$. Thus, the probability of a message was given by $p(y) \prod_{j=1}^{d} p(x_j \mid y)$

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 $\{1, \ldots, |V|\}$, where $|V|$ is the size of our vocabulary (dictionary). An email of d words is now represented by a vector $(x_1, x_2, \ldots, x_d)$ 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 $x_2 = 35000$ (if "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 ($p(x_1 \mid y)$). 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 $p(y) \prod_{j=1}^{d} p(x_j \mid y)$. 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 $x_j \mid y$ is now a multinomial, rather than a Bernoulli distribution.

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

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

$$\mathcal{L}(\phi_y, \phi_{k|y=0}, \phi_{k|y=1}) = \prod_{i=1}^{n} p(x^{(i)}, y^{(i)}) \tag{5.19}$$

$$= \prod_{i=1}^{n} \left(\prod_{j=1}^{d_i} p(x_j^{(i)} \mid y; \phi_{k|y=0}, \phi_{k|y=1}) \right) p(y^{(i)}; \phi_y). \tag{5.20}$$

Maximizing this yields the maximum likelihood estimates of the parameters:

$$\phi_{k|y=1} = \frac{\sum_{i=1}^{n} \sum_{j=1}^{d_i} \mathbb{1}\{x_j^{(i)} = k \wedge y^{(i)} = 1\}}{\sum_{i=1}^{n} \mathbb{1}\{y^{(i)} = 1\} d_i} \tag{5.21}$$

$$\phi_{k|y=0} = \frac{\sum_{i=1}^{n} \sum_{j=1}^{d_i} \mathbb{1}\{x_j^{(i)} = k \wedge y^{(i)} = 0\}}{\sum_{i=1}^{n} \mathbb{1}\{y^{(i)} = 0\} d_i} \tag{5.22}$$

$$\phi_y = \frac{\sum_{i=1}^{n} \mathbb{1}\{y^{(i)} = 1\}}{n}. \tag{5.23}$$

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:

$$\phi_{k|y=1} = \frac{1 + \sum_{i=1}^{n} \sum_{j=1}^{d_i} \mathbb{1}\{x_j^{(i)} = k \wedge y^{(i)} = 1\}}{|V| + \sum_{i=1}^{n} \mathbb{1}\{y^{(i)} = 1\} d_i} \tag{5.24}$$

$$\phi_{k|y=0} = \frac{1 + \sum_{i=1}^{n} \sum_{j=1}^{d_i} \mathbb{1}\{x_j^{(i)} = k \wedge y^{(i)} = 0\}}{|V| + \sum_{i=1}^{n} \mathbb{1}\{y^{(i)} = 0\} d_i} \tag{5.25}$$

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.

Part III: Kernel Methods

6 Kernel methods

From CS229 Fall 2020, Tengyu Ma, Moses Charikar, Andrew Ng & Christopher Ré, Stanford University.

6.1 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 $y = \theta_3 x^3 + \theta_2 x^2 + \theta_1 x + \theta_0$. It turns out that we can view the cubic function as a linear function over a different set of feature variables (defined below). Concretely, let the function $\phi : \mathbb{R} \mapsto \mathbb{R}^4$ be defined as

$$\phi(x) = \begin{bmatrix} 1 \\ x \\ x^2 \\ x^3 \end{bmatrix} \in \mathbb{R}^4. \tag{6.1}$$

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:

$$\theta_3 x^3 + \theta_2 x^2 + \theta_1 x + \theta_0 = \theta^\top \phi(x)$$

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 ϕ a **feature map**, which maps the attributes to the features.

6.2 *LMS (least mean squares) with features*

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

$$\theta := \theta + \alpha \sum_{i=1}^{n} \left(y^{(i)} - h_\theta(x^{(i)}) \right) x^{(i)} \tag{6.2}$$

$$:= \theta + \alpha \sum_{i=1}^{n} \left(y^{(i)} - \theta^\top x^{(i)} \right) x^{(i)}. \tag{6.3}$$

Let $\phi : \mathbb{R}^d \mapsto \mathbb{R}^p$ 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 $\theta^\top \phi(x)$, with θbeing 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:

$$\theta := \theta + \alpha \sum_{i=1}^{n} \left(y^{(i)} - \theta^\top \phi(x^{(i)}) \right) \phi(x^{(i)}). \tag{6.4}$$

Similarly, the corresponding stochastic gradient descent update rule is:

$$\theta := \theta + \alpha \left(y^{(i)} - \theta^\top \phi(x^{(i)}) \right) \phi(x^{(i)}). \tag{6.5}$$

6.3 *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 6.1 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 ≤ 3

$$\phi(x) = \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ \vdots \\ x_1^2 \\ x_1 x_2 \\ x_1 x_3 \\ \vdots \\ x_2 x_1 \\ \vdots \\ x_1^3 \\ x_1^2 x_2 \\ \vdots \end{bmatrix} \tag{6.6}$$

The dimension of the features $\phi(x)$ is on the order of d^3.[1] 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 for ordinary least squares updates in equation 6.3.

[1] Here, for simplicity, we include all the monomials with repetitions (so that, e.g., $x_1x_2x_3$ and $x_2x_3x_1$ both appear in $\phi(x)$). Therefore, there are totally $1 + d + d^2 + d^3$ entries in $\phi(x)$.

It may appear at first that such d^3 runtime per update and memory usage are inevitable, because the vector θ itself is of dimension $p \approx d^3$, and we may need to update every entry of θ and store it. However, we will introduce the kernel trick with which we will not need to store θ 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 in equation 6.4. The main observation is that at any time, θ can be represented as a linear combination of the vectors $\phi(x^{(1)}), \ldots, \phi(x^{(n)})$. 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, θ can be represented as

$$\theta = \sum_{i=1}^{n} \beta_i \phi(x^{(i)}) \tag{6.7}$$

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

$$\theta := \theta + \alpha \sum_{i=1}^{n} \left(y^{(i)} - \theta^\top \phi(x^{(i)}) \right) \phi(x^{(i)}) \tag{6.8}$$

$$= \sum_{i=1}^{n} \beta_i \phi(x^{(i)}) + \alpha \sum_{i=1}^{n} \left(y^{(i)} - \theta^\top \phi(x^{(i)}) \right) \phi(x^{(i)}) \tag{6.9}$$

$$= \sum_{i=1}^{n} \underbrace{\left(\beta_i + \alpha \left(y^{(i)} - \theta^\top \phi(x^{(i)}) \right) \right)}_{\text{new } \beta_i} \phi(x^{(i)}) \tag{6.10}$$

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

$$\beta_i := \beta_i + \alpha \left(y^{(i)} - \theta^\top \phi(x^{(i)}) \right) \tag{6.11}$$

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

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

We often rewrite $\phi(x^{(j)})^\top \phi(x^{(i)})$ as $\langle \phi(x^{(j)}), \phi(x^{(i)}) \rangle$ to emphasize that it's the inner product of the two feature vectors. Viewing β_i's as the new representation of θ, we have successfully translated the batch gradient descent algorithm into an algorithm that updates the value of β 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 ϕ defined in 6.6 (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:

$$\langle \phi(x), \phi(z) \rangle = 1 + \sum_{i=1}^{d} x_i z_i + \sum_{i,j \in \{1,\dots,d\}} x_i x_j z_i z_j + \sum_{i,j,k \in \{1,\dots,d\}} x_i x_j x_k z_i z_j z_k \tag{6.12}$$

$$= 1 + \sum_{i=1}^{d} x_i z_i + \left(\sum_{i=1}^{d} x_i z_i \right)^2 + \left(\sum_{i=1}^{d} x_i z_i \right)^3 \tag{6.13}$$

$$= 1 + \langle x, z \rangle + \langle x, z \rangle^2 + \langle x, z \rangle^3 \tag{6.14}$$

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 ϕ as a function that maps $\mathcal{X} \times \mathcal{X} \mapsto \mathbb{R}$ satisfying:[2]

$$K(x, z) \triangleq \langle \phi(x), \phi(z) \rangle \tag{6.15}$$

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

To wrap up the discussion, we write the 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 6.14 for all $i, j \in \{1, \dots, n\}$. Set $\beta := 0$.

2. **Loop:**

$$\forall_i \in \{1, \dots, n\}, \beta_i := \beta_i + \alpha \left(y^{(i)} - \sum_{j=1}^{n} \beta_j K(x^{(i)}, x^{(j)}) \right) \tag{6.16}$$

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

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

With the algorithm above, we can update the representation β of the vector θ efficiently with $O(n^2)$ time per update. Finally, we need to show that the knowledge of the representation β suffices to compute the prediction $\theta^\top \phi(x)$. Indeed, we have:

$$\theta^\top \phi(x) = \sum_{i=1}^{n} \beta_i \phi(x^{(i)})^\top \phi(x) = \sum_{i=1}^{n} \beta_i K(x^{(i)}, x) \tag{6.17}$$

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.

6.4 Properties of kernels

In the last subsection, we started with an explicitly defined feature map ϕ, 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 ϕ, so can the prediction of a test example x (equation 6.17.)

Therefore, it would be tempting to define other kernel functions $K(\cdot, \cdot)$ and run the algorithm 6.16. Note that the algorithm 6.16 does not need to explicitly access the feature map ϕ, and therefore we only need to ensure the existence of the feature map ϕ, but do not necessarily need to be able to explicitly write ϕ down.

What kinds of functions $K(\cdot, \cdot)$ can correspond to some feature map ϕ? In other words, can we tell if there is some feature mapping ϕ so that $K(x, z) = \phi(x)^\top \phi(z)$ 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 ϕ 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 ϕ that K corresponds to), and then we can run update rule 6.16. The benefit here is that we don't have to be able to compute ϕ 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:

$$K(x, z) = (x^\top z)^2$$

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 the feature mapping ϕ 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

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

(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_ix_j (second order) terms.

More broadly, the kernel $K(x,z) = (x^\top z + c)^k$ corresponds to a feature mapping to an $\binom{d+k}{k}$ feature space, corresponding of all monomials of the form $x_{i_1}x_{i_2}\cdots x_{i_k}$ 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 $K(x,z) = \phi(x)^\top\phi(z)$ to be large. Conversely, if $\phi(x)$ and $\phi(z)$ are far apart— say nearly orthogonal to each other—then $K(x,z) = \phi(x)^\top\phi(z)$ 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 ϕ such that the kernel K defined above satisfies $K(x,z) = \phi(x)^\top \phi(z)$? In this particular example, the answer is yes. This kernel is called the **Gaussian kernel**, and corresponds to an infinite dimensional feature mapping ϕ. 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 ϕ.

Necessary conditions for valid kernels. Suppose for now that K is indeed a valid kernel corresponding to some feature mapping ϕ, and we will first see what properties it satisfies. Now, consider some finite set of n points (not necessarily the training set) $\{x^{(1)}, \ldots, x^{(n)}\}$, 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 $K_{ij} = K(x^{(i)}, x^{(j)}) = \phi(x^{(i)})^\top \phi(x^{(j)}) = \phi(x^{(j)})^\top \phi(x^{(i)}) = K(x^{(j)}, x^{(i)}) = K_{ji}$, 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

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

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 ϕ), 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.[3]

[3] 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.

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

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

In class, we also briefly talked about a couple of other examples of kernels. For instance, consider the digit recognition problem, in which given an image (16×16 pixels) of a handwritten digit (0-9), we have to figure out which digit it was. Using either a simple polynomial kernel $K(x, z) = (x^\top z)^k$ or the Gaussian kernel, support vector machines (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 $K(x, z) = \phi(x)^\top \phi(z)$, so that we can now implicitly work in this 26^k-dimensional feature space, but without ever explicitly computing feature vectors in this space.

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. We won't 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 IV: Support Vector Machines

7 Support vector machines

From CS229 Fall 2020, Tengyu Ma, Andrew Ng, Moses Charikar, & Christopher Ré, Stanford University.

This set of notes presents the Support Vector Machine (SVM) learning al- gorithm. 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.

7.1 Margins: Intuition

We'll start our story on SVMs by talking about margins. This section will give the intuitions about margins and about the ''confidence'' of our predictions; these ideas will be made formal in Section 7.3.

Consider logistic regression, where the probability $p(y = 1 \mid x; \theta)$ is modeled by $h_\theta(x) = g(\theta^\top x)$. We then predict ''1'' on an input x if and only if $h_\theta(x) \geq 0.5$, or equivalently, if and only if $\theta^\top x \geq 0$. Consider a positive training example $(y = 1)$. The larger $\theta^\top x$ is, the larger also is $h_\theta(x) = p(y = 1 \mid x; \theta)$, 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 $\theta^\top x \gg 0$. Similarly, we think of logistic regression as confidently predicting $y = 0$, if $\theta^\top x \ll 0$. Given a training set, again informally it seems that we'd have found a good fit to the training data if we can find θ so that $\theta^\top x^{(i)} \gg 0$ whenever $y^{(i)} = 1$, and $\theta^\top x^{(i)} \ll 0$ 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 $\theta^\top x = 0$, 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 out prediction to be $y = 0$. Hence, we're much more confident about our prediction at A than at C. The point B lies in-between these two cases, and more broadly, we see that if a point is far from the separating hyperplane, then we may be significantly more confident in our predictions. Again, informally we think it 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.

7.2 Notation

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

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

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

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).

7.3 *Functional and geometric margins*

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

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

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 $w^\top x^{(i)} + b$ to be a large positive number. Conversely, if $y^{(i)} = -1$, then for the functional margin to be large, we need $w^\top x^{(i)} + b$ to be a large negative number. Moreover, if $y^{(i)}(w^\top x^{(i)} + b) > 0$, 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 $g(w^\top x + b) = g(2w^\top x + 2b)$, 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 $w^\top x + b$. However, replacing (w, b) with $(2w, 2b)$ also results in multiplying our functional margin by a factor of 2. Thus, it seems that by exploiting our freedom to scale w and b, we can make the functional margin arbitrarily large without really changing anything meaningful. Intuitively, it might therefore make sense to impose some sort of normalization condition such as that $\|w\|_2 = 1$; i.e., we might replace (w, b) with $(w/\|w\|_2, b/\|w\|_2)$, and instead consider the functional margin of $(w/\|w\|_2, b/\|w\|_2)$. We'll come back to this later.

Given a training set $S = \{(x^{(i)}, y^{(i)}); i = 1, \ldots, n\}$, 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:

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

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°) 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 $w^\top x + b = 0$. Hence,

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

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

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

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

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

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

Finally, given a training set $S = \{(x^{(i)}, y^{(i)}); i = 1, \ldots, n\}$, 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:

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

7.4 The optimal margin classifier

Given a training set, it seems from our previous discussion that a natural desideratum is to try to find a decision boundary that maximizes the (geometric) margin, since this would reflect a very confident set of predictions on the training set and a good ''fit'' to the training data. Specifically, this will result in a classifier that separates the positive and the negative training examples with a ''gap'' (geometric margin).

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

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

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

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

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

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:

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

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.[1]

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

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

7.5 *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:

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

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

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

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

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

and solve for w and β.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Now, let's look at a slightly different problem. We define

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

Here, the "$\mathcal{D}$" subscript stands for "dual." Note also that whereas in the definition of $\theta_{\mathcal{P}}$ we were optimizing (maximizing) with respect to α, β, here 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 $d^* = \max_{\alpha,\beta:\alpha_i \geq 0} \theta_{\mathcal{D}}(w)$.

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

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

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

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

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

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

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

$$\frac{\partial}{\partial w_i}\mathcal{L}(w^*, \alpha^*, \beta^*) = 0, \quad i = 1, \ldots, d \tag{7.1}$$

$$\frac{\partial}{\partial \beta_i}\mathcal{L}(w^*, \alpha^*, \beta^*) = 0, \quad i = 1, \ldots, l \tag{7.2}$$

$$\alpha_i^* g_i(w^*) = 0, \quad i = 1, \ldots, k \tag{7.3}$$

$$g_i(w^*) \leq 0, \quad i = 1, \ldots, k \tag{7.4}$$

$$\alpha^* \geq 0, \quad i = 1, \ldots, k \tag{7.5}$$

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

We draw attention to Equation (7.3), 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.

7.6 Optimal margin classifiers

***Note**: The equivalence of optimization problem 7.6 and the optimization problem 7.11, and the relationship between the primary and dual variables in equation 7.8 are the most important take home messages of this section.*

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

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

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

$$\min_{w,b} \quad \frac{1}{2}\|w\|^2 \tag{7.6}$$

$$\text{s.t.} \quad y^{(i)}(w^\top x^{(i)} + b) \geq 1, \quad i = 1, \ldots, n \tag{7.7}$$

We can write the constraints as

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

We have one such constraint for each training example. Note that from the KKT dual complementarity condition, we will have $\alpha_i > 0$ only for the training examples that have functional margin exactly equal to one (i.e., the ones corresponding to constraints that hold with equality, $g_i(w) = 0$). Consider the figure below, in which a maximum margin separating hyperplane is shown by the solid line.

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

Let's move on. Looking ahead, as we develop the dual form of the problem, one key idea to watch out for is that we'll try to write our algorithm in terms of only the inner product $\langle x^{(i)}, x^{(j)} \rangle$ (think of this as $(x^{(i)})^\top x^{(j)}$) 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:

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

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

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

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

This implies that

$$w = \sum_{i=1}^{n} \alpha_i y^{(i)} x^{(i)}. \tag{7.8}$$

As for the derivative with respect to b, we obtain

$$\frac{\partial}{\partial b} \mathcal{L}(w, b, \alpha) = \sum_{i=1}^{n} \alpha_i y^{(i)} = 0. \tag{7.9}$$

If we take the definition of w in Equation (7.8) and plug that back into the Lagrangian (Section 7.6), and simplify, we get

$$\mathcal{L}(w, b, \alpha) = \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{n} y^{(i)} y^{(j)} \alpha_i \alpha_j (x^{(i)})^\top x^{(j)} - b \sum_{i=1}^{n} \alpha_i y^{(i)}. \tag{7.10}$$

But from Equation (7.9), the last term must be zero, so we obtain

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

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 from equation (7.9), we obtain the following dual optimization problem:

$$\max_{\alpha} \quad W(\alpha) = \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{n} y^{(i)} y^{(j)} \alpha_i \alpha_j \langle x^{(i)}, x^{(j)} \rangle. \tag{7.11}$$

$$\text{s. t.} \quad \alpha_i \geq 0, \quad i = 1, \ldots, n \tag{7.12}$$

$$\sum_{i=1}^{n} \alpha_i y^{(i)} = 0. \tag{7.13}$$

You should also be able to verify that the conditions required for $p^* = d^*$ and the KKT conditions (Equations (7.1) to (7.5)) to hold are indeed satisfied in our optimization problem. Hence, we can solve the dual in lieu of solving the primal problem. Specifically, in the dual problem above, we have a maximization problem in which the parameters are the α_i's. We'll talk later about the specific algorithm that we're going to use to solve the dual problem, but if we are indeed able to solve it (i.e., find the α's that maximize $W(\alpha)$ subject to the constraints), then we can use Equation (7.8) to go back and find the optimal w's as a function of the α's. Having found w^* , by considering the primal problem, it is also straightforward to find the optimal value for the intercept term b as

$$b^* = -\frac{\max_{i:y^{(i)}=-1} w^{*\top} x^{(i)} + \min_{i:y^{(i)}=1} w^{*\top} x^{(i)}}{2}. \tag{7.14}$$

(Check for yourself that this is correct.)

Before moving on, let's also take a more careful look at Equation (7.8), which gives the optimal value of w in terms of (the optimal value of) α. Suppose we've fit our model's parameters to a training set, and now wish to make a prediction at a new point input x. We would then calculate $w^\top x + b$, and predict $y = 1$ if and only if this quantity is bigger than zero. But using equation (7.8), this quantity can also be written:

$$w^\top x + b = \left(\sum_{i=1}^{n} \alpha_i y^{(i)} x^{(i)} \right)^\top x + b \tag{7.15}$$

$$= \sum_{i=1}^{n} \alpha_i y^{(i)} \langle x^{(i)}, x \rangle + b. \tag{7.16}$$

Hence, if we've found the α_i's, in order to make a prediction, we have to calculate a quantity that depends only on the inner product between x and the points in the training set. Moreover, we saw earlier that the α_i's will all be zero except for the support vectors. Thus, many of the terms in the sum above will be zero, and we really need to find only the inner products between x and the support vectors (of which there is often only a small number) in order calculate equation (7.16) and make our prediction.

By examining the dual form of the optimization problem, we gained significant insight into the structure of the problem, and were also able to write the entire algorithm in terms of only inner products between input feature vectors. In the

next section, we will exploit this property to apply the kernels to our classification problem. The resulting algorithm, **support vector machines**, will be able to efficiently learn in very high dimensional spaces.

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

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

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

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:

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

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

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

As before, we also have that w can be expressed in terms of the α_i 's as given in equation (7.8), so that after solving the dual problem, we can continue to use equation (7.16) to make our predictions. Note that, somewhat surprisingly, in adding ℓ_1 regularization, the only change to the dual problem is that what was originally a constraint that $0 \leq \alpha_i$ has now become $0 \leq \alpha_i \leq C$. The calculation for b^* also has to be modified (equation (7.14) 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:

$$\alpha_i = 0 \implies y^{(i)}(w^\top x^{(i)} + b) \geq 1 \tag{7.17}$$

$$\alpha_i = C \implies y^{(i)}(w^\top x^{(i)} + b) \leq 1 \tag{7.18}$$

$$0 < \alpha_i < C \implies y^{(i)}(w^\top x^{(i)} + b) = 1. \tag{7.19}$$

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

7.8 *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.

7.8.1 Coordinate ascent

Consider trying to solve the unconstrained optimization problem

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

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

repeat
  for $i = 1, \ldots, n$ **do**
    $\alpha_i := \arg\max_{\hat{\alpha}_i} W(\alpha_1, \ldots, \alpha_{i-1}, \hat{\alpha}_i, \alpha_{i+1}, \ldots, \alpha_n)$.
  end for
until convergence

Algorithm 7.1. Coordinate ascent.

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

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

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

7.9 SMO

We close off the discussion of SVMs by sketching the derivation of the SMO algorithm.

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

$$\max_{\alpha} \quad W(\alpha) = \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{n} y^{(i)} y^{(j)} \alpha_i \alpha_j \langle x^{(i)}, x^{(j)} \rangle. \tag{7.20}$$

$$\text{s.t.} \quad 0 \leq \alpha_i \leq C, \quad i = 1, \ldots, n \tag{7.21}$$

$$\sum_{i=1}^{n} \alpha_i y^{(i)} = 0.(33) \tag{7.22}$$

Let's say we have set of α_i's that satisfy the constraints in equations (7.21) and (7.22). Now, suppose we want to hold $\alpha_2, \ldots, \alpha_n$ fixed, and take a coordinate ascent step and reoptimize the objective with respect to α_1. Can we make any progress? The answer is no, because the constraint 7.22 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, α_1 is exactly determined by the other α_i's, and if we were to hold $\alpha_2, \ldots, \alpha_n$ fixed, then we can't make any change to α_1 without violating the constraint 7.22 in the optimization problem.

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

To test for convergence of this algorithm, we can check whether the KKT conditions (equations (7.17) to (7.19)) are satisfied to within some *tol*. Here, *tol* is the convergence tolerance parameter, and is typically set to around 0.01 to 0.001. (See the paper and pseudocode for details.)

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

repeat

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

until convergence

Algorithm 7.2. SMO algorithm.

Let's say we currently have some setting of the α_i's that satisfy the constraints 7.21–7.22, and suppose we've decided to hold $\alpha_3, \ldots, \alpha_n$ fixed, and want to reoptimize $W(\alpha_1, \alpha_2, \ldots, \alpha_n)$ with respect to α_1 and α_2 (subject to the constraints). From equation (7.22), 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 $\alpha_3, \ldots \alpha_n$), we can just let it be denoted by some constant ζ:

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

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

From the constraints 7.21, we know that α_1 and α_2 must lie within the box $[0, C] \times [0, C]$ shown. Also plotted is the line $\alpha_1 y^{(1)} + \alpha_2 y^{(2)} = \zeta$, on which we know α_1 and α_2 must lie. Note also that, from these constraints, we know $L \leq \alpha_2 \leq H$; otherwise, (α_1, α_2) can't simultaneously satisfy both the box and the straight line constraint. In this example, $L = 0$. But depending on what the line $\alpha_1 y^{(1)} + \alpha_2 y^{(2)} = \zeta$ looks like, this won't always necessarily be the case; but more generally, there will be some lower-bound L and some upper-bound H on the permissible values for α_2 that will ensure that α_1, α_2 lie within the box $[0, C] \times [0, C]$.

Using section 7.9, we can also write α_1 as a function of α_2:

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

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

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

Treating $\alpha_3, \ldots, \alpha_n$ as constants, you should be able to verify that this is just some quadratic function in α_2. I.e., this can also be expressed in the form $a\alpha_2^2 + b\alpha_2 + c$ for some appropriate a, b, and c. If we ignore the "box" constraints 7.21 (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 $\alpha_2^{\text{new,unclipped}}$ denote the resulting value of α_2. You should also be able to convince yourself that if we had instead wanted to maximize W with respect to α_2 but subject to the box constraint, then we can find the resulting value optimal simply by taking $\alpha_2^{\text{new,unclipped}}$ and "clipping" it to lie in the $[L, H]$ interval, to get

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

Finally, having found the α_2^{new}, we can use section 7.9 to go back and find the optimal value of α_1^{new}.

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

Part V: 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.

From CS229 Fall 2020, Tengyu Ma, Anand Avati, Kian Katanforoosh, Andrew Ng, Moses Charikar, & Christopher Ré, Stanford University.

8 Supervised Learning with Non-Linear Models

In the supervised learning setting (predicting y from the input x), suppose our model/hypothesis is $h_\theta(x)$. In the past lectures, we have considered the cases when $h_\theta(x) = \theta^\top x$ (in linear regression or logistic regression) or $h_\theta(x) = \theta^\top \phi(x)$ (where $\phi(x)$ is the feature map). A commonality of these two models is that they are linear in the parameters θ. Next we will consider learning general family of models that are **non-linear in both** the parameters θ and the inputs x. The most common non-linear models are *neural networks*, which we will define staring from the next section. For this section, it suffices to think $h_\theta(x)$ as an abstract non-linear model.[1]

[1] If a concrete example is helpful, perhaps think about the model $h_\theta(x) = \theta_1^2 x_1^2 + \theta_2^2 x_2^2 + \cdots + \theta_d^2 x_d^2$ in this subsection, even though it's not a neural network.

Suppose $\{(x^{(i)}, y^{(i)})\}_{i=1}^n$ are the training examples. For simplicity, we start with the case where $y^{(i)} \in \mathbb{R}$ and $h_\theta(x) \in \mathbb{R}$.

Cost/loss function. We define the least square cost function for the i-th example $(x^{(i)}, y^{(i)})$ as

$$J^{(i)}(\theta) = \frac{1}{2}\left(h_\theta(x^{(i)}) - y^{(i)}\right)^2 \tag{8.1}$$

and define the mean-square cost function for the dataset as

$$J(\theta) = \frac{1}{n}\sum_{i=1}^{n} J^{(i)}(\theta) \tag{8.2}$$

which is same as in linear regression except that we introduce a constant $1/n$ in front of the cost function to be consistent with the convention. Note that multiplying the cost function with a scalar will not change the local minima or global minima of the cost function. Also note that the underlying parameterization for

$h_\theta(x)$ is different from the case of linear regression, even though the form of the cost function is the same mean-squared loss. Throughout the notes, we use the words ''loss'' and ''cost'' interchangeably.

***Optimizers* (*SGD*).** Commonly, people use gradient descent (GD), stochastic gradient (SGD), or their variants to optimize the loss function $J(\theta)$. GD's update rule can be written as[2]

$$\theta := \theta - \alpha \nabla_\theta J(\theta) \tag{8.3}$$

where $\alpha > 0$ is often referred to as the *learning rate* or *step size*. Next, we introduce a version of the SGD (algorithm 8.1), which is slightly different from that in the first lecture notes. Oftentimes computing the gradient of B examples simultaneously for the parameter θ can be faster than computing B gradients separately due to hardware parallelization. Therefore, a mini-batch version of SGD is most commonly used in deep learning, as shown in algorithm 8.2. There are also other variants of the SGD or mini-batch SGD with slightly different sampling schemes.

[2] Recall that, as defined in the previous lecture notes, 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.

Algorithm 8.1. Stochastic gradient descent.

Hyperparameter: learning rate α, number of total iteration n_{iter}.
Initialize θ randomly.
for $i = 1$ to n_{iter} **do**
  Sample j uniformly from $1, \ldots, n$, and update θ by

$$\theta := \theta - \alpha \nabla_\theta J^{(j)}(\theta)$$

end for

With these generic algorithms, a typical deep learning model is learned with the following steps:

1. Define a neural network parametrization $h_\theta(x)$, which we will introduce in chapter 9.
2. Write the backpropagation algorithm to compute the gradient of the loss function $J^{(j)}(\theta)$ efficiently, which will be covered in chapter 10.
3. Run SGD or mini-batch SGD (or other gradient-based optimizers) with the loss function $J(\theta)$.

Hyperparameter: learning rate α, batch size B, # iteration n_{iter}.
Initialize θ randomly.
for $i = 1$ to n_{iter} **do**
Sample j uniformly from $1, \ldots, n$, and update θ by
Sample B examples $j_1, \ldots, j_B$ (without replacement) uniformly from $\{1, \ldots, n\}$, and update θ by

$$\theta := \theta - \frac{\alpha}{B} \sum_{k=1}^{B} \nabla_\theta J^{(j_k)}(\theta)$$

end for

Algorithm 8.2. Mini-batch stochastic gradient descent

9 *Neural Networks*

Neural networks refer to broad type of non-linear models/parametrizations $h_\theta(x)$ that involve combinations of matrix multiplications and other entrywise non-linear operations. We will start small and slowly build up a neural network, step by step.

A neural network with a single neuron. Recall the housing price prediction problem from before: given the size of the house, we want to predict the price. We will use it as a running example in this subsection.

Previously, we fit a straight line to the graph of size vs. housing price. Now, instead of fitting a straight line, we wish to prevent negative housing prices by setting the absolute minimum price as zero. This produces a ''kink'' in the graph as shown in figure 9.1. How do we represent such a function with a single kink as $h_\theta(x)$ with unknown parameter? (After doing so, we can invoke the machinery in part V.)

We define a parameterized function $h_\theta(x)$ with input x, parameterized by θ, which outputs the price of the house y. Formally, $h_\theta : x \mapsto y$. Perhaps one of the simplest parametrization would be

$$h_\theta(x) = \max(wx + b, 0), \quad \text{where } \theta = (w, b) \in \mathbb{R}^2 \tag{9.1}$$

Here $h_\theta(x)$ returns a single value: $(wx + b)$ or zero, whichever is greater. In the context of neural networks, the function $\max\{t, 0\}$ is called a ReLU (pro- nounced ''ray-lu''), or *rectified linear unit*, and often denoted by $\text{ReLU}(t) \triangleq \max\{t, 0\}$.

Generally, a one-dimensional non-linear function that maps $\mathbb{R}$ to $\mathbb{R}$ such as ReLU is often referred to as an **activation function**. The model $h_\theta(x)$ is said to have a single neuron partly because it has a single non-linear activation function. (We will discuss more about why a non-linear activation is called neuron.)

When the input $x \in \mathbb{R}^d$ has multiple dimensions, a neural network with a single neuron can be written as

$$h_\theta(x) = \text{ReLU}(w^\top x + b), \quad \text{where } w \in \mathbb{R}^d, b \in \mathbb{R}, \text{ and } \theta = (w, b) \tag{9.2}$$

Figure 9.1. Housing prices with a ''kink'' in the graph.

The term b is often referred to as the ''bias'', and the vector w is referred to as the weight vector. Such a neural network has 1 layer. (We will define what multiple layers mean in the sequel.)

Stacking neurons. 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 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 9.2). 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.

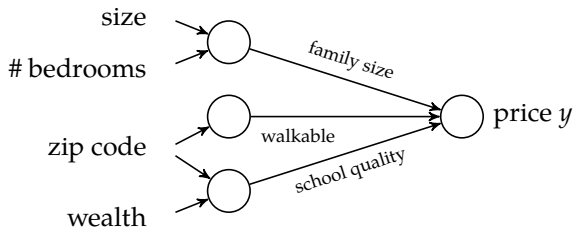


Figure 9.2. Diagram of a small neural network for predicting housing prices.

Formally, the input to a neural network is a set of input features x_1, x_2, x_3, x_4. We denote the intermediate variables for ''family size'', ''walk- able'', and ''school quality'' by a_1, a_2, a_3 (these a_i's are often referred to as ''hidden units'' or ''hidden neurons''). We represent each of the a_i's as a neural network with a single neuron with a subset of $x_1, \ldots, x_4$ as inputs. Then as in figure 9.1, we will have the parameterization:

$$\begin{aligned} a_1 &= \mathrm{ReLU}(\theta_1 x_1 + \theta_2 x_2 + \theta_3) \\ a_2 &= \mathrm{ReLU}(\theta_4 x_3 + \theta_5) \\ a_3 &= \mathrm{ReLU}(\theta_6 x_3 + \theta_7 x_4 + \theta_8) \end{aligned}$$

where $(\theta_1, \ldots, \theta_8)$ are parameters. Now we represent the final output $h_\theta(x)$ as another linear function with a_1, a_2, a_3 as inputs, and we get[1]

$$h_\theta(x) = \theta_9 a_1 + \theta_{10} a_2 + \theta_{11} a_3 + \theta_{12} \tag{9.3}$$

where θ contains all the parameters $(\theta_1, \ldots, \theta_{12})$.

Now we represent the output as a quite complex function of x with parameters θ. Then you can use this parametrization h_θ with the machinery of part V to learn the parameters θ.

[1] Typically, for multi-layer neural network, at the end, near the output, we don't apply ReLU, especially when the output is not necessarily a positive number.

Inspiration from biological neural networks. As the name suggests, artificial neural networks were inspired by biological neural networks. The hidden units $a_1, \ldots, a_m$ correspond to the neurons in a biological neural network, and the

parameters θ_i's correspond to the synapses. However, it's unclear how similar the modern deep artificial neural networks are to the biological ones. For example, perhaps not many neuroscientists think biological neural networks could have 1000 layers, while some modern artificial neural networks do (we will elaborate more on the notion of layers.) Moreover, it's an open question whether human brains update their neural networks in a way similar to the way that computer scientists learn artificial neural networks (using backpropagation, which we will introduce in the next section.).

Two-layer fully-connected neural networks. We constructed the neural network in equation (9.3) using a significant amount of prior knowledge/belief about how the ''family size'', ''walkable'', and ''school quality'' are determined by the inputs. We implicitly assumed that we know the family size is an important quantity to look at and that it can be determined by only the ''size'' and ''# bedrooms''. Such a prior knowledge might not be available for other applications. It would be more flexible and general to have a generic parameterization. A simple way would be to write the intermediate variable a_1 as a function of all $x_1, \dots, x_4$:

$$\begin{aligned} a_1 &= \mathrm{ReLU}(w_1^\top x + b_1), \quad \text{where } w_1 \in \mathbb{R}^4 \text{ and } b_1 \in \mathbb{R} \\ a_2 &= \mathrm{ReLU}(w_2^\top x + b_2), \quad \text{where } w_2 \in \mathbb{R}^4 \text{ and } b_2 \in \mathbb{R} \\ a_3 &= \mathrm{ReLU}(w_3^\top x + b_3), \quad \text{where } w_3 \in \mathbb{R}^4 \text{ and } b_3 \in \mathbb{R} \end{aligned}$$

We still define $h_\theta(x)$ using equation (9.3) with a_1, a_2, a_3 being defined as above. Thus we have a so-called **fully-connected neural network** as visualized in the dependency graph in figure 9.3 because all the intermediate variables a_i's depend on all the inputs x_i's.

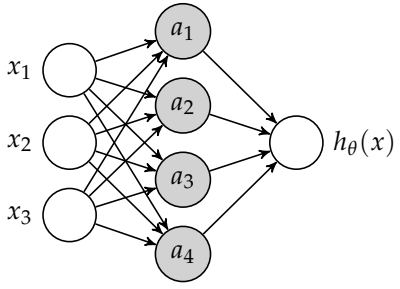


Figure 9.3. Diagram of a two-layer fully connected neural network. Each edge from node x_i to node a_j indicates that a_j depends on x_i. The edge from x_i to a_j is associated with the weight $(w_j^{[1]})_i$ which denotes the i-th coordinate of the vector $w_j^{[1]}$. The activation a_j can be computed by taking the ReLU of the weighted sum of x_i's with the weights being the weights associated with the incoming edges, that is, $a_j = \mathrm{ReLU}\left(\sum_{i=1}^d (w_j^{[1]})_i x_i\right)$.

For full generality, a two-layer fully-connected neural network with m hidden units and d dimensional input $x \in \mathbb{R}^d$ is defined as

$$\forall j \in [1, \ldots, m], \quad z_j = w_j^{[1]\top} x + b_j^{[1]} \text{ where } w_j^{[1]} \in \mathbb{R}^d, b_j^{[1]} \in \mathbb{R} \tag{9.4}$$

$$a_j = \text{ReLU}(z_j) \tag{9.5}$$

$$a = [a_1, \ldots, a_m]^\top \in \mathbb{R}^m \tag{9.6}$$

$$h_\theta(x) = w^{[2]\top} a + b^{[2]} \text{ where } w^{[2]} \in \mathbb{R}^m, b^{[2]} \in \mathbb{R} \tag{9.7}$$

Note that by default the vectors in $\mathbb{R}^d$ are viewed as column vectors, and in particular a is a column vector with components $a_1, a_2, \ldots, a_m$. The indices $^{[1]}$ and $^{[2]}$ are used to distinguish two sets of parameters: the $w_j^{[1]}$'s (each of which is a vector in $\mathbb{R}^d$) and $w^{[2]}$ (which is a vector in $\mathbb{R}^m$). We will have more of these later.

Vectorization. Before we introduce neural networks with more layers and more complex structures, we will simplify the expressions for neural networks with more matrix and vector notations. Another important motivation of vectorization is the speed perspective in the implementation. In order to implement a neural network efficiently, one must be careful when using for loops. The most natural way to implement equation (9.4) in code is perhaps to use a for loop. In practice, the dimensionalities of the inputs and hidden units are high. As a result, code will run very slowly if you use for loops. Leveraging the parallelism in GPUs is/was crucial for the progress of deep learning.

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.

We vectorize the two-layer fully-connected neural network as below. We define a weight matrix $W^{[1]}$ in $\mathbb{R}^{m \times d}$ as the concatenation of all the vectors $w_j^{[1]}$'s in the

following way:

$$W^{[1]} = \begin{bmatrix} —{w_1^{[1]}}^\top — \\ —{w_2^{[1]}}^\top — \\ \vdots \\ —{w_m^{[1]}}^\top — \end{bmatrix} \in \mathbb{R}^{m \times d}$$

Now by the definition of matrix vector multiplication, we can write $z = [z_1, \ldots, z_m]\top \in \mathbb{R}^m$ as:

$$\underbrace{\begin{bmatrix} z_1 \\ \vdots \\ \vdots \\ z_m \end{bmatrix}}_{z \in \mathbb{R}^{m \times 1}} = \underbrace{\begin{bmatrix} —{w_1^{[1]}}^\top — \\ —{w_2^{[1]}}^\top — \\ \vdots \\ —{w_m^{[1]}}^\top — \end{bmatrix}}_{W^{[1]} \in \mathbb{R}^{m \times d}} \underbrace{\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ z_d \end{bmatrix}}_{x \in \mathbb{R}^{d \times 1}} + \underbrace{\begin{bmatrix} b_1^{[1]} \\ x_2^{[1]} \\ \vdots \\ b_m^{[1]} \end{bmatrix}}_{b^{[1]} \in \mathbb{R}^{m \times 1}}$$

Or succinctly,

$$z = W^{[1]} x + b^{[1]} \tag{9.8}$$

We remark again that a vector in $\mathbb{R}^d$ in these notes, following the conventions previously established, is automatically viewed as a column vector, and can also be viewed as a $d \times 1$ dimensional matrix. (Note that this is different from `numpy` where a vector is viewed as a row vector in broadcasting.)

Computing the activations $a \in \mathbb{R}^m$ from $z \in \mathbb{R}^m$ involves an element-wise non-linear application of the ReLU function, which can be computed in parallel efficiently. Overloading ReLU for element-wise application of ReLU (meaning, for a vector $t \in \mathbb{R}^d$, $\text{ReLU}(t)$ is a vector such that $\text{ReLU}(t)_i = \text{ReLU}(t_i)$), we have:

$$a = \text{ReLU}(z) \tag{9.9}$$

Define $W^{[2]} = [{w^{[2]}}^\top] \in \mathbb{R}^{1 \times m}$ similarly. Then, the model in equation (9.7) can be summarized as:

$$a = \text{ReLU}(W^{[1]} x + b^{[1]}) \tag{9.10}$$

$$h_\theta(x) = W^{[2]} a + b^{[2]} \tag{9.11}$$

Here θ consists of $W^{[1]}, W^{[2]}$ (often referred to as the weight matrices) and $b^{[1]}, b^{[2]}$ (referred to as the biases). The collection of $W^{[1]}, b^{[1]}$ is referred to as the first layer, and $W^{[2]}, b^{[2]}$ the second layer. The activation a is referred to as the hidden layer. A two-layer neural network is also called one-hidden-layer neural network.

Multi-layer fully-connected neural networks. With this succinct notations, we can stack more layers to get a deeper fully-connected neural network. Let r be the number of layers (weight matrices). Let $W^{[1]}, \ldots, W^{[r]}, b^{[1]}, \ldots, b^{[r]}$ be the weight matrices and biases of all the layers. Then a multi-layer neural network can be written as:

$$a^{[1]} = \mathrm{ReLU}(W^{[1]}x + b^{[1]}) \tag{9.12}$$
$$a^{[2]} = \mathrm{ReLU}(W^{[2]}a^{[1]} + b^{[2]}) \tag{9.13}$$
$$\cdots \tag{9.14}$$
$$a^{[r-1]} = \mathrm{ReLU}(W^{[r-1]}a^{[r-2]} + b^{[r-1]}) \tag{9.15}$$
$$h_\theta(x) = W^{[r]}a^{[r-1]} + b^{[r]} \tag{9.16}$$

We note that the weight matrices and biases need to have compatible dimensions for the equations above to make sense. If $a^{[k]}$ has dimension m_k , then the weight matrix $W^{[k]}$ should be of dimension $m_k \times m_{k-1}$, and the bias $b^{[k]} \in \mathbb{R}^{m_k}$. Moreover, $W^{[1]} \in \mathbb{R}^{m_1 \times d}$ and $W^{[r]} \in \mathbb{R}^{1 \times m_{r-1}}$.

The total number of neurons in the network is $m_1 + \cdots + m_r$, and the total number of parameters in this network is $(d+1)m_1 + (m_1+1)m_2 + \cdots + (m_{r-1}+1)m_r$.

Sometimes for notational consistency we also write $a^{[0]} = x$, and $a^{[r]} = h_\theta(x)$. Then we have simple recursion that

$$a^{[k]} = \mathrm{ReLU}(W^{[k]}a^{[k-1]} + b^{[k]}), \quad \forall k = 1, \ldots, r-1 \tag{9.17}$$

Note that this would have be true for $k = r$ if there were an additional ReLU in equation (9.16), but often people like to make the last layer linear (aka without a ReLU) so that negative outputs are possible and it's easier to interpret the last layer as a linear model. (More on the interpretability at the ''connection to kernel method'' paragraph of this section.)

Other activation functions. The activation function ReLU can be replaced by many other non-linear function $\sigma(\cdot)$ that maps $\mathbb{R}$ to $\mathbb{R}$ such as:

$$\sigma(z) = \frac{1}{1+e^{-z}} \tag{sigmoid}$$
$$\sigma(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \tag{tanh}$$

Why do we not use the identity function for $\sigma(z)$? That is, why not use $\sigma(z) = z$? Assume for sake of argument that $b^{[1]}$ and $b^{[2]}$ are zeros. Suppose $\sigma(z) = z$, then for two-layer neural network, we have that

$$\begin{aligned} h_\theta(x) &= W^{[2]}a^{[1]} \\ &= W^{[2]}\sigma(z^{[1]}) && \text{(by definition)} \\ &= W^{[2]}z^{[1]} && \text{(since } \sigma(z) = z\text{)} \\ &= W^{[2]}W^{[1]}x && \text{(from chapter 9)} \\ &= \tilde{W}x && \text{(where } \tilde{W} = W^{[2]}W^{[1]}\text{)} \end{aligned}$$

Notice how $W^{[2]}W^{[1]}$ collapsed into $\tilde{W}$.

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

Connection to the Kernel Method. In the previous lectures, we covered the concept of feature maps. Recall that the main motivation for feature maps is to represent functions that are non-linear in the input x by $\theta^\top\phi(x)$, where θ are the parameters and $\phi(x)$, the feature map, is a handcrafted function non-linear in the raw input x. The performance of the learning algorithms can significantly depends on the choice of the feature map $\phi(x)$. Oftentimes people use domain knowledge to design the feature map $\phi(x)$ that suits the particular applications. The process of choosing the feature maps is often referred to as **feature engineering**.

We can view deep learning as a way to automatically learn the right feature map (sometimes also referred to as ''the representation'') as follows. Suppose we denote by β the collection of the parameters in a fully-connected neural networks (equation (9.16)) except those in the last layer. Then we can abstract right $a^{[r-1]}$ as a function of the input x and the parameters in β : $a^{[r-1]} = \phi_\beta(x)$. Now we can write the model as:

$$h_\theta(x) = W^{[r]}\phi_\beta(x) + b^{[r]} \tag{9.18}$$

When β is fixed, then $\phi_\beta(\cdot)$ can viewed as a feature map, and therefore $h_\theta(x)$ is just a linear model over the features $\phi_\beta(x)$. However, we will train the neural networks, both the parameters in β and the parameters $W^{[r]}, b^{[r]}$ are optimized, and therefore we are not learning a linear model in the feature space, but also learning a good feature map $\phi_\beta(\cdot)$ itself so that it's possible to predict accurately with a linear model on top of the feature map. Therefore, deep learning tends to depend less on the domain knowledge of the particular applications and requires often less feature engineering. The penultimate layer $a^{[r-1]}$ is often (informally) referred to as the learned features or representations in the context of deep learning.

In the example of house price prediction, a fully-connected neural network does not need us to specify the intermediate quantity such ''family size'', and may automatically discover some useful features in the last penultimate layer (the activation $a^{[r-1]}$), and use them to linearly predict the housing price. Often the feature map / representation obtained from one datasets (that is, the function $\phi_\beta(\cdot)$ can be also useful for other datasets, which indicates they contain essential information about the data. However, oftentimes, the neural network will discover complex features which are very useful for predicting the output but may be difficult for a human to understand or interpret. This is why some people refer to neural networks as a *black box*, as it can be difficult to understand the features it has discovered.

10 *Backpropagation*

In this section, we introduce *backpropgation* or *auto-differentiation*, which computes the gradient of the loss $\nabla J^{(j)}(\theta)$ efficiently. We will start with an informal theorem that states that as long as a real-valued function f can be efficiently computed/evaluated by a differentiable network or circuit, then its gradient can be efficiently computed in a similar time. We will then show how to do this concretely for fully-connected neural networks.

Because the formality of the general theorem is not the main focus here, we will introduce the terms with informal definitions. By a differentiable circuit or a differentiable network, we mean a composition of a sequence of differentiable arithmetic operations (additions, subtraction, multiplication, divisions, etc) and elementary differentiable functions (ReLU, exp, log, sin, cos, etc.). Let the size of the circuit be the total number of such operations and elementary functions. We assume that each of the operations and functions, and their derivatives or partial derivatives can be computed in $O(1)$ time in the computer.

Theorem 1 (backpropagation or auto-differentiation, informally stated)**.** *Suppose a differentiable circuit of size N computes a real-valued function $f : R^{\ell} \mapsto R$. Then, the gradient ∇f can be computed in time $O(N)$, by a circuit of size $O(N)$.*[1]

We note that the loss function $J^{(j)}(\theta)$ for the j-th example can be indeed computed by a sequence of operations and functions involving additions, subtraction, multiplications, and non-linear activations. Thus the theorem suggests that we should be able to compute $\nabla J^{(j)}(\theta)$ in a similar time to that for computing $J^{(j)}(\theta)$ itself. This does not only apply to the fully-connected neural network introduced in chapter 9, but also many other types of neural networks.

In the rest of the section, we will showcase how to compute the gradient of the loss efficiently for fully-connected neural networks using backpropagation. Even though auto-differentiation or backpropagation is implemented in all the deep learning packages such as TensorFlow and PyTorch, understanding it is very helpful for gaining insights into the workings of deep learning.

[1] We note if the output of the function f does not depend on some of the input coordinates, then we set by default the gradient w.r.t that coordinate to zero. Setting to zero does not count towards the total runtime here in our accounting scheme. This is why when $N \leq \ell$, we can compute the gradient in $O(N)$ time, which might be potentially even less than ℓ.

10.1 Preliminary: chain rule

We first recall the chain rule in calculus. Suppose the variable J depends on the variables $\theta_1, \ldots, \theta_p$ via the intermediate variables $g_1, \ldots, g_k$:

$$g_j = g_j(\theta_1, \ldots, \theta_p), \forall j \in \{1, \ldots, k\} J \qquad = J(g_1, \ldots, g_k) \tag{10.1}$$

Here we overload the meaning of g_j's: they denote both the intermediate variables but also the functions used to compute the intermediate variables. Then, by the chain rule, we have that $\forall i$:

$$\frac{\partial J}{\partial \theta_i} = \sum_{j=1}^{k} \frac{\partial J}{\partial gj} \frac{\partial gj}{\partial \theta_i} \tag{10.2}$$

For the ease of invoking the chain rule in the following subsections in various ways, we will call J the output variable, $g_1, \ldots, g_k$ intermediate variables, and $\theta_1, \ldots, \theta_p$ the input variables in the chain rule.

10.2 Backpropagation for two-layer neural networks

Now we consider the two-layer neural network defined in equation (9.11). Our general approach is to first unpack the vectorized notation to scalar form to apply the chain rule, but as soon as we finish the derivation, we will pack the scalar equations back to a vectorized form to keep the notations succinct.

Recall the following equations are used for the computation of the loss J:

$$z = W^{[1]} x + b^{[1]} \tag{10.3}$$

$$a = \text{ReLU}(z) \tag{10.4}$$

$$h_\theta(x) \triangleq o = W^{[2]} a + b^{[2]} \tag{10.5}$$

$$J = \frac{1}{2}(y - o)^2 \tag{10.6}$$

Recall that $W^{[1]} \in \mathbb{R}^{m \times d}$, $W^{[2]} \in \mathbb{R}^{1 \times m}$, and $b^{[1]}, z, a \in \mathbb{R}^m$, and $o, y, b^{[2]} \in \mathbb{R}$. Recall that a vector in $\mathbb{R}^d$ is automatically interpreted as a column vector (like a matrix in $\mathbb{R}^{d \times 1}$) if need be.[2]

[2] We also note that even though this is the convention in math, it's different from the convention in numpy where an one dimensional array will be automatically interpreted as a row vector.

10.2.1 *Computing* $\frac{\partial J}{\partial W^{[2]}}$

Suppose $W^{[2]} = [W_1^{[2]}, \ldots, W_m^{[2]}]$. We start by computing $\frac{\partial J}{\partial W_i^{[2]}}$ using the chain rule (equation (10.2)) with o as the intermediate variable.

$$\begin{aligned}\frac{\partial J}{\partial W[2]_i} &= \frac{\partial J}{\partial o} \cdot \frac{\partial o}{\partial W[2]_i} \\ &= (o - y) \cdot \frac{\partial o}{\partial W[2]_i} \\ &= (o - y) \cdot a_i \qquad \text{(because } o = \textstyle\sum_{i=1}^{m} W[2]_i a_i + b^{[2]}\text{)}\end{aligned}$$

Vectorized notation. The equation above in vectorized notation becomes:

$$\frac{\partial J}{\partial W^{[2]}} = (o - y) \cdot a^\top \in \mathbb{R}^{1 \times m} \tag{10.7}$$

Similarly, we leave the reader to verify that:

$$\frac{\partial J}{\partial b^{[2]}} = (o - y) \in \mathbb{R} \tag{10.8}$$

Clarification for the dimensionality of the partial derivative notation. We will use the notation $\frac{\partial J}{\partial A}$ frequently in the rest of the lecture notes. We note that here we only use this notation for the case when J is a **real-valued** variable,[3] but A can be a vector or a matrix. Moreover, $\frac{\partial J}{\partial A}$ has the same dimensionality as A. For example, when A is a matrix, the (i, j)-th entry of $\frac{\partial J}{\partial A}$ is equal to $\frac{\partial J}{\partial A_{ij}}$. If you are familiar with the notion of total derivatives, we note that the convention for dimensionality here is different from that for total derivatives.

[3] There is an extension of this notation to vector or matrix variable J. However, in practice, it's often impractical to compute the derivatives of high-dimensional outputs. Thus, we will avoid using the notation $\frac{\partial J}{\partial A}$ for J that is not a real-valued variable.

10.2.2 *Computing* $\frac{\partial J}{\partial W^{[1]}}$

Next we compute $\frac{\partial J}{\partial W^{[1]}}$. We first unpack the vectorized notation: let $W_{ij}^{[1]}$ denote the (i, j)-the entry of $W^{[1]}$, where $i \in [m]$ and $j \in [d]$. We compute $\frac{\partial J}{\partial W_{ij}^{[1]}}$ using chain rule (equation (10.2)) with z_i as the intermediate variable:

$$\begin{aligned}\frac{\partial J}{\partial W_{ij}^{[1]}} &= \frac{\partial J}{\partial z_i} \cdot \frac{\partial z_i}{\partial W_{ij}^{[1]}} \\ &= \frac{\partial J}{\partial z_i} \cdot x_j \qquad \text{(because } z_i = \textstyle\sum_{k=1}^{d} W_{ik}^{[1]} x_k + b_i^{[1]}\text{)}\end{aligned}$$

Vectorized notation. The equation above can be written compactly as:

$$\frac{\partial J}{\partial W^{[1]}} = \frac{\partial J}{\partial z} \cdot x^\top \tag{10.9}$$

We can verify that the dimensions match: $\frac{\partial J}{\partial W^{[1]}} \in \mathbb{R}^{m \times d}$, $\frac{\partial J}{\partial z} \in \mathbb{R}^{m \times 1}$ and $x^\top \in \mathbb{R}^{1 \times d}$.

Abstraction. For future usage, the computations for $\frac{\partial J}{\partial W^{[1]}}$ and $\frac{\partial J}{\partial W^{[2]}}$ above can be abstractified into the following claim:

Claim 1. Suppose J is a real-valued output variable, $z \in \mathbb{R}^m$ is the intermediate variable, and $W \in \mathbb{R}^{m \times d}, u \in \mathbb{R}^d, b \in \mathbb{R}^m$ are the input variables, and suppose they satisfy the following:

$$\begin{aligned} z &= Wu + b \\ J &= J(z) \end{aligned}$$

Then $\frac{\partial J}{\partial W}$ and $\frac{\partial J}{\partial b}$ satisfy:

$$\begin{aligned} \frac{\partial J}{\partial W} &= \frac{\partial J}{\partial z} \cdot u^\top \\ \frac{\partial J}{\partial b} &= \frac{\partial J}{\partial z} \end{aligned}$$

10.2.3 *Computing* $\frac{\partial J}{\partial z}$

Equation (10.9) tells us that to compute $\frac{\partial J}{\partial W^{[1]}}$, it suffices to compute $\frac{\partial J}{\partial z}$, which is the goal of the next few derivations.

We invoke the chain rule with J as the output variable, a_i as the intermediate variable, and z_i as the input variable:

$$\begin{aligned} \frac{\partial J}{\partial z_i} &= \frac{\partial J}{\partial a_i} \frac{\partial a_i}{\partial z_i} \\ &= \frac{\partial J}{\partial a_i} \cdot \mathbb{1}\{z_i \geq 0\} \end{aligned}$$

Vectorization and abstraction. The computation above can be summarized into:

Claim 2. Suppose the real-valued output variable J and vectors $z, a \in \mathbb{R}^m$ satisfy the following:

$$\begin{aligned} a &= \sigma(z) && \text{(where } \sigma \text{ is an element-wise activation, } z, a \in \mathbb{R}^m\text{)} \\ J &= J(a) \end{aligned}$$

Then, we have that

$$\frac{\partial J}{\partial z} = \frac{\partial J}{\partial a} \odot \sigma'(z)$$

where $\sigma'(\cdot)$ is the element-wise derivative of the activation function σ, and $\odot$ denotes the element-wise product of two vectors of the same dimensionality.

10.2.4 *Computing* $\frac{\partial J}{\partial a}$

Now it suffices to compute $\frac{\partial J}{\partial a}$. We invoke the chain rule with J as the output variable, o as the intermediate variable, and a_i as the input variable:

$$\begin{aligned} \frac{\partial J}{\partial a_i} &= \frac{\partial J}{\partial o} \frac{\partial o}{\partial a_i} \\ &= (o - y) \cdot W_i^{[2]} && \text{(because } o = \textstyle\sum_{i=1}^m W_i^{[2]} a_i + b^{[2]}\text{)} \end{aligned}$$

Vectorization. In vectorized notation, we have:

$$\frac{\partial J}{\partial a} = {W^{[2]}}^\top \cdot (o - y) \tag{10.10}$$

Abstraction. We now present a more general form of the computation above.

Claim 3. Suppose J is a real-valued output variable, $v \in \mathbb{R}^m$ is the intermediate variable, and $W \in \mathbb{R}^{m \times d}, u \in \mathbb{R}^d, b \in \mathbb{R}^m$ are the input variables, and suppose they satisfy the following:

$$\begin{aligned} v &= Wu + b \\ J &= J(v) \end{aligned}$$

Then,

$$\frac{\partial J}{\partial u} = W^\top \frac{\partial J}{\partial v}. \tag{10.11}$$

10.2.5 Summary for two-layer neural networks

Now combining the equations above, we arrive at algorithm 10.1 which computes the gradients for two-layer neural networks.

Compute the values of $z \in \mathbb{R}^m$, $a \in \mathbb{R}^m$, and $o \in \mathbb{R}$
Compute:

$$\delta^{[2]} \triangleq \frac{\partial J}{\partial o} = (o - y) \in \mathbb{R}$$
$$\delta^{[1]} \triangleq \frac{\partial J}{\partial z} = (W^{[2]^\top}(o - y)) \odot \mathbb{1}\{z \geq 0\} \in \mathbb{R}^{m \times 1} \quad \text{(by claim 2 and 10.10)}$$

Compute:

$$\frac{\partial J}{\partial W^{[2]}} = \delta^{[2]} a^\top \in \mathbb{R}^{1 \times m} \quad \text{(by equation (10.7))}$$
$$\frac{\partial J}{\partial b^{[2]}} = \delta^{[2]} \in \mathbb{R} \quad \text{(by equation (10.8))}$$
$$\frac{\partial J}{\partial W^{[1]}} = \delta^{[1]} x^\top \in \mathbb{R}^{m \times d} \quad \text{(by equation (10.9))}$$
$$\frac{\partial J}{\partial b^{[1]}} = \delta^{[1]} \in \mathbb{R}^m \quad \text{(as an exercise)}$$

Algorithm 10.1. Back-propogation for two-layer neural networks.

10.3 Multi-layer neural networks

In this section, we will derive the backpropagation algorithms for the model defined in equation (9.16). With the notation $a^{[0]} = x$, recall that we have:

$$\begin{aligned}
a^{[1]} &= \text{ReLU}(W^{[1]} a^{[0]} + b^{[1]}) \\
a^{[2]} &= \text{ReLU}(W^{[2]} a^{[1]} + b^{[2]}) \\
&\cdots \\
a^{[r-1]} &= \text{ReLU}(W^{[r-1]} a^{[r-2]} + b^{[r-1]}) \\
a^{[r]} &= z^{[r]} = W^{[r]} a^{[r-1]} + b^{[r]} \\
J &= \frac{1}{2}\left(a^{[r]} - y\right)^2
\end{aligned}$$

Here we define both $a^{[r]}$ and $z^{[r]}$ as $h_\theta(x)$ for notational simplicity.

First, we note that we have the following local abstraction for $k \in \{1, \ldots, r\}$:

$$z^{[k]} = W^{[k]} a^{[k-1]} + b^{[k]}$$
$$J = J(z^{[k]})$$

Invoking Claim 1, we have that

$$\frac{\partial J}{\partial W^{[k]}} = \frac{\partial J}{\partial z^{[k]}} \cdot a^{[k-1]^\top} \tag{10.12}$$
$$\frac{\partial J}{\partial b^{[k]}} = \frac{\partial J}{\partial z^{[k]}} \tag{10.13}$$

Therefore, it suffices to compute $\frac{\partial J}{\partial z^{[k]}}$. For simplicity, let's define $\delta^{[k]} \triangleq \frac{\partial J}{\partial z^{[k]}}$. We compute $\delta^{[k]}$ from $k = r$ to 1 inductively. First we have that:

$$\delta^{[r]} \triangleq \frac{\partial J}{\partial z^{[r]}} = (z^{[r]} - y) \tag{10.14}$$

Next for $k \leq r - 1$, suppose we have computed the value of $\delta^{[k+1]}$, then we will compute $\delta^{[k]}$. First, using claim 2, we have that:

$$\delta^{[k]} \triangleq \frac{\partial J}{\partial z^{[k]}} = \frac{\partial J}{\partial a^{[k]}} \odot \mathrm{ReLU}'(z^{[k]}) \tag{10.15}$$

Then we note that the relationship between $a^{[k]}$ and $z^{[k+1]}$ can be abstractly written as:

$$z^{[k+1]} = W^{[k+1]} a^{[k]} + b^{[k+1]} \tag{10.16}$$
$$J = J(z^{[k+1]}) \tag{10.17}$$

Therefore by claim 3 we have that:

$$\frac{\partial J}{\partial a^{[k]}} = W^{[k+1]^\top} \frac{\partial J}{\partial z^{[k+1]}} \tag{10.18}$$

It follows that:

$$\begin{aligned}\delta^{[k]} &= \left(W^{[k+1]^\top} \frac{\partial J}{\partial z^{[k+1]}} \right) \odot \mathrm{ReLU}'(z^{[k]}) \\ &= \left(W^{[k+1]^\top} \delta^{[k+1]} \right) \odot \mathrm{ReLU}'(z^{[k]})\end{aligned}$$

Compute and store the values of $a^{[k]}$'s and $z^{[k]}$'s for $k = 1, \ldots, r$, and J.
▷ (This is often called the ''forward pass''.)
for $k = r$ to 1 **do** ▷ (This is often called the ''backward pass'')
  if $k = r$ **then**
    Compute $\delta^{[r]} \triangleq \frac{\partial J}{\partial z^{[r]}}$
  else
    Compute:

$$\delta^{[k]} \triangleq \frac{\partial J}{\partial z^{[k]}} = \left(W^{[k+1]^\top} \delta^{[k+1]}\right) \odot \mathrm{ReLU}'(z^{[k]})$$

  Compute:

$$\frac{\partial J}{\partial W^{[k]}} = \delta^{[k]} a^{[k-1]^\top}$$
$$\frac{\partial J}{\partial b^{[k]}} = \delta^{[k]}$$

end for

Algorithm 10.2. Back-propagation for multi-layer neural networks.

11 *Vectorization Over Training Examples*

As we discussed in chapter 9, in the implementation of neural networks, we will leverage the parallelism across multiple examples. This means that we will need to write the forward pass (the evaluation of the outputs) of the neural network and the backward pass (backpropagation) for multiple training examples in matrix notation.

The basic idea. The basic idea is simple. Suppose you have a training set with three examples $x^{(1)}, x^{(2)}, x^{(3)}$. The first-layer activations for each example are as follows:

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

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

$$X = \begin{bmatrix} | & | & | \\ x^{(1)} & x^{(2)} & x^{(3)} \\ | & | & | \end{bmatrix} \in \mathbb{R}^{d \times 3} \tag{11.1}$$

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

$$Z^{[1]} = \begin{bmatrix} | & | & | \\ z^{[1](1)} & z^{[1](2)} & z^{[1](3)} \\ | & | & | \end{bmatrix} = W^{[1]} X + b^{[1]} \tag{11.2}$$

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

$$\tilde{b}^{[1]} = \begin{bmatrix} | & | & | \\ b^{[1]} & b^{[1]} & b^{[1]} \\ | & | & | \end{bmatrix} \tag{11.3}$$

We can then perform the computation: $Z^{[1]} = W^{[1]}X + \tilde{b}^{[1]}$. Often times, it is not necessary to explicitly construct $\tilde{b}^{[1]}$. By inspecting the dimensions in equation (11.1), you can assume $b^{[1]} \in \mathbb{R}^{4\times 1}$ is correctly broadcast to $W^{[1]}X \in \mathbb{R}^{4\times 3}$.

The matricization approach as above can easily generalize to multiple layers, with one subtlety though, as discussed below.

Complications/subtlety in the implementation. All the deep learning packages or implementations put the data points in the rows of a data matrix. (If the data point itself is a matrix or tensor, then the data are concentrated along the zero-th dimension.) However, most of the deep learning papers use a similar notation to these notes where the data points are treated as column vectors.[1] There is a simple conversion to deal with the mismatch: in the implementation, all the columns become row vectors, row vectors become column vectors, all the matrices are transposed, and the orders of the matrix multiplications are flipped. In the example above, using the row major convention, the data matrix is $X \in \mathbb{R}^{3\times d}$, the first layer weight matrix has dimensionality $d \times m$ (instead of $m \times d$ as in the two layer neural net section), and the bias vector $b^{[1]} \in \mathbb{R}^{1\times m}$. The computation for the hidden activation becomes:

$$Z^{[1]} = XW^{[1]} + b^{[1]} \in \mathbb{R}^{3\times m} \tag{11.4}$$

[1] The instructor suspects that this is mostly because in mathematics we naturally multiply a matrix to a vector on the left hand side.

Part VI: Regularization and Model Selection

From CS229 Spring 2021, Andrew Ng, Moses Charikar, Christopher Ré & Yoann Le Calonnec, Stanford University.

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

For the sake of concreteness, in these notes we assume we have some finite set of models $\mathcal{M} = \{M_1, \ldots, M_d\}$ that we're trying to select among. For instance, in our first example above, the model M_i would be an i-th order polynomial regression model. (The generalization to infinite $\mathcal{M}$ is not hard.[3]) Alternatively, if we are trying to decide between using an SVM, a neural network or logistic regression, then $\mathcal{M}$ may contain these models.

[2] 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.

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

12 Cross validation

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

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

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

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

1. Randomly split S into S_{train} (say, 70% of the data) and S_{cv} (the remaining 30%). Here, S_{cv} is called the hold-out cross validation set.

2. Train each model M_i on S_{train} only, to get some hypothesis h_i .

3. Select and output the hypothesis h_i that had the smallest error $\hat{\varepsilon}_{S_{\text{cv}}}(h_i)$ on the hold out cross validation set. (Recall, $\hat{\varepsilon}_{S_{\text{cv}}}(h)$ denotes the empirical error of h on the set of examples in S_{cv}.)

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

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

The disadvantage of using hold out cross validation is that it ''wastes'' about 30% of the data. Even if we were to take the optional step of retraining the model on the entire training set, it's still as if we're trying to find a good model for a learning problem in which we had $0.7m$ training examples, rather than 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 $S_1, \ldots, S_k$.

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

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

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

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

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

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

Finally, even though we have described the different versions of cross validation as methods for selecting a model, they can also be used more simply to evaluate a *single* model or algorithm. For example, if you have implemented some learning algorithm and want to estimate how well it performs for your application (or if you have invented a novel learning algorithm and want to report in a technical paper how well it performs on various test sets), cross validation would give a reasonable way of doing so.

13 *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 $d \gg n$), 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**:

Algorithm 13.1. Forward search.

Initialize $\mathcal{F} = \emptyset$.
repeat
  for $i = 1, \ldots, d$ **do**
    if $i \notin \mathcal{F}$ **then**
      $\mathcal{F}_i = \mathcal{F} \cup \{i\}$
      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.)
  end for
  Set $\mathcal{F}$ to be the best feature subset found in the previous step.
until convergence
Select and output the best feature subset that was evaluated during the entire search procedure.

The outer loop of the algorithm can be terminated either when $\mathcal{F} = \{1, \ldots, d\}$ 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 $\mathcal{F} =$

$\{1, \ldots, d\}$ 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 $\mathcal{F} = \{1, \ldots, d\}$) 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 $\mathrm{MI}(x_i, y)$ between x_i and y:

$$\mathrm{MI}(x_i, y) = \sum_{x_i \in \{0,1\}} \sum_{y \in \{0,1\}} p(x_i, y) \log \frac{p(x_i, y)}{p(x_i)p(y)} \tag{13.1}$$

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

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

$$\mathrm{MI}(x_i, y) = \mathrm{KL}(p(x_i, y) \mid\mid p(x_i)p(y)) \tag{13.2}$$

You'll get to play more with KL-divergence in the problem sets, but informally, this gives a measure of how different the probability distributions $p(x_i, y)$ and $p(x_i)p(y)$ are. If x_i and y are independent random variables, then we would have $p(x_i, y) = p(x_i)p(y)$, and the KL-divergence between the two distributions will be zero. This is consistent with the idea if x_i and y are independent, then x_i is clearly very ''non-informative'' about y, and thus the score $S(i)$ should be small. Conversely, if x_i is very ''informative'' about y, then their mutual information $\mathrm{MI}(x_i, y)$ would be large.

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

14 *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

$$\theta_{\text{MLE}} = \arg\max_{\theta} \prod_{i=1}^{n} p(y^{(i)} \mid x^{(i)}; \theta). \tag{14.1}$$

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

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

$$p(\theta \mid S) = \frac{p(S \mid \theta)p(\theta)}{p(S)} \tag{14.2}$$

$$= \frac{\left(\prod_{i=1}^{n} p(y^{(i)} \mid x^{(i)}, \theta)\right) p(\theta)}{\int_{\theta} \left(\prod_{i=1}^{n} p(y^{(i)} \mid x^{(i)}, \theta) p(\theta)\right) d\theta} \tag{14.3}$$

In the equation above, $p(y^{(i)} \mid x^{(i)}, \theta)$ comes from whatever model you're using for your learning problem. For example, if you are using Bayesian logistic regression, then you might choose $p(y^{(i)} \mid x^{(i)}, \theta) = h_\theta(x^{(i)})^{y^{(i)}}(1 - h_\theta(x^{(i)}))^{(1-y^{(i)})}$, where $h_\theta(x^{(i)}) = 1/(1 + \exp(-\theta^\top x^{(i)}))$.[1]

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

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

$$p(y \mid x, S) = \int_\theta p(y \mid x, \theta) p(\theta \mid S) d\theta \tag{14.4}$$

In the equation above, $p(\theta \mid S)$ comes from equation (14.2). Thus, for example, if the goal is to the predict the expected value of y given x, then we would output:[2]

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

$$\mathbb{E}[y \mid x, S] = \int_y y p(y \mid x, S) dy \tag{14.5}$$

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 $p(\theta \mid S)$ over θ. Unfortunately, in general it is computationally very difficult to compute this posterior distribution. This is because it requires taking integrals over the (usually high-dimensional) θ as in equation (14.2), and this typically cannot be done in closed-form.

Thus, in practice we will instead approximate the posterior distribution for θ. One common approximation is to replace our posterior distribution for θ (as in equation (14.4)) with a single point estimate. The **MAP** (**maximum a posteriori**) estimate for θ is given by:

$$\theta_{\text{MAP}} = \arg\max_\theta \prod_{i=1}^{n} p(y^{(i)} \mid x^{(i)}, \theta) p(\theta) \tag{14.6}$$

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

In practical applications, a common choice for the prior $p(\theta)$ is to assume that $\theta \sim \mathcal{N}(0, \tau^2 I)$. Using this choice of prior, the fitted parameters θ MAP will have smaller norm than that selected by maximum likelihood. 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 $d \gg n$.

15 Some calculations from bias variance

From CS229 Fall 2020, Christopher Ré, Stanford University.

This section contains a reprise of the eigenvalue arguments to understand how variance is reduced by regularization. We also describe different ways regularization can occur including from the algorithm or initialization. This note contains some additional calculations from the lecture and Piazza, just so that we have typeset versions of them. They contain no new information over the lecture, but they do supplement the previous sections.

Recall we have a design matrix $X \in \mathbb{R}^{n \times d}$ and labels $y \in \mathbb{R}^n$. We are interested in the underdetermined case $n < d$ so that $\text{rank}(X) \leq n < d$. We consider the following optimization problem for least squares with a regularization parameter $\lambda \geq 0$:

$$\ell(\theta; \lambda) = \min_{\theta \in \mathbb{R}^d} \frac{1}{2}\|X\theta - y\|^2 + \frac{\lambda}{2}\|\theta\|^2 \tag{15.1}$$

Normal equations. Computing derivatives as we did for the normal equations, we see that:

$$\nabla_\theta \ell(\theta; \lambda) = X^\top (X\theta - y) + \lambda\theta = (X^\top X + \lambda I)\theta - X^\top y \tag{15.2}$$

By setting $\nabla_\theta \ell(\theta, \lambda) = 0$ we can solve for the $\hat{\theta}$ that minimizes the above problem. Explicitly, we have:

$$\hat{\theta} = (X^\top X + \lambda I)^{-1} X^\top y \tag{15.3}$$

To see that the inverse in equation (15.3) exists, we observe that $X^\top X$ is a symmetric, real $d \times d$ matrix so it has d eigenvalues (some may be 0). Moreover, it is positive semidefinite, and we capture this by writing $\text{eig}(X^\top X) = \{\sigma_1^2, \ldots, \sigma_d^2\}$. Now, inspired by the regularized problem, we examine:

$$\text{eig}(X^\top X + \lambda I) = \left\{\sigma_1^2 + \lambda, \ldots, \sigma_d^2 + \lambda\right\} \tag{15.4}$$

Since $\sigma_i^2 \geq 0$ for all $i \in [d]$, if we set $\lambda > 0$ then $X^\top X + \lambda I$ is full rank, and the inverse of $(X^\top X + \lambda I)$ exists. In turn, this means there is a unique such $\hat{\theta}$.

Variance. Recall that in bias-variance, we are concerned with the variance of $\hat{\theta}$ as we sample the training set. We want to argue that as the regularization parameter λ increases, the variance in the fitted $\hat{\theta}$ decreases. We won't carry out the full formal argument, but it suffices to make one observation that is immediate from equation (15.3): *the variance of $\hat{\theta}$ is proportional to the eigenvalues of* $(X^\top X + \lambda I)^{-1}$. To see this, observe that the eigenvalues of an inverse are just the inverse of the eigenvalues:

$$\text{eig}\left((X^\top X + \lambda I)^{-1}\right) = \left\{\frac{1}{\sigma_1^2 + \lambda}, \ldots, \frac{1}{\sigma_d^2} + \lambda\right\} \tag{15.5}$$

Now, condition on the points we draw, namely X. Then, recall that randomness is in the label noise (recall the linear regression model $y \sim X\theta^* + \mathcal{N}(0, \tau^2 I) = \mathcal{N}(X\theta^*, \tau^2 I)$).

Recall a fact about the multivariate normal distribution:

$$\text{if } y \sim N(\mu, \Sigma) \text{ then } Ay \sim \mathcal{N}(A\mu, A\Sigma A^\top) \tag{15.6}$$

Using linearity, we can verify that the expectation of $\hat{\theta}$ is:

$$\begin{aligned}
\mathbb{E}[\hat{\theta}] &= \mathbb{E}\left[(X^\top X + \lambda I)^{-1} X^\top y\right] && (15.7)\\
&= \mathbb{E}\left[(X^\top X + \lambda I)^{-1} X^\top (X\theta^* + \mathcal{N}(0, \tau^2 I))\right] && (15.8)\\
&= \mathbb{E}\left[(X^\top X + \lambda I)^{-1} X^\top (X\theta^*)\right] && (15.9)\\
&= (X^\top X + \lambda I)^{-1}(X^\top X)\theta^* && \text{(essentially a ''shrunk'' } \theta^*)
\end{aligned}$$

The last line above suggests that the more regularization we add (larger the λ), the more the estimated $\hat{\theta}$ will be shrunk towards 0. In other words, regularization adds bias (towards zero in this case). Though we paid the cost of higher bias, we gain by reducing the variance of $\hat{\theta}$. To see this bias-variance tradeoff concretely, observe the covariance matrix of $\hat{\theta}$:

$$\begin{aligned}
C &:= \text{Cov}[\hat{\theta}] && (15.10)\\
&= \left((X^\top X + \lambda I)^{-1} X^\top\right)(\tau^2 I)\left(X(X^\top X + \lambda I)^{-1}\right) && (15.11)
\end{aligned}$$

and

$$\text{eig}(C) = \left\{\frac{\tau^2\sigma_1^2}{(\sigma_1^2 + \lambda)^2}, \ldots, \frac{\tau^2\sigma_d^2}{(\sigma_d^2 + \lambda)^2}\right\} \tag{15.12}$$

Notice that the entire spectrum of the covariance is a decreasing function of λ. By decomposing in the eigenvalue basis, we can see that actually $\mathbb{E}\left[\|\hat{\theta} - \theta^*\|^2\right]$ is a decreasing function of λ, as desired.

Gradient descent. We show that you can initialize gradient descent in a way that effectively regularizes undetermined least squares—even with no regularization penalty ($\lambda = 0$). Our first observation is that any point $x \in \mathbb{R}^d$ can be decomposed into two orthogonal components x_0, x_1 such that:

$$x = x_0 + x_1 \text{ and } x_0 \in \text{Null}(X) \text{ and } x_1 \in \text{Range}(X^\top) \tag{15.13}$$

Recall that $\text{Null}(X)$ and $\text{Range}(X^\top)$ are orthogonal subspaces by the fundamental theory of linear algebra. We write P_0 for the projection on the null and P_1 for the projection on the range, then $x_0 = P_0(x)$ and $x_1 = P_1(x)$.

If one initializes at a point θ then, we observe that the gradient is orthogonal to the null space. That is, if $g(\theta) = X^\top(X\theta - y)$ then $g^\top P_0(v) = 0$ for any $v \in \mathbb{R}^d$. But, then:

$$P_0(\theta^{(t+1)}) = P_0(\theta^t - \alpha g(\theta^{(t)})) = P_0(\theta^t) - \alpha P_0 g(\theta^{(t)}) = P_0(\theta^{(t)}) \tag{15.14}$$

That is, no learning happens in the null. Whatever portion is in the null that we initialize stays there throughout execution.

A key property of the Moore-Penrose pseudoinverse, is that if $\hat{\theta} = (X^\top X) + X^\top y$ then $P_0(\hat{\theta}) = 0$. Hence, the gradient descent solution initialized at θ_0 can be written $\hat{\theta} + P_0(\theta_0)$. Two immediate observations:

- Using the Moore-Penrose inverse acts as regularization, because it selects the solution $\hat{\theta}$.
- So does gradient descent—provided that we initialize at $\theta_0 = 0$. This is particularly interesting, as many modern machine learning techniques operate in these underdetermined regimes.

We've argued that there are many ways to find equivalent solutions, and that this allows us to understand the effect on the model fitting procedure as regularization. Thus, there are many ways to find that equivalent solution. Many modern methods of machine learning including dropout and data augmentation are not penalty, but their effect is understood as regularization. One contrast with the above methods is that they often depend on some property of the data or for

how much they effectively regularization. In some sense, they adapt to the data. A final comment is that in the same sense above, adding more data regularizes the model as well!

16 Bias-variance and error analysis

From CS229 Fall 2017, Yoann Le Calonnec, Stanford University.

16.1 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):

$$\mathbb{E}_{(x,y)\sim\text{test set}}\left[|\hat{f}(x) - y|^2\right] \tag{16.1}$$

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

$$y_i = f(x_i) + \epsilon_i, \quad \text{where the noise } \epsilon_i \text{ satisfies} \quad \mathbb{E}(\epsilon_i) = 0, \text{Var}(\epsilon_i) = \sigma^2 \tag{16.2}$$

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 ϵ_j is a real random variable with mean 0 and variance σ^2. The crucial observation is that $\hat{f}(x_j)$ is random since it depends on the values ϵ_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 traning set):

$$\begin{aligned}
\text{test MSE} &= \mathbb{E}\left[(y - \hat{f}(x))^2\right] && (16.3)\\
&= \mathbb{E}\left[((\epsilon + f(x) - \hat{f}(x))^2\right] && (16.4)\\
&= \mathbb{E}[\epsilon^2] + \mathbb{E}\left[(f(x) - \hat{f}(x))^2\right] && (16.5)\\
&= \sigma^2 + \left(\mathbb{E}[f(x) - \hat{f}(x)]\right)^2 + \text{Var}\left(f(x) - \hat{f}(x)\right) && (16.6)\\
&= \sigma^2 + \left(\text{Bias } \hat{f}(x)\right)^2 + \text{Var}\left(\hat{f}(x)\right) && (16.7)
\end{aligned}$$

There is nothing we can do about the first term σ^2 as we can not predict the noise ϵ 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 train and varies a lot with the choice of our training set.

To sum up, we can understand our MSE as follows:

$$\begin{aligned}
\text{High Bias} &\longleftrightarrow \text{Underfitting}\\
\text{High Variance} &\longleftrightarrow \text{Overfitting}\\
\text{Large } \sigma^2 &\longleftrightarrow \text{Noisy data}
\end{aligned}$$

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.

16.2 *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 **??**. 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
5. Final logistic regression step to predict the label

If you biuld 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%.

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 16.1. Accuracy when providing the system with the perfect component.

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 16.1.

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 analyze the error of an algorithm and explain which parts should be improved.

16.3 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 16.2.

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 16.2. Accuracy when removing feature from logistic regression.

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.

16.3.1 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.

Part VII: Unsupervised Learning

From CS229 Spring 2021, Andrew Ng, Moses Charikar, Christopher Ré & Tengyu Ma, Stanford University.

17 The k-means Clustering Algorithm

In the clustering problem, we are given a training set $\{x^{(1)}, \ldots, x^{(n)}\}$, 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** $\mu_1, \mu_2, \ldots, \mu_k \in \mathbb{R}^d$ randomly.
2. Repeat until convergence:
   - For every i, set:
   $$c^{(i)} := \arg\min_j \|x^{(i)} - \mu - j\|^2$$
   - For each j, set:
   $$\mu_j := \frac{\sum_{i=1}^n \mathbb{1}\{c^{(i)} = j\} x^{(i)}}{\sum_{i=1}^n \mathbb{1}\{c^{(i)} = j\}}$$

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

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

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 μ fixed, and then minimizes J with respect to μ while holding c fixed. Thus, J must monotonically decrease, and the value of J must converge. (Usually, this implies that c and μ will converge too. In theory, it is possible for k-means to oscillate between a few different clusterings—i.e., a few different values for c and/or μ—that have exactly the same value of J, but this almost never happens in practice.)

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

18 Mixtures of Gaussians and the EM Algorithm

In this chapter, we discuss the EM (Expectation-Maximization) algorithm for density estimation.

Suppose that we are given a training set $\{x^{(1)}, \ldots, x^{(n)}\}$ 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 $p(x^{(i)}, z^{(i)}) = p(x^{(i)} \mid z^{(i)})p(z^{(i)})$. Here, $z^{(i)} \sim \text{Multinomial}(\phi)$ (where $\phi_j \geq 0, \sum_{j=1}^{k} \phi_j = 1$, and the parameter ϕ_j gives $p(z^{(i)} = j)$), and $x^{(i)} \mid z^{(i)} = j \sim \mathcal{N}(\mu_j, \Sigma_j)$. We let k denote the number of values that the $z^{(i)}$'s can take on. Thus, our model posits that each

$x^{(i)}$ was generated by randomly choosing $z^{(i)}$ from $\{1, \ldots, k\}$, and then $x^{(i)}$ was drawn from one of k Gaussians depending on $z^{(i)}$. This is called the **mixture of Gaussians** model. Also, note that the $z^{(i)}$'s are **latent** random variables, meaning that they're hidden/unobserved. This is what will make our estimation problem difficult.

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

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

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:

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

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

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

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

However, in our density estimation problem, the $z^{(i)}$'s are not known. What can we do? The EM algorithm is an iterative algorithm that has two main steps.

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

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:

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

  - (M-step) Update the parameters:

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

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:

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

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

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

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

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 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.

Part VIII: The EM Algorithm

From CS229 Spring 2021, Andrew Ng, Moses Charikar, Christopher Ré & Tengyu Ma, Stanford University.

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**.

19 *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:

$$\mathbb{E}[f(X)] \geq f(\mathbb{E}[X]). \tag{19.1}$$

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

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

20 *The EM algorithm*

Suppose we have an estimation problem in which we have a training set $\{x^{(1)}, \ldots, x^{(n)}\}$ 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:

$$p(x; \theta) = \sum_z p(x, z; \theta) \tag{20.1}$$

We wish to fit the parameters θ by maximizing the log-likelihood of the data, defined by:

$$\ell(\theta) = \sum_{i=1}^{n} \log p(x^{(i)}; \theta) \tag{20.2}$$

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

$$\ell(\theta) = \sum_{i=1}^{n} \log p(x^{(i)}; \theta) \tag{20.3}$$

$$= \sum_{i=1}^{n} \log \sum_{z^{(i)}} p(x^{(i)}, z^{(i)}; \theta) \tag{20.4}$$

But explicitly finding the maximum likelihood estimates of the parameters θ may be hard since it will result in difficult non-convex optimization problems.[1] Here, the $z^{(i)}$'s are the latent random variables; and it is often the case that if the $z^{(i)}$'s were observed, then maximum likelihood estimation would be easy.

In such a setting, the EM algorithm gives an efficient method for maximum likelihood estimation. Maximizing $\ell(\theta)$ explicitly might be difficult, and our strategy will be to instead repeatedly construct a lower-bound on ℓ (E-step), and then optimize that lower-bound (M-step).[2]

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 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:

$$\log p(x;\theta) = \log \sum_z p(x,z;\theta) \tag{20.5}$$

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:[3]

$$\log p(x;\theta) = \log \sum_z p(x,z;\theta) \tag{20.6}$$

$$= \log \sum_z Q(z) \frac{p(x,z;\theta)}{Q(z)} \tag{20.7}$$

$$\geq \sum_z Q(z) \log \frac{p(x,z;\theta)}{Q(z)} \tag{20.8}$$

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.[4] By Jensen's inequality, we have

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

[1] It's mostly an empirical observation that the optimization problem is difficult to optimize.

[2] 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.

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

[4] 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.

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 (20.7) to equation (20.8).

Now, for *any* distribution Q, the formula 20.8 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 θ of the parameters, it seems natural to try to make the lower-bound tight at that value of θ. I.e., we will make the inequality above hold with equality at our particular value of θ. To make the bound tight for a particular value of θ, we need for the step involving Jensen's inequality in our derivation above to hold with equality. For this to be true, we know it is sufficient that 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

$$Q(z) = \frac{p(x,z;\theta)}{\sum_z p(x,z;\theta)} \tag{20.9}$$

$$= \frac{p(x,z;\theta)}{p(x;\theta)} \tag{20.10}$$

$$= p(z \mid x;\theta) \tag{20.11}$$

Thus, we simply set the Q's to be the posterior distribution of the z's given x and the setting of the parameters θ.

Indeed, we can directly verify that when $Q(z) = p(z \mid x;\theta)$, then equation (20.8) is an equality because:

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

For convenience, we call the expression in equation (20.8) the **evidence lower bound** (ELBO) and we denote it by:

$$\text{ELBO}(x;Q,\theta) = \sum_z Q(z) \log \frac{p(x,z;\theta)}{Q(z)} \tag{20.12}$$

With this equation, we can re-write equation (20.8) as:

$$\forall Q, \theta, x, \quad \log p(x;\theta) \geq ELBO(x;Q,\theta) \tag{20.13}$$

Intuitively, the EM algorithm alternatively updates Q and θ by a) setting $Q(z) = p(z \mid x;\theta)$ following equation (20.11) so that $\text{ELBO}(x;Q,\theta) = \log p(x;\theta)$ for x and the current θ, and b) maximizing $\text{ELBO}(x;Q,\theta)$ w.r.t θ 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 needs 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 $\{x^{(1)}, \ldots, x^{(n)}\}$. Note that the optimal choice of Q is $p(z \mid x;\theta)$, and it depends on the particular example x. Therefore here we will introduce n distributions $Q_1, \ldots, Q_n$, 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:

$$\ell(\theta) \geq \sum_i \text{ELBO}(x^{(i)}; Q_i, \theta) \tag{20.14}$$

$$= \sum_i \sum_{z^{(i)}} Q_i(z^{(i)}) \log \frac{p(x^{(i)}, z^{(i)}; \theta)}{Q_i(z^{(i)})} \tag{20.15}$$

For *any* set of distributions $Q_1, \ldots, Q_n$, the formula 20.14 gives a lower-bound on $\ell(\theta)$, and analogous to the argument around equation (20.11), the Q_i that attains equality satisfies:

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

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 θ.

Now, for this choice of the Q_i's, equation (20.14) gives a lower-bound on the loglikelihood ℓ that we're trying to maximize. This is the E-step. In the M-step of the algorithm, we then maximize our formula in equation (20.14) with respect to the parameters to obtain a new setting of the θ's. Repeatedly carrying out these two steps gives us the EM algorithm, which is as follows:

- Repeat until convergence:
  - (E-step) For each i, set:

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

  - (M-step) Set:

$$\theta := \arg\max_\theta \sum_{i=1}^n \text{ELBO}(x^{(i)}; Q_i, \theta) \tag{20.16}$$

$$= \arg\max_\theta \sum_i \sum_{z^{(i)}} Q_i(z^{(i)}) \log \frac{p(x^{(i)}, z^{(i)}; \theta)}{Q_i(z^{(i)})}. \tag{20.17}$$

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 $Q_i^{(t)}(z^{(i)}) := p(z^{(i)} \mid x^{(i)}; \theta^{(t)})$. We saw earlier that this choice ensures that Jensen's inequality, as applied to get equation (20.14), holds with equality, and hence:

$$\ell(\theta^{(t)}) = \sum_{i=1}^{n} \text{ELBO}(x^{(i)}; Q_i^{(t)}, \theta^{(t)}) \tag{20.18}$$

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

$$\begin{aligned}
\ell(\theta^{(t+1)}) &\geq \sum_{i=1}^{n} \text{ELBO}(x^{(i)}; Q_i^{(t)}, \theta^{(t+1)}) \\
&\qquad\qquad \text{(because ineqaulity 20.14 holds for all } Q \text{ and } \theta) \\
&\geq \sum_{i=1}^{n} \text{ELBO}(x^{(i)}; Q_i^{(t)}, \theta^{(t)}) \qquad \text{(see reason below)} \\
&= \ell(\theta^{(t)}) \qquad \text{(by equation (20.18))}
\end{aligned}$$

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

$$\text{ELBO}(Q, \theta) = \sum_{i=1}^{n} \text{ELBO}(x^{(i)}; Q_i, \theta) = \sum_{i} \sum_{z^{(i)}} Q_i(z^{(i)}) \log \frac{p(x^{(i)}, z^{(i)}; \theta)}{Q_i(z^{(i)})} \tag{20.19}$$

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 θ.

20.1 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 (20.12). There are several other forms of ELBO. First, we can rewrite

$$\begin{aligned} \text{ELBO}(x; Q, \theta) &= \mathbb{E}_{z\sim Q}[\log p(x, z; \theta)] - \mathbb{E}_{z\sim Q}[\log Q(z)] && (20.20) \\ &= \mathbb{E}_{z\sim Q}[\log p(x \mid z; \theta)] - D_{KL}(Q \mid\mid p_z) && (20.21) \end{aligned}$$

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:

$$D_{KL}(Q \mid\mid p_z) = \sum_z Q(z) \log \frac{Q(z)}{p(z)} \tag{20.22}$$

In many cases, the marginal distribution of z does not depend on the parameter θ. In this case, we can see that maximizing ELBO over θ is equivalent to maximizing the first term in 20.21. 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):

$$\text{ELBO}(x; Q, \theta) = \log p(x) - D_{KL}(Q \mid\mid p_{z|x}) \tag{20.23}$$

where $p_{z|x}$ is the conditional distribution of z given x under the parameter θ. 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 (20.11) before.

21 Mixture of Gaussians revisited

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

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

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

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

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

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

Let's maximize this with respect to μ_l . If we take the derivative with respect to μ_l, we find:

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

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

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

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 ϕ_j. Grouping together only the terms that depend on ϕ_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 ϕ_j's sum to 1, since they represent the probabilities $\phi_j = p(z^{(i)} = j; \phi)$. To deal with the constraint that $\sum_{j=1}^{k} \phi_j = 1$, we construct the Lagrangian

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

where β is the Lagrange multiplier.[1] Taking derivatives, we find:

[1] 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.

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

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 $-\beta = \sum_{i=1}^{n} \sum_k j = 1 w_j^{(i)} = \sum_{i=1}^{n} 1 = n$. (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 ϕ_j:

$$\phi_j := \frac{1}{n} \sum_{i=1}^{n} w_j^{(i)} \tag{21.1}$$

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

22 *Variational inference and variational auto-encoder*

Loosely speaking, variational auto-encoder[1] 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

[1] D.P. Kingma and M. Welling, ''Auto-Encoding Variational Bayes,'' *ArXiv Preprint ArXiv:1312.6114*, 2013.

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 θ 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:

$$z \sim \mathcal{N}(0, I_{k \times k}) \tag{22.1}$$

$$x \mid z \sim \mathcal{N}(g(z; \theta), \sigma^2 I_{d \times d}) \tag{22.2}$$

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

For the Gaussian mixture models in section 20.1, the optimal choice of $Q(z) = p(z \mid x; \theta)$ for each fixed θ, that is the posterior distribution of z, can be analytically computed. In many more complex models such as the model 22.2, it's intractable to compute the exact the posterior distribution $p(z \mid x; \theta)$.

Recall that from equation (20.13), 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 θ defined in equation (20.19):

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

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

$$\max_{Q \in \mathcal{Q}} \max_{\theta} \text{ELBO}(Q, \theta) \tag{22.3}$$

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 Blei, Kucukelbir, and McAuliffe 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 equation (22.3). 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 $p(z^{(i)} \mid x^{(i)}; \theta)$. It would make sense let all the means of the Q_i's be some function of $x^{(i)}$. Concretely, let $q(\cdot; \phi), v(\cdot; \phi)$ be two functions that map from dimension d to k, which are parameterized by ϕ and ψ, we assume that:

$$Q_i = \mathcal{N}(q(x^{(i)}; \phi), \mathrm{diag}(v(x^{(i)}; \psi))^2) \tag{22.4}$$

Here $\mathrm{diag}(w)$ 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.[2] 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**.

[2] q and v can also share parameters. We sweep this level of details under the rug in this note.

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 22.4)

Before optimizing the ELBO, let's first verify whether we can efficiently evaluate the value of the ELBO for fixed Q of the form 22.4 and θ. We rewrite the ELBO as

a function of ϕ, ψ, θ by:

$$\text{ELBO}(\phi, \psi, \theta) = \sum_{i=1}^{n} \mathbb{E}_{z^{(i)} \sim Q_i} \left[\log \frac{p(x^{(i)}, z^{(i)}; \theta)}{Q_i(z^{(i)})} \right], \tag{22.5}$$

$$\text{where } Q_i = \mathcal{N}(q(x^{(i)}; \phi), \text{diag}(v(x^{(i)}; \psi))^2) \tag{22.6}$$

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 $Q_i = N(q(x^{(i)}; \phi), \text{diag}(v(x^{(i)}; \psi))^2)$, we are able to do both efficiently.

Now let's optimize the ELBO. It turns out that we can run gradient ascent over ϕ, ψ, θ 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 20.1, computing the maximum is analytically feasible and relatively cheap, and therefore we did alternating maximization.) Mathematically, let η 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 θ is simple because:

$$\nabla_\theta \text{ELBO}(\phi, \psi, \theta) = \nabla_\theta \sum_{i=1}^{n} \mathbb{E}_{z^{(i)} \sim Q_i} \left[\frac{\log p(x^{(i)}, z^{(i)}; \theta)}{Q_i(z^{(i)})} \right] \tag{22.7}$$

$$= \nabla_\theta \sum_{i=1}^{n} \mathbb{E}_{z^{(i)} \sim Q_i} \left[\log p(x^{(i)}, z^{(i)}; \theta) \right] \tag{22.8}$$

$$= \sum_{i=1}^{n} \mathbb{E}_{z^{(i)} \sim Q_i} \left[\nabla_\theta \log p(x^{(i)}, z^{(i)}; \theta) \right] \tag{22.9}$$

But computing the gradient over ϕ and ψ is tricky because the sampling distribution Q_i depends on ϕ and ψ. (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 ϕ. 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_ϕ on ϕ 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:

$$z^{(i)} = q(x^{(i)};\phi) + v(x^{(i)};\psi) \odot \xi^{(i)} \text{ where } \xi^{(i)} \sim \mathcal{N}(0, I_{k\times k}) \tag{22.10}$$

Here $x \odot y$ denotes the entry-wise product of two vectors of the same dimension. Here we used the fact that $x \sim \mathcal{N}(\mu, \sigma^2)$ is equivalent to that $x = \mu + \xi\sigma$ with $\xi \sim \mathcal{N}(0,1)$. 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:

$$\mathbb{E}_{z^{(i)}\sim Q_i}\left[\log \frac{p(x^{(i)}, z^{(i)};\theta)}{Q_i(z^{(i)})}\right] \tag{22.11}$$

$$= \mathbb{E}_{\xi^{(i)}\sim\mathcal{N}(0,1)}\left[\log \frac{p(x^{(i)}, q(x^{(i)};\phi) + v(x^{(i)};\psi)\odot\xi^{(i)};\theta)}{Q_i(q(x^{(i)};\phi) + v(x^{(i)};\psi)\odot\xi^{(i)})}\right] \tag{22.12}$$

It follows that:

$$\nabla_\phi \mathbb{E}_{z^{(i)}\sim Q_i}\left[\log \frac{p(x^{(i)}, z^{(i)};\theta)}{Q_i(z^{(i)})}\right] \tag{22.13}$$

$$= \nabla_\phi \mathbb{E}_{\xi^{(i)}\sim\mathcal{N}(0,1)}\left[\log \frac{p(x^{(i)}, q(x^{(i)};\phi) + v(x^{(i)};\psi)\odot\xi^{(i)};\theta)}{Q_i(q(x^{(i)};\phi) + v(x^{(i)};\psi)\odot\xi^{(i)})}\right] \tag{22.14}$$

$$= \mathbb{E}_{\xi^{(i)}\sim\mathcal{N}(0,1)}\left[\nabla_\phi \log \frac{p(x^{(i)}, q(x^{(i)};\phi) + v(x^{(i)};\psi)\odot\xi^{(i)};\theta)}{Q_i(q(x^{(i)};\phi) + v(x^{(i)};\psi)\odot\xi^{(i)})}\right] \tag{22.15}$$

We can now sample multiple copies of $\xi^{(i)}$'s to estimate the expectation in the RHS of the equation above.[3] We can estimate the gradient with respect to ψ similarly, and with these, we can implement the gradient ascent algorithm to optimize the ELBO over ϕ, ψ, θ.

[3] Empirically people sometimes just use one sample to estimate it for maximum computational efficiency.

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

References

1. D. M. Blei, A. Kucukelbir, and J. D. McAuliffe, ''Variational Inference: A Review for Statisticians,'' *Journal of the American Statistical Association*, vol. 112, no. 518, pp. 859–877, 2017 (cit. on p. 130).
2. D. P. Kingma and M. Welling, ''Auto-Encoding Variational Bayes,'' *ArXiv Preprint ArXiv:1312.6114*, 2013 (cit. on pp. 128, 132).
3. M. J. Kochenderfer and T. A. Wheeler, *Algorithms for Optimization*. MIT Press, 2019 (cit. on p. vii).