EECS 16B Designing Information Devices and Systems II
Fall 2021

Note 20: Classification

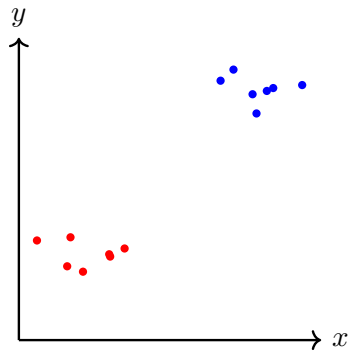
Overview

In this note, we will look at various related techniques for learning how to *classify* data. Classification is when we want to take an observation and return a discrete label corresponding to that observation. You have already seen this idea in 16A during the positioning module. Given recorded data, you had to decide whether a particular Gold Code was present or not — this is called a problem of binary classification since there are two labels: "present" or "not present." But you also had to decide which shift was present — this also involved choosing among discrete alternatives and so can be considered a classification problem. It is not binary however since there are more than two choices.

In this note, we will focus on the problem of *binary classification* — when there are two alternatives. We do this to focus on the issue of how to learn the classification rule from data. In 16A's positioning module, you already knew the codes and designed the rules to classify incoming data. Here, we want to learn the key parameters of the rules from training data: lots of labeled examples of each class. This training data plays the same role[1] that system input-output traces played in system identification.

1 Known Approaches

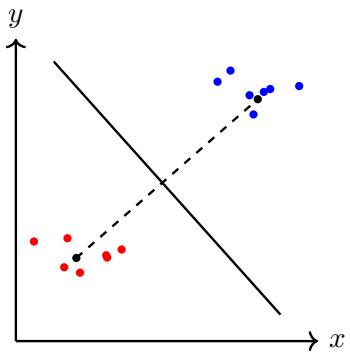
Before we do anything, let's see what this problem looks like visually. Imagine that our data is two-dimensional, and our points look something like the following:



The categories of our points are represented by their color. Our points can be separated by a line passing from the top-left to the bottom-right of our diagram.

[1] Notice the rough parallels here to what we have seen in control. We could get a discrete-time model and control law for a system by discretizing known continuous-time dynamics given by a differential equation and then doing control design for it. The continuous-time differential equation represents an underlying known model for ground-truth. But we also know what to do if we don't have that. We could use system identification to learn the discrete-time model from data obtained from traces of state input pairs. In 16A, we already knew the underlying model that generates the data that we would be classifying — the Gold Codes. Here, we are going to be seeing how to do the counterpart of system identification for classification.

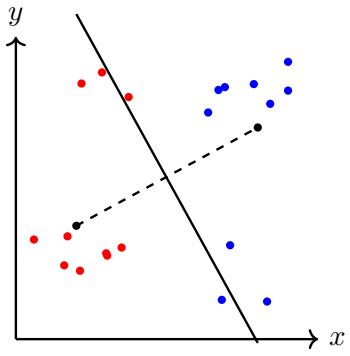
How could we construct such a line? One natural thing to do would be to take the average location of the points in each of the two categories, connect them with a segment, and then draw the perpendicular bisector of that segment, as shown below:



This strategy often works reasonably well. How can we understand what is going on here in a more systematic way? First, we are using the average location of the points in a category as a kind of paradigmatic representative for that category. This is easy to learn from training data. Then, what we are doing for classification is asking which paradigmatic representative we are closer to. We assign a point to the category which has the closest[2] paradigmatic representative. It is a basic fact of geometry that for two points, the points that are equidistant to both of them are the hyperplane that is the perpendicular bisector of the segment between the two points. Hence, such planes are useful for binary classification.

So are we done? To understand whether we need something better, we need to explore what can happen. We stick with the assumption that a hyperplane (or in the 2D case, a line) is the underlying right answer for how to separate two categories. However, the data that we learn from need not be tightly clustered around paradigmatic representatives. What else could happen?

For instance, consider the following sets of points:



Visually, one can see that the best dividing line is a vertical line passing between the two categories. That cleanly splits the two categories. But because the observed averages (also called "means" in the literature to reflect language used in probability) of the two categories are not at the same height, their perpendicular bisector slopes downwards, meaning that it does not divide the data points correctly.

[2] If you recall, this was also the motivation used in 16A as a precursor to the maximum correlation approach. You noticed that $\|\vec{x}-\vec{t}\|^2 = \vec{x}^\top\vec{x} + \vec{t}^\top\vec{t} - 2\vec{t}^\top\vec{x}$. If we assumed that all of the templates $\vec{t}$ had the same length, then *minimizing* this distance to the templates was the same as *maximizing* the correlation with the templates. (After all, the $\vec{x}^\top\vec{x}$ is the same for each of the potential templates. So the only thing that is different for different templates (all of which have the same length) is the correlation.) If you think about it, this is actually a way that you could have discovered the Euclidean inner product as a concept for yourself.

Fundamentally, this "means-based" approach to learning does not work well in these kinds of cases there is some information in the distribution that can't be accurately captured by the mean of the data points of each category — it just calculates the averages. It also doesn't care about whether points are being classified correctly, since the separating line stays the same as long as the means are the same, even when it causes training data points to be misclassified.

2 Directly learning the rule

What this suggests is that we should try to learn the classification rule directly. For binary classification, a separating line is reasonable. How can we express this mathematically? If we think of the n-dimensional inputs that we want to classify as $\vec{x}$, this means that we want to compute $\sum_{i=1}^{n} w[i]x[i]$, where $w[i]$ and $x[i]$ are the i^{th} entry of $\vec{w}$ and $\vec{x}$ respectively, and compare it to some threshold. This is equivalent to computing $w[0]+\sum_{i=1}^{n} w[i]x[i]$ and comparing it to 0 — i.e. looking at the sign of the result. For notational convenience, we can augment our input with a 1 in the zeroth position and then what we want to learn is an $n+1$ dimensional vector $\vec{w}$.

How do we learn an $n+1$-dimensional vector $\vec{w}$ given a bunch of data? This sounds exactly like what we had to do in system identification, and so we might as well do what we know how to do — least squares.

But to do least-squares, we need to figure out how to get the "$\vec{y}$" that least squares is going to project onto an appropriate subspace to get the estimate for $\vec{w}$. In system-identification, the observed next states provided that information. After all, the goal of a system model is to predict the next state given the current state and the input. What should play that role here?

What we have are the binary labels. This is categorical information. What we need are target y values that are real numbers. Since we're going to use the sign of $\vec{x}^\top \vec{w}$ to determine the classification, it is natural to assign all the points in one category the target value -1, and all the points in the other the target value $+1$. Then we use least squares to construct an affine function that minimizes the squared error between its predictions and the actual values. We then draw our dividing line by looking at where our function equals 0.

To be explicit: let our raw observed training points be n-dimensional vectors $\vec{x}_1, \vec{x}_2, \ldots, \vec{x}_m$. We silently modify our data points to become:

$$\begin{bmatrix} 1 \\ \vec{x}_1 \end{bmatrix}, \begin{bmatrix} 1 \\ \vec{x}_2 \end{bmatrix}, \cdots, \begin{bmatrix} 1 \\ \vec{x}_m \end{bmatrix},$$

so that we can use the sign of $\vec{x}^\top \vec{w}$ to classify points.

Our target linear function is of the form $f(\vec{x}) = \vec{w}^\top \vec{x}$ for some unknown $\vec{w}$. Note that, if we want an affine function, to allow for a constant offset by varying the first entry of $\vec{w}$. Now, let the binary labels of our sample points be $\ell_1, \ldots, \ell_m$, where $\ell_i = +1$ if $\vec{x}_i$ is a point in the category we deem[3] "positive" and $\ell_i = -1$ if $\vec{x}_i$ is a point in the category we deem "negative."

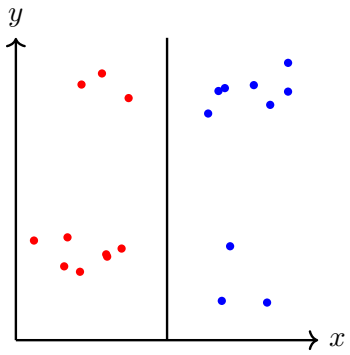
The standard least squares approach to classification is to set up the approximate equation:

$$\begin{bmatrix} \vec{x}_1^\top \\ \vec{x}_2^\top \\ \vdots \\ \vec{x}_m^\top \end{bmatrix} \vec{w} \approx \begin{bmatrix} \ell_1 \\ \ell_2 \\ \vdots \\ \ell_m \end{bmatrix} \tag{1}$$

[3] "+" and "-" are conventional names for these binary categories, but in actual situations, these could refer to any pair of mutually exclusive alternatives. "Cats" vs "Dogs" or "Red" vs "Blue." Anything.

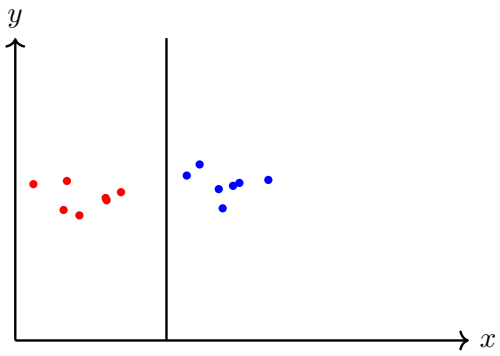
and solve it to get the solution $\vec{w}$ that minimizes $\sum_{i=1}^{m}(\vec{x}_i^\top \vec{w} - \ell_i)^2$.

We can try it for our example to see the resulting dividing line

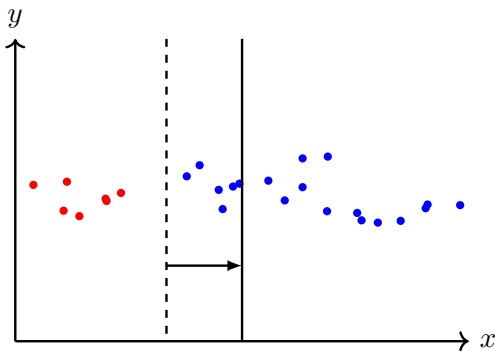


and this dividing line seems pretty good.

So are we done? Unfortunately not. Consider the following dataset:

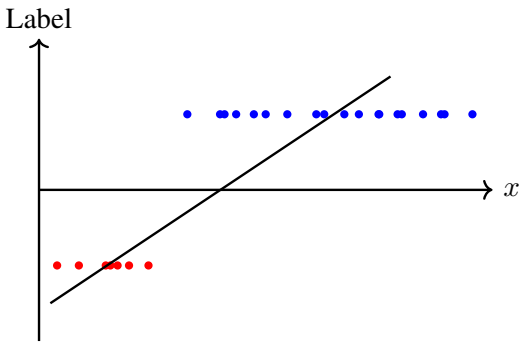


Least squares works correctly, as we expect, and constructs a dividing line. Now let's add some more points far to the right that are in the same $+1$ category. Our existing dividing line *already* classifies them correctly, as they are already on the right side of the line. But what happens when we run least squares again?



As shown above, our new points "drag" our dividing line to the right, making our classifier *worse* than before!

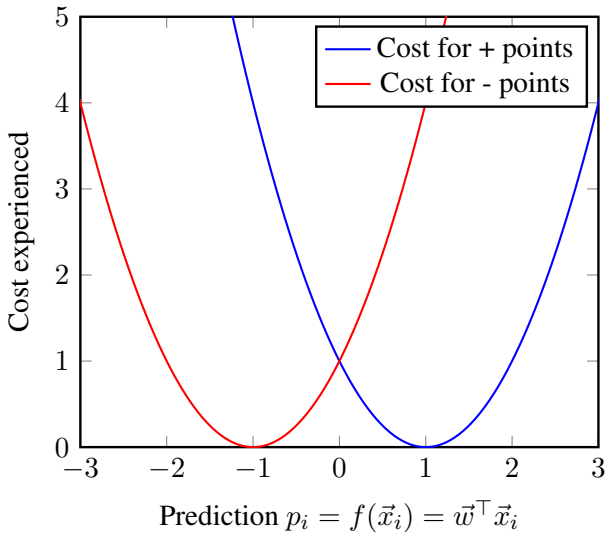
Why is least squares failing us? To see the problem, we will plot the x-coordinates of our points against their labels ($+1$ or -1), along with the line of best fit, omitting the y-coordinate since we do not want it to be important for classification in this example:



Observe that the slope of the line of best fit is "dragged down" in order to avoid overestimating the labels of the points furthest to the right. However, this causes the dividing line to move to the right, making our linear classifier less accurate on the training data.

From the internal point of view of least-squares, the quantity $f(\vec{x}) = \vec{x}^\top \vec{w}$ is supposed to be a real number that it is trying to predict correctly by picking the right weights $\vec{w}$. Meanwhile, what we actually care about is getting the sign correct. Least-squares is minimizing the sum of the squared differences of the prediction and the true label (it is called "least squares" after all). So given a point with label $+1$, a prediction of -1 or $+3$ would both be penalized by a cost of $2^2 = 4$. But the former prediction is much worse than the latter, since the first leads to a misclassification of the point while the second gives a prediction whose classification agrees with the training data.

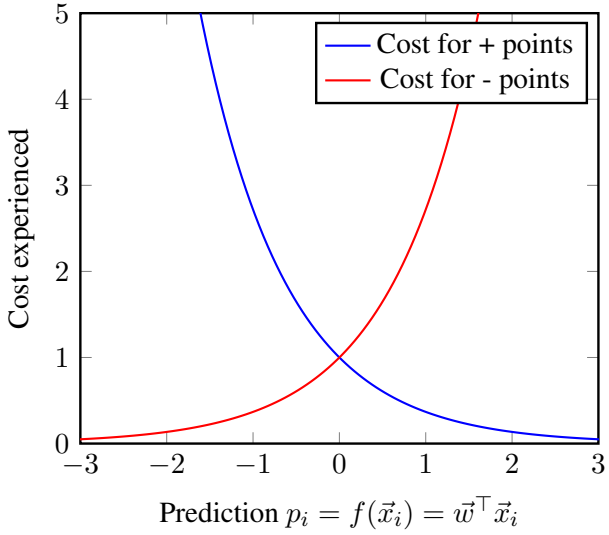
We can illustrate this behavior graphically. Let $p_i = \vec{x}_i^\top \vec{w}$ be the predicted value of point i, and let ℓ_i be its label. Plotting the cost function $(p_i - \ell_i)^2$ for points labeled -1 and $+1$, we obtain



The problem here comes from the cost curves bending back upwards where they really shouldn't. To achieve our goal of classification, we have no desire to penalize strong predictions that have the correct sign. But just using a quadratic cost makes us do this.

3 Exponential Cost Function

Ideally, we'd like a cost function that only severely penalizes errors that could lead to misclassifications. One such function is the following: for points in the $+1$ category, we use e^{-p_i}, and for points in the -1 category, we use e^{+p_i}. Plotting these functions, we obtain:



These look a lot better! For points in the $+1$ category, the predictions of -1 and $+3$ are no longer penalized equally: -1 has a high cost, while $+3$ is essentially not penalized at all, even though both correspond to an absolute error of 2 from the true label. So the issue we faced earlier about points far to one side of the dividing line has been resolved. This modification is known as using an *exponential cost function*.

Now, let's look at how we can compute the classifier that minimizes this new cost function. This is not least-squares anymore.

Let the cost function corresponding to the ith point be $c_i(p)$. In the case of quadratic-cost linear regression, this cost function looks like

$$c_i(p) = (p - \ell_i)^2.$$

Now, with our exponential cost function, our cost function becomes

$$c_i(p) = e^{-\ell_i p}.$$

Since in either case our goal is to pick the linear classifier with minimum cost, we can express our problem as trying to compute

$$\underset{\vec{w}}{\text{argmin}} \left(\sum_{i=1}^{m} c_i(f(\vec{x}_i)) \right) = \underset{\vec{w}}{\text{argmin}} \left(\sum_{i=1}^{m} c_i(\vec{w}^\top \vec{x}_i) \right).$$

The question is how can we solve such problems. In 16A, we were able to use geometry and the Pythagorean theorem to solve the least-squares case. Can we somehow leverage that work for more general cases?

4 Quadratic Approximations

How can we compute the desired $\vec{w}$? If our cost function were quadratic, we know how to use least squares to immediately identify the minimizing value of $\vec{w}$. But with our exponential cost function, or with some other arbitrary cost function, least squares doesn't give us the desired answer immediately anymore.

Rather than trying to solve the problem in one shot, we will instead try to develop an iterative way of computing the solution. Given a particular value of $\vec{w}$, we will attempt to make an improvement $\vec{\delta w}$ to $\vec{w}$ that lowers our cost. Then we will repeat this process, continuously updating $\vec{w}$ until we converge to a minimum.

How can we find such an improvement? Well, we know how to minimize quadratic costs using least squares. So if we approximate our cost function by a quadratic near a particular $\vec{w}$, we can hope to use least squares to give us an improvement. Then re-approximate the cost function near the improved $\vec{w}$, and repeat!

So the main remaining task is coming up with a quadratic approximation to our cost function. For convenience, let

$$c(\vec{w}) = \sum_{i=1}^{m} c_i(\vec{x}_i^\top \vec{w}),$$

so we are simply trying to compute

$$\underset{\vec{w}}{\operatorname{argmin}} \left(c(\vec{w})\right).$$

Let's review what we know that could be helpful. One thing we know is how to *linearize* $c(\vec{w})$ around a particular expansion point $\vec{w}_*$. We know that such a linear approximation looks like

$$c(\vec{w}_* + \vec{\delta w}) \approx c(\vec{w}_*) + \begin{bmatrix} \frac{\partial c}{\partial \vec{w}[1]} & \cdots & \frac{\partial c}{\partial \vec{w}[n]} \end{bmatrix} \vec{\delta w},$$

where all the partial derivatives are taken at $\vec{w} = \vec{w}_*$.

Another thing we know is how to compute a quadratic approximation for a *scalar* function $f(x)$ using the Taylor series about some point x_*, where

$$f(x_* + \delta x) \approx f(x_*) + \left(\frac{\mathrm{d}f}{\mathrm{d}x}\bigg|_{x=x_*}\right)(\delta x) + \frac{1}{2}\left(\frac{\mathrm{d}^2 f}{\mathrm{d}x^2}\bigg|_{x=x_*}\right)(\delta x)^2.$$

Our plan will be to try and understand where the quadratic term of the Taylor expansion comes from, and then apply similar reasoning to determine the quadratic term in our expansion for $c(\vec{w})$.

One argument is that we want our approximation of $f(x)$ to have the same value, first, and second derivatives at $x = x_*$, and it is the only quadratic polynomial with that property. While this argument makes sense, it is not particularly useful since it does not obviously or immediately generalize to our multidimensional case.

Alternatively, we can ask ourselves what the purpose of the quadratic term is. With just the linear term, we model $f(x)$ as a linear function passing through $(x_*, f(x_*))$ with constant slope. The quadratic term essentially allows us to better approximate $f(x)$ by treating its *slope* as a linear function. We can see this by first approximating the derivative of f as a linear function

$$\frac{\mathrm{d}f}{\mathrm{d}x}\bigg|_{x=x_*+\delta x} \approx \left(\frac{\mathrm{d}f}{\mathrm{d}x}\bigg|_{x=x_*}\right) + \left(\frac{\mathrm{d}^2 f}{\mathrm{d}x^2}\bigg|_{x=x_*}\right)(\delta x).$$

We then integrate with respect to δx from x_* to $x_* + \delta x$ to see that

$$f(x_* + \delta x) - f(x_*) = \left(\frac{\mathrm{d}f}{\mathrm{d}x}\bigg|_{x=x_*} \right) (\delta x) + \frac{1}{2} \left(\frac{\mathrm{d}^2 f}{\mathrm{d}x^2}\bigg|_{x=x_*} \right) (\delta x)^2,$$

which when rearranged gives us our quadratic approximation. This "integration" is computing an area. The $\frac{1}{2}$ comes from the area of a triangle — the average slope over the duration δx isn't the initial slope $\frac{\mathrm{d}f}{\mathrm{d}x}\Big|_{x=x_*}$ nor is it the estimate for the final slope $\frac{\mathrm{d}f}{\mathrm{d}x}\Big|_{x=x_*} + \left(\frac{\mathrm{d}^2 f}{\mathrm{d}x^2}\Big|_{x=x_*} \right) \delta x$. It is halfway in between: $\frac{\mathrm{d}f}{\mathrm{d}x}\Big|_{x=x_*} + \frac{1}{2} \left(\frac{\mathrm{d}^2 f}{\mathrm{d}x^2}\Big|_{x=x_*} \right) \delta x$. Multiplying this average slope by the duration δx gives rise to the approximate update to the function value $f(x_*)$ to give us an estimate of $f(x_* + \delta x)$.

This line of reasoning is more amenable to generalization to functions of vectors. Consider the derivative of $c(\vec{w})$. We have that

$$\frac{\mathrm{d}c}{\mathrm{d}\vec{w}} = \begin{bmatrix} \frac{\partial c}{\partial w[1]} & \cdots & \frac{\partial c}{\partial w[n]} \end{bmatrix}.$$

This is a function that takes in a column vector $\vec{w}$ (representing where we want to evaluate this derivative) and returns a row vector. We do not yet know how to linearize such a function. However, we *do* know how to linearize functions that take in column vectors and return *column* vectors. How can we get such a function? We take the transpose of the derivative! Doing so, we obtain

$$\left(\frac{\mathrm{d}c}{\mathrm{d}\vec{w}} \right)^\top = \begin{bmatrix} \frac{\partial c}{\partial w[1]} \\ \vdots \\ \frac{\partial c}{\partial w[n]} \end{bmatrix}.$$

Now we can linearize this about $\vec{w} = \vec{w}_*$, to obtain

$$\left(\frac{\mathrm{d}c}{\mathrm{d}\vec{w}}\bigg|_{\vec{w}=\vec{w}_*+\vec{\delta w}} \right)^\top \approx \begin{bmatrix} \frac{\partial c}{\partial w[1]} \\ \vdots \\ \frac{\partial c}{\partial w[n]} \end{bmatrix}\bigg|_{\vec{w}=\vec{w}_*} + \begin{bmatrix} \frac{\partial^2 c}{\partial w[1]\partial w[1]} & \cdots & \frac{\partial c}{\partial w[n]\partial w[1]} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 c}{\partial w[1]\partial w[n]} & \cdots & \frac{\partial c}{\partial w[n]\partial w[n]} \end{bmatrix}\bigg|_{\vec{w}=\vec{w}_*} \vec{\delta w}.$$

Finally, we can take transposes once more to obtain

$$\frac{\mathrm{d}c}{\mathrm{d}\vec{w}}\bigg|_{\vec{w}=\vec{w}_*+\vec{\delta w}} \approx \begin{bmatrix} \frac{\partial c}{\partial w[1]} & \cdots & \frac{\partial c}{\partial w[n]} \end{bmatrix}\bigg|_{\vec{w}=\vec{w}_*} + (\vec{\delta w})^\top \begin{bmatrix} \frac{\partial^2 c}{\partial w[1]\partial w[1]} & \cdots & \frac{\partial c}{\partial w[1]\partial w[n]} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 c}{\partial w[n]\partial w[1]} & \cdots & \frac{\partial c}{\partial w[n]\partial w[n]} \end{bmatrix}\bigg|_{\vec{w}=\vec{w}_*}.$$

What next? Now that we've got a more accurate model of the derivative of our function, how can we use it to develop the desired quadratic approximation? In the purely scalar case, we could simply integrate, and the quadratic approximation popped out. However, we can't do the same thing here, since our functions all take in multiple arguments.

Let's think about what should happen at an intuitive level. Imagine trying to compute $c(\vec{w}_* + \vec{\delta w})$ using this new model of its derivative. We want to figure out how much $c(\vec{w})$ changes as we move $\vec{w}$ from $\vec{w}_*$

to $\vec{w}_* + \vec{\delta w}$. How a function changes under small perturbations, of course, is governed by its derivative. The first term in our above linear approximation is a constant, so it will contribute equally throughout this movement. The second term, however, grows linearly with $\vec{\delta w}$. So it will not contribute at all to begin with, but will contribute proportional to $\vec{\delta w}$ at the end of the movement. This suggests a "triangle-like" behavior similar to what we saw in the scalar case, where we should halve its impact to account for its linear growth over the movement. Basically, look at the average derivative for the actual range traveled in the input.

Thus, our intuition suggests that we can write

$$c(\vec{w}_* + \vec{\delta w}) - c(\vec{w}_*) = (\left[\begin{array}{ccc} \frac{\partial c}{\partial w[1]} & \cdots & \frac{\partial c}{\partial w[n]} \end{array}\right]\Big|_{\vec{w}=\vec{w}_*} + \frac{1}{2}(\vec{\delta w})^\top \left[\begin{array}{ccc} \frac{\partial^2 c}{\partial w[1]\partial w[1]} & \cdots & \frac{\partial^2 c}{\partial w[1]\partial w[n]} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 c}{\partial w[n]\partial w[1]} & \cdots & \frac{\partial^2 c}{\partial w[n]\partial w[n]} \end{array}\right]\Big|_{\vec{w}=\vec{w}_*})(\vec{\delta w}) \tag{2}$$

$$= \left[\begin{array}{ccc} \frac{\partial c}{\partial w[1]} & \cdots & \frac{\partial c}{\partial w[n]} \end{array}\right]\Big|_{\vec{w}=\vec{w}_*}(\vec{\delta w}) + \frac{1}{2}(\vec{\delta w})^\top \left[\begin{array}{ccc} \frac{\partial^2 c}{\partial w[1]\partial w[1]} & \cdots & \frac{\partial^2 c}{\partial w[1]\partial w[n]} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 c}{\partial w[n]\partial w[1]} & \cdots & \frac{\partial^2 c}{\partial w[n]\partial w[n]} \end{array}\right]\Big|_{\vec{w}=\vec{w}_*}(\vec{\delta w}) \tag{3}$$

which can be rearranged to give us a quadratic approximation for $c(\vec{w})$ near $\vec{w} = \vec{w}_*$.

The matrix $\left[\begin{array}{ccc} \frac{\partial^2 c}{\partial w[1]\partial w[1]} & \cdots & \frac{\partial^2 c}{\partial w[1]\partial w[n]} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 c}{\partial w[n]\partial w[1]} & \cdots & \frac{\partial^2 c}{\partial w[n]\partial w[n]} \end{array}\right]\Big|_{\vec{w}=\vec{w}_*}$ in the quadratic term is known as the *Hessian*, often denoted by H. It is important to note that it is *not*, strictly speaking, the second derivative of c with respect to $\vec{w}$. *If* we were able to linearize

$$\left(\frac{\mathrm{d}c}{\mathrm{d}\vec{w}}\Big|_{\vec{w}=\vec{w}_*+\vec{\delta w}}\right) - \left(\frac{\mathrm{d}c}{\mathrm{d}\vec{w}}\Big|_{\vec{w}=\vec{w}_*}\right) \approx A(\vec{\delta w}),$$

using some matrix A, then A would be our desired second derivative, since we could post-multiply it by $\vec{\delta w}$ to obtain the change in the derivative over a small perturbation. But instead we can only write

$$\left(\frac{\mathrm{d}c}{\mathrm{d}\vec{w}}\Big|_{\vec{w}=\vec{w}_*+\vec{\delta w}}\right) - \left(\frac{\mathrm{d}c}{\mathrm{d}\vec{w}}\Big|_{\vec{w}=\vec{w}_*}\right) \approx (\vec{\delta w})^\top H,$$

so the Hessian matrix is not quite the second derivative, though it can be thought of as a *representation*[4] of it.

Now that we have developed the necessary intuition to define a quadtratic approximation of interest, we can "check our work." How can we "check our work" in this case? By seeing if what we got is consistent with what we know about scalar functions. Essentially, we know we can treat $c(\vec{w})$ as a scalar function of a

[4]This issue of representing derivatives is important and it turns out, at some point, it will no longer be possible to represent derivatives using ordinary vectors and matrices. But if you've internalized the spirit of 16AB, you will have no trouble coming up with the relevant definitions to deal with the multidimensional arrays that will arise as you consider higher-order derivatives. Although they are out of the scope of this course, if you follow this path forward, you will naturally discover higher-order *tensors* for yourself. The key is just to not confuse derivatives with their representatives.

scalar argument defined by only evaluating $c(\vec{w})$ along a particular direction starting at $\vec{w}_*$ and compute the quadratic approximation of this scalar function of a scalar argument using the Taylor series. We will then rewrite that approximation to show that it agrees with the above approximation.

To do so, let $\vec{r}$ be a unit vector pointing along our arbitrary direction, and let t be a (presumably) small scalar argument. Thus, $t\vec{r}$ is a small perturbation in the direction of $\vec{r}$. By our linearization of the derivative of $c(\cdot)$,

$$\begin{aligned}\left.\frac{\mathrm{d}c}{\mathrm{d}\vec{w}}\right|_{\vec{w}=\vec{w}_*+t\vec{r}} &\approx \left.\begin{bmatrix}\frac{\partial c}{\partial w[1]} & \cdots & \frac{\partial c}{\partial w[n]}\end{bmatrix}\right|_{\vec{w}=\vec{w}_*} + (t\vec{r})^\top \left.\begin{bmatrix}\frac{\partial^2 c}{\partial w[1]\partial w[1]} & \cdots & \frac{\partial^2 c}{\partial w[1]\partial w[n]} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 c}{\partial w[n]\partial w[1]} & \cdots & \frac{\partial^2 c}{\partial w[n]\partial w[n]}\end{bmatrix}\right|_{\vec{w}=\vec{w}_*} \\ &= \left.\begin{bmatrix}\frac{\partial c}{\partial w[1]} & \cdots & \frac{\partial c}{\partial w[n]}\end{bmatrix}\right|_{\vec{w}=\vec{w}_*} + (\vec{r})^\top \left.\begin{bmatrix}\frac{\partial^2 c}{\partial w[1]\partial w[1]} & \cdots & \frac{\partial^2 c}{\partial w[1]\partial w[n]} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 c}{\partial w[n]\partial w[1]} & \cdots & \frac{\partial^2 c}{\partial w[n]\partial w[n]}\end{bmatrix}\right|_{\vec{w}=\vec{w}_*} t.\end{aligned}$$

Now, we can define the scalar function of a scalar argument $c_r(t) = c(\vec{w}_* + t\vec{r})$. Then

$$\frac{\mathrm{d}c_r}{\mathrm{d}t} = \left(\left.\frac{\mathrm{d}c}{\mathrm{d}\vec{w}}\right|_{\vec{w}=\vec{w}_*+t\vec{r}}\right)\vec{r},$$

so by our linearization

$$\frac{\mathrm{d}c_r}{\mathrm{d}t} \approx \begin{bmatrix}\frac{\partial c}{\partial w[1]} & \cdots & \frac{\partial c}{\partial w[n]}\end{bmatrix}\vec{r} + (\vec{r})^\top \begin{bmatrix}\frac{\partial^2 c}{\partial w[1]\partial w[1]} & \cdots & \frac{\partial^2 c}{\partial w[1]\partial w[n]} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 c}{\partial w[n]\partial w[1]} & \cdots & \frac{\partial^2 c}{\partial w[n]\partial w[n]}\end{bmatrix}(\vec{r})t.$$

Now, we can integrate with respect to t, to find that

$$\begin{aligned}c_r(t) - c_r(0) &\approx \begin{bmatrix}\frac{\partial c}{\partial w[1]} & \cdots & \frac{\partial c}{\partial w[n]}\end{bmatrix}t\vec{r} + \frac{1}{2}(\vec{r})^\top \begin{bmatrix}\frac{\partial^2 c}{\partial w[1]\partial w[1]} & \cdots & \frac{\partial^2 c}{\partial w[1]\partial w[n]} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 c}{\partial w[n]\partial w[1]} & \cdots & \frac{\partial^2 c}{\partial w[n]\partial w[n]}\end{bmatrix}(\vec{r})t^2 \\ &= \begin{bmatrix}\frac{\partial c}{\partial w[1]} & \cdots & \frac{\partial c}{\partial w[n]}\end{bmatrix}t\vec{r} + \frac{1}{2}(t\vec{r})^\top \begin{bmatrix}\frac{\partial^2 c}{\partial w[1]\partial w[1]} & \cdots & \frac{\partial^2 c}{\partial w[1]\partial w[n]} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 c}{\partial w[n]\partial w[1]} & \cdots & \frac{\partial^2 c}{\partial w[n]\partial w[n]}\end{bmatrix}(t\vec{r}).\end{aligned}$$

Substituting back our definition of $c_r(t)$, defining $\delta\vec{w} = t\vec{r}$, and rearranging, we find that

$$c(\vec{w}_* + \delta\vec{w}) \approx c(\vec{w}_*) + \begin{bmatrix}\frac{\partial c}{\partial w[1]} & \cdots & \frac{\partial c}{\partial w[n]}\end{bmatrix}(\delta\vec{w}) + \frac{1}{2}(\delta\vec{w})^\top \begin{bmatrix}\frac{\partial^2 c}{\partial w[1]\partial w[1]} & \cdots & \frac{\partial^2 c}{\partial w[1]\partial w[n]} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 c}{\partial w[n]\partial w[1]} & \cdots & \frac{\partial^2 c}{\partial w[n]\partial w[n]}\end{bmatrix}(\delta\vec{w}),$$

which agrees. Since we can create any small $\delta\vec{w}$ by appropriately choosing t and $\vec{r}$, the above approximation

is reasonable for any small $\delta\vec{w}$. Thus, we have shown how the multivariate quadratic approximation is consistent with the Taylor series in the scalar case. Such consistency is always a good way to sanity check our definitions and approximations.

5 Iteratively Reweighted Least Squares

Now that we know how to come up with a quadratic approximation for a general function $c(\vec{w})$, let's see how to use it in our case. One approach is natural — we should iterate. We have seen this kind of iterative pattern before. The idea is to approximate the problem as something we know how to solve, solve the approximate problem, and then pass the solution on update the approximation and repeat the process. In effect, we are counting on the iterations to take care of the approximation error. Explicitly, we start with a candidate solution $\vec{w}_*$, and want to first come up with a quadratic approximation around it, and then find its minimizing $\delta\vec{w}$ in order to improve our solution. We defined

$$c(\vec{w}) = \sum_{i=1}^{m} c_i(\vec{w}^\top \vec{x}_i).$$

To approximate this function around a point, we need to know its partial derivatives. Differentiating first with respect to the gth component of $\vec{w}$, we can use the standard univariate chain rule you know from calculus to obtain

$$\frac{\partial c}{\partial w[g]} = \sum_{i=1}^{m} c_i'(\vec{w}^\top \vec{x}_i) \left(\frac{\partial}{\partial w[g]} (\vec{w}^\top \vec{x}_i) \right).$$

Note that we use $c_i'(\vec{w}^\top \vec{x}_i)$ to represent the derivative of $c_i(p)$ with respect to p, evaluated at $\vec{w}^\top \vec{x}_i$. $c_i''(\cdot)$ is the analogous notation that we will use later to represent the second derivative. We use this notation for compactness later in the calculation.

Now, observe that we can expand out the inner product to obtain

$$\vec{w}^\top \vec{x}_i = w[1]x_i[1] + w[2]x_i[2] + \cdots + w[g]x_i[g] + \cdots + w[n]x_i[n]$$
$$\implies \quad \frac{\partial}{\partial w[g]} (\vec{w}^\top \vec{x}_i) = x_i[g],$$

since all the other terms in the summation do not vary with $w[g]$. Thus,

$$\frac{\partial c}{\partial w[g]} = \sum_{i=1}^{m} c_i'(\vec{w}^\top \vec{x}_i) x_i[g].$$

Of course, to compute the Hessian, we require the second partial derivatives as well. Differentiating the above quantity with respect to $w[h]$, we obtain

$$\begin{aligned}
\frac{\partial^2 c}{\partial w[h] \partial w[g]} &= \frac{\partial}{\partial w[h]} \sum_{i=1}^{m} c_i'(\vec{w}^\top \vec{x}_i) x_i[g] \\
&= \sum_{i=1}^{m} \frac{\partial}{\partial w[h]} \left(c_i'(\vec{w}^\top \vec{x}_i) x_i[g] \right) \\
&= \sum_{i=1}^{m} x_i[g] \frac{\partial}{\partial w[h]} c_i'(\vec{w}^\top \vec{x}_i)
\end{aligned}$$

$$= \sum_{i=1}^{m} x_i[g] c_i''(\vec{w}^\top \vec{x}_i) \frac{\partial}{\partial w[h]} (\vec{w}^\top \vec{x}_i),$$

using the chain rule once again. Note that we pulled $x_i[g]$ out of the derivative as a constant, since it does not vary with $w[h]$. Now, we can use our above result on the partial derivatives of $\vec{w}^\top \vec{x}_i$, to find that

$$\frac{\partial^2 c}{\partial w[h] \partial w[g]} = \sum_{i=1}^{m} c_i''(\vec{w}^\top \vec{x}_i) x_i[g] x_i[h].$$

Next, we will use this result to compute the Hessian. Observe that the entry at the gth row and hth column of the Hessian is

$$\begin{aligned} H[g][h] &= \left. \frac{\partial^2 c}{\partial w[h] \partial w[g]} \right|_{\vec{w}=\vec{w}_*} \\ &= \sum_{i=1}^{m} c_i''(\vec{w}_*^\top \vec{x}_i) x_i[g] x_i[h] \\ &= \sum_{i=1}^{m} c_i''(\vec{w}_*^\top \vec{x}_i) (\vec{x}_i \vec{x}_i^\top)[g][h], \end{aligned}$$

since $(\vec{x}_i \vec{x}_i^\top)[g][h] = x_i[g] x_i[h]$. Thus, we can express the matrix

$$H = \sum_{i=1}^{m} c_i''(\vec{w}_*^\top \vec{x}_i) \vec{x}_i \vec{x}_i^\top.$$

Finally, we can plug back in to find the quadratic approximation of $c(\vec{w})$ near $\vec{w} = \vec{w}_*$. Using our result from the previous section,

$$\begin{aligned} c(\vec{w}_* + \vec{\delta w}) &= c(\vec{w}_*) + \begin{bmatrix} \frac{\partial c}{\partial w[1]} & \cdots & \frac{\partial c}{\partial w[n]} \end{bmatrix} (\vec{\delta w}) + \frac{1}{2} (\vec{\delta w})^\top H (\vec{\delta w}) \\ &= c(\vec{w}_*) + \sum_{g=1}^{n} \sum_{i=1}^{m} c_i'(\vec{w}_*^\top \vec{x}_i) x_i[g] (\vec{\delta w})[g] + \frac{1}{2} (\vec{\delta w})^\top \sum_{i=1}^{m} c_i''(\vec{w}_*^\top \vec{x}_i) \vec{x}_i \vec{x}_i^\top (\vec{\delta w}) \\ &= c(\vec{w}_*) + \sum_{i=1}^{m} c_i'(\vec{w}_*^\top \vec{x}_i) \sum_{g=1}^{n} x_i[g] (\vec{\delta w})[g] + \frac{1}{2} \sum_{i=1}^{m} c_i''(\vec{w}_*^\top \vec{x}_i) (\vec{\delta w})^\top \vec{x}_i \vec{x}_i^\top (\vec{\delta w}) \\ &= c(\vec{w}_*) + \sum_{i=1}^{m} \left[c_i'(\vec{w}_*^\top \vec{x}_i) \vec{x}_i^\top (\vec{\delta w}) + \frac{1}{2} c_i''(\vec{w}_*^\top \vec{x}_i) (\vec{\delta w})^\top \vec{x}_i \vec{x}_i^\top (\vec{\delta w}) \right]. \end{aligned}$$

This doesn't look particularly meaningful yet. Let's see what we can do to simplify it. First, notice that the leading term $c(\vec{w}_*)$ does not depend on $\vec{\delta w}$. Since we are interested only in the value of $\vec{\delta w}$ that minimizes the overall cost, we can therefore ignore the leading term since it does not affect the minimizing value of $\vec{\delta w}$. Furthermore, observe that

$$(\vec{\delta w})^\top \vec{x}_i \vec{x}_i^\top (\vec{\delta w}) = ((\vec{\delta w})^\top \vec{x}_i) \times (\vec{x}_i^\top (\vec{\delta w})) = (\vec{x}_i^\top (\vec{\delta w}))^2,$$

since the two terms in the product are both simply the same scalar. Thus, we can rewrite the quantity we are

trying to minimize as

$$\sum_{i=1}^{m}\left[c_i'(\vec{w}_*^\top\vec{x}_i)(\vec{x}_i^\top(\vec{\delta w}))+\frac{1}{2}c_i''(\vec{w}_*^\top\vec{x}_i)(\vec{x}_i^\top(\vec{\delta w}))^2\right].$$

Observe that $\vec{x}_i^\top(\vec{\delta w})$ is just a scalar. Recall that when working with learning linear classifiers using least-squares, the expression $\vec{x}_i^\top\vec{w}$ came up a lot inside summations over i. Furthermore, our above expression to minimize now only has linear and quadratic terms in that same expression. Thus, it is natural to try and fit it into the form of the least squares cost function, using $\vec{\delta w}$ as the vector of weights that we want to learn.

To do so, we should first ask ourselves: what is our goal? What does the least squares cost function look like? Since we want to minimize the sum of the squared residuals, the least squares cost function looks like

$$\sum_{i=1}^{m}(b_i-\vec{a}_i^\top(\vec{\delta w}))^2,$$

where our points are the $\vec{a}_i$, the values we are trying to model are the b_i, and the parameters of our least squares model are in the vector $\vec{\delta w}$.

So we need to get a square of the form $(b_i-a_i^\top(\vec{\delta w}))^2$ inside our summation somehow, given linear and quadratic terms in $\vec{x}_i^\top(\vec{\delta w})$. You should recall that you know a technique that can do exactly this — completing the square! This was something that you saw when you leaned the quadratic formula to find the roots of a quadratic equation — the goal is to write a quadratic[5] as a perfect square plus a constant. When you learned the quadratic formula, the goal was to say that once an expression is that form, the roots are obvious. You just have to set the perfect square equal to the negative of the constant, and so you'll get two roots.

Completing the square of the summands, we see that

$$c_i'(\vec{w}_*^\top\vec{x}_i)(\vec{x}_i^\top(\vec{\delta w}))+\frac{1}{2}c_i''(\vec{w}_*^\top\vec{x}_i)(\vec{x}_i^\top(\vec{\delta w}))^2=\frac{c_i''(\vec{w}_*^\top\vec{x}_i)}{2}\left(\vec{x}_i^\top(\vec{\delta w})+\frac{c_i'(\vec{w}_*^\top\vec{x}_i)}{c_i''(\vec{w}_*^\top\vec{x}_i)}\right)^2-\frac{(c_i'(\vec{w}_*^\top\vec{x}_i)^2)}{2\times c_i''(\vec{w}_*^\top\vec{x}_i)}.$$

Notice again that the trailing additive constant does not depend on $\vec{\delta w}$, so it can be disregarded when trying to minimize. Similarly, we can drop the leading factor of $1/2$ in the squared term. Doing so, and then plugging this new summand back into the overall expression to be minimized, we are ultimately interested in computing

$$\underset{\vec{\delta w}}{\operatorname{argmin}}\left[\sum_{i=1}^{m}c_i''(\vec{w}_*^\top\vec{x}_i)\left(\vec{x}_i^\top(\vec{\delta w})+\frac{c_i'(\vec{w}_*^\top\vec{x}_i)}{c_i''(\vec{w}_*^\top\vec{x}_i)}\right)^2\right].$$

We're very nearly there! The only issue is the weights in front of each summand. Their presence is why the technique we are developing is commonly known as *iteratively reweighted least squares*, because these weights are updated at each iteration of the algorithm, while the points themselves remain the same $\vec{x}_i$.

However, we do not yet know how to solve a weighted least squares problem, we only know how to solve the regular unweighted case. So how can we get rid of these weights? The key is to recognize that, in both the quadratic and exponential cases, the second derivative of our cost function is always positive. Thus, we

[5] Explicitly: $ax^2+bx+c=a(x^2+\frac{b}{a}x+\frac{c}{a})=a((x+\frac{b}{2a})^2+\frac{c}{a}-\frac{b^2}{4a^2})=a((x+\frac{b}{2a})^2-\frac{b^2-4ac}{4a^2})$. This has roots whenever $(x+\frac{b}{2a})^2=\frac{b^2-4ac}{4a^2}$. In other words, $x+\frac{b}{2a}=\pm\frac{\sqrt{b^2-4ac}}{2a}$, or put into traditional form: $x=\frac{-b\pm\sqrt{b^2-4ac}}{2a}$.

can simply take the square root of the weight inside our squared term, to obtain

$$\underset{\vec{\delta w}}{\operatorname{argmin}} \left[\sum_{i=1}^{m} \left(\sqrt{c_i''(\vec{w}_*^\top \vec{x}_i)} \vec{x}_i^\top (\vec{\delta w}) - \frac{-c_i'(\vec{w}_*^\top \vec{x}_i)}{\sqrt{c_i''(\vec{w}_*^\top \vec{x}_i)}} \right)^2 \right].$$

Notice the double minus sign in the scalar part of the squared term. This is simply to more clearly put our expression into the standard form of the least squares cost function.

Expressed in words, at each iteration, we set up a least squares problem with the m point, value pairs:

$$\left(\sqrt{c_i''(\vec{w}_*^\top \vec{x}_i)} \vec{x}_i^\top, -\frac{c_i'(\vec{w}_*^\top \vec{x}_i)}{\sqrt{c_i''(\vec{w}_*^\top \vec{x}_i)}} \right)$$

for $i \in \{1, 2, \ldots, m\}$.

We put the coefficients of the solution to this least squares problem in a vector $\vec{\delta w}$, and update our weights to be $\vec{w} = \vec{w}_* + \vec{\delta w}$. We make these weights our new $\vec{w}_*$ and repeat the procedure until we converge on a solution.

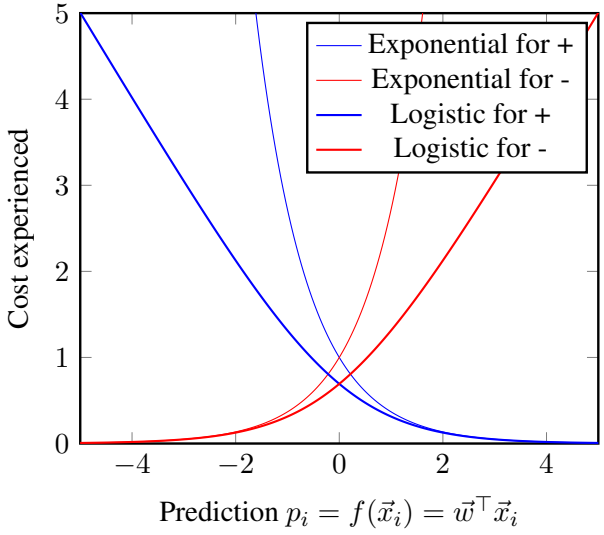
We have not proven that this process will always converge, but it can be shown in later classes that this is in fact the case for the examples given so far — the key is that the second derivative is always positive. Taking it as given that it converges on some $\vec{w}^*$, we can solve for $(\vec{w}^*)^\top \vec{x} = 0$ to see what our decision boundary is for classification.

6 Choosing an even better cost function: deriving logistic regression

At this point, we actually have a very powerful generalization of least-squares at our disposal. We can use this iteratively reweighted procedure on pretty much any cost function whose second derivative is always positive. In the homework, you will explore this freedom and see that there is one more natural modification that we might want to make to our cost function. The issue is that the exponential cost function obsesses too much about points that are very largely misclassified. However, in many real contexts, we have some mislabeled points in our training data itself. It doesn't make sense to let those mislabeled points dominate the optimization. So the question is how can we mellow-out the exploding part of the exponential cost function without removing the very nice part. Explicitly, we like the part of e^{-p} that is decaying to zero, but don't like the exploding behavior on the negative p side.

The natural way to mellow out an exponential is to take a logarithm, but simply taking a logarithm of the exponential e^{-p} would give rise to $-p$ which is also not desirable because this cost can be made arbitrarily negative which doesn't make sense. A single correctly classified point shouldn't give unbounded happiness to the optimization. So what can be done? It turns out[6] that $\ln(1+x)$ has interesting behavior. For numbers x that are close to zero, $\ln(1+x) \approx x$. Meanwhile, if we consider a very big number x, then $\ln(1+x) \approx \ln(x)$ since $\ln(1+x) - \ln(x) = \ln\left(\frac{1+x}{x}\right) = \ln\left(1+\frac{1}{x}\right) \to 0$. So, this suggests using the cost function $c_i(p) = \ln\left(1+e^{-\ell_i p}\right)$.

[6] As the homework shows, this observation is natural once we consider bringing approximation thinking to Bode plots. But here, we will simply invoke this observation.



We can see the much mellower behavior of the logistic cost function in the plots above as compared to the exponential cost. It turns out that the almost linear behavior of the logistic cost functions in the misclassified region results in much better native resilience to outliers. A few mislabeled and crazily placed points just can't impact the chosen $\vec{w}$ that much. However the deeper reasons for that will have to wait for future courses like 127.

There are also deeper connections between probability and logistic regression, even though we have shown here that binary logistic regression can be naturally discovered for yourself on optimization-related grounds alone. Those connections too will have to wait for later courses.

7 Generalizing beyond binary classification (Bonus)

The ideas here can naturally generalize beyond the binary case to the maximum-correlation style perspective that you saw in 16A. The goal becomes to learn templates $\vec{w}_k$ corresponding to each of the K classes so that any given input $\vec{x}$ can be classified by computing the maximum correlation $\arg\max_k \vec{w}_k^\top \vec{x}$. Following the design pattern we have seen above, the key is choosing a loss function to optimize. The straightforward approach of doing binary least squares (or exponential or logistic) regression can be used to learn each of the $\vec{w}_k$ individually by treating the binary problem as "class k or not class k." But this ignores the fact that the final decision will involve taking a maximum.

Whereas logistic regression for the binary case emerges naturally, how to generalize it beyond the binary case is a bit out of the scope of 16B because many of the most natural ways to come up with a solution involve looking at the problem with the strategic perspective that probability provides, and probability is a topic for our successor course 70 (and realistically, for this intuition, you also need 126). However, it is possible to come up with a natural cost function that takes all the "predictions" $\vec{p}_i = \begin{bmatrix} \vec{w}_1^\top \vec{x}_i \\ \vec{w}_2^\top \vec{x}_i \\ \vdots \\ \vec{w}_K^\top \vec{x}_i \end{bmatrix} = \begin{bmatrix} \vec{w}_1^\top \\ \vec{w}_2^\top \\ \vdots \\ \vec{w}_K^\top \end{bmatrix} \vec{x}_i$ in at once[7]. Let us slightly refresh the notation to be more friendly to this case. Let the label ℓ_i for the i-th

[7] Notice how here, the "predictions" are just a bank of correlations of the data $\vec{x}_i$ with the various templates $\vec{w}_k$. This should be reminiscent of what you saw in 16A in Module 3.

training data point $\vec{x}_i$ be a natural number from $1, 2, \ldots, K$. What we want is a cost function $c^{\ell}(\vec{p})$ that gives us a cost when the true label is ℓ and the vector of real predictions is $\vec{p}$. The desiderata are that the cost should be very low when $p[\ell]$ is actually the maximum with a clear advantage, and that the cost should be relatively higher when $p[\ell]$ is not the maximum. We also would like to essentially recover the case of logistic regression when $K = 2$.

Here, it is good to first wonder what the *two* predictions $p[1]$ and $p[2]$ should be for the case of binary logistic regression since earlier, we only had a single weight vector $\vec{w}$. If we consider the first position $p[1]$ to correspond to the score of the "-" class and the second position $p[2]$ to correspond to the score of the "+" class, then it is natural to split the difference and assign $\vec{w}_1 = -\frac{1}{2}\vec{w}$ and $\vec{w}_2 = +\frac{1}{2}\vec{w}$. This way, if $\vec{w}^{\top}\vec{x}$ is positive, then $p[2]$ is going to be positive while $p[1]$ is negative — this means that $p[2]$ will win[8] the argmax and we will declare the second class, the "+" class, victorious in this case. Meanwhile, if $\vec{w}^{\top}\vec{x}$ is negative, then $p[1]$ is going to be positive while $p[2]$ is negative — this means that $p[1]$ will win the argmax and we will declare the first class, the "-" class, victorious in this case.

So, what does the logistic loss look like? For something that was labeled "+" to begin with, the logistic loss was $\ln\left(1 + e^{-p}\right)$ which can be rewritten to involve $p[1]$ and $p[2]$ by observing:

$$\begin{aligned}
\ln\left(1 + e^{-p}\right) &= \ln\left(1 + e^{p[1]-p[2]}\right) \\
&= \ln\left(\frac{e^{p[2]} + e^{p[1]}}{e^{p[2]}}\right) \\
&= \ln\left(e^{p[2]} + e^{p[1]}\right) - \ln\left(e^{p[2]}\right) \\
&= \ln\left(e^{p[2]} + e^{p[1]}\right) - p[2]
\end{aligned}$$

where we were driven to try and make the expression as symmetric in the different $p[k]$ as possible. To verify the symmetry of the pattern, let us look at the case of something that was labeled "-."

$$\begin{aligned}
\ln(1 + e^{p}) &= \ln\left(1 + e^{p[2]-p[1]}\right) \\
&= \ln\left(\frac{e^{p[2]} + e^{p[1]}}{e^{p[1]}}\right) \\
&= \ln\left(e^{p[2]} + e^{p[1]}\right) - p[1].
\end{aligned}$$

This immediately suggests a natural extrapolation/generalization to the case of K classes:

$$c^{\ell}(\vec{p}) = \ln\left(\sum_{k} e^{p[k]}\right) - p[\ell]. \tag{4}$$

The intuition behind this is that the sum of exponentials is dominated by the largest term. So taking a log of a sum of exponentials is tantamount to taking a softer version of the max. We would like the loss to be small when the max is indeed the desired one, and the loss function we have crafted certainly satisfies that. At the same time, when we are getting the answer wrong, we want the loss to be positive and since

[8] Notice that there is a slight difference between the maximum cross-correlation approach we are proposing here and what was done in 16A. In 16A, because of the nature of the problem, we chose to look at the maximum absolute value of the cross-correlation. Here, we are not taking the absolute value and are considering the correlation as a signed quantity.

$\ln\left(\sum_k e^{p[k]}\right) \geq \max(p[1], \ldots, p[K]) \geq p[\ell]$ this is certainly also true.

The iteratively reweighted least-squares algorithm described above can be adapted[9] for this multi-class logistic loss function treating all the $\vec{w}_k$ together as one big parameter vector to be learned. Once again, we will get a sequence of least-squares-style problems to solve, and converge to a set of learned $\vec{w}_k$.

This is sometimes called multinomial logistic regression (as well as minimizing cross-entropy loss) and is one of the workhorse building blocks for modern neural-net based approaches to classification. You will learn more about this in later courses.

What is important to take away from this lesson is that understanding classification in machine learning draws on the same underlying family of ideas involving linearization and approximation thinking that we need to use for control, differential equations, or nonlinear transistor circuits[10]. The differences are only skin deep, while the commonalities in thinking and technical tools go down to the bone.

Contributors:

- Rahul Arya.
- Anant Sahai.
- Gaoyue Zhou.

[9]Remember, the heart of the iterative approach is to approximate the total loss in the neighborhood of existing set of weights by a quadratic, and then minimizing the quadratic to get an update. In general, such forms $\frac{1}{2}\vec{u}^\top H\vec{u} + \vec{q}^\top\vec{u}$ are minimized by $\vec{u} = H^{-1}\vec{q}$ — this can be seen by an appropriate change of coordinates and completing the square. The pseudo-inverse should be used here in place of the inverse in case H has a nullspace, as you've seen in the HW.

In our case, we stack all the $\vec{w}_k$ on top of each other to get a Kn-long parameter vector $\vec{w}$. Because the total loss is the sum of individual losses, you just have to do the quadratic approximation for each of those individual losses and then add them up. This gives rise to a quadratic approximation of the form $\frac{1}{2}\delta\vec{w}^\top(\sum_i H_i)\delta\vec{w} + (\sum_i \vec{q}_i^\top)\delta\vec{w} +$ stuff. In general, the standard scalar chain rule implies that this is going to involve computing $\frac{\partial}{\partial \vec{w}_k}c^{\ell_i}(\vec{p}_i) = \frac{\partial}{\partial p[k]}c^{\ell_i}(\vec{p}_i)\vec{x}_i^\top$. Basically, the different $\vec{w}_k$ experience different scalar derivatives $\frac{\partial}{\partial p[k]}c^{\ell_i}(\vec{p}_i)$ instead of a single c'. When the derivative is viewed as a Kn-long row vector, notice how there is a pattern to this: each of the K blocks is a multiple of $\vec{x}_i^\top$.

For the second derivative, we are going to get different terms that cross across the different weights $\vec{w}_k$. $\frac{\partial^2}{\partial w_j[h]\partial w_k[g]}c^{\ell_i}(\vec{p}_i) = \frac{\partial}{\partial w_j[h]}(\frac{\partial}{\partial p[k]}c^{\ell_i}(\vec{p}_i)x_i[g]) = (\frac{\partial^2}{\partial p[j]\partial p[k]}c^{\ell_i}(\vec{p}_i))x_i[h]x_i[g]$. This will in general result in a Hessian matrix that is made of $K \times K$ blocks, each of which are $n \times n$ in size and of the form $(\frac{\partial^2}{\partial p[j]\partial p[k]}c^{\ell_i}(\vec{p}_i))\vec{x}_i\vec{x}_i^\top$. Notice that each of these is a scalar $(\frac{\partial^2}{\partial p[j]\partial p[k]}c^{\ell_i}(\vec{p}_i))$ times the same matrix $\vec{x}_i\vec{x}_i^\top$.

This kind of scaled copy-paste for matrices happens often and so is usually denoted using what's called the Kronecker (or tensor) product $\otimes$ so that $H_i = \begin{bmatrix} \frac{\partial^2}{\partial p[1]\partial p[1]}c^{\ell_i}(\vec{p}_i) & \frac{\partial^2}{\partial p[2]\partial p[1]}c^{\ell_i}(\vec{p}_i) & \cdots & \frac{\partial^2}{\partial p[K]\partial p[1]}c^{\ell_i}(\vec{p}_i) \\ \frac{\partial^2}{\partial p[1]\partial p[2]}c^{\ell_i}(\vec{p}_i) & \frac{\partial^2}{\partial p[2]\partial p[2]}c^{\ell_i}(\vec{p}_i) & \cdots & \frac{\partial^2}{\partial p[K]\partial p[2]}c^{\ell_i}(\vec{p}_i) \\ \vdots & \ddots & \ddots & \vdots \\ \frac{\partial^2}{\partial p[1]\partial p[K]}c^{\ell_i}(\vec{p}_i) & \frac{\partial^2}{\partial p[2]\partial p[K]}c^{\ell_i}(\vec{p}_i) & \cdots & \frac{\partial^2}{\partial p[K]\partial p[K]}c^{\ell_i}(\vec{p}_i) \end{bmatrix} \otimes \vec{x}_i\vec{x}_i^\top$. Meanwhile, the derivative also has a representation using the Kronecker product: $\vec{q}_i^\top = \begin{bmatrix} \frac{\partial}{\partial p[1]}c^{\ell_i}(\vec{p}_i) & \frac{\partial}{\partial p[2]}c^{\ell_i}(\vec{p}_i) & \cdots & \frac{\partial}{\partial p[K]}c^{\ell_i}(\vec{p}_i) \end{bmatrix} \otimes \vec{x}_i^\top$.

For (4) in particular, the key distinction is that the partial derivative with respect to $p[k]$ is different depending on whether $\ell = k$ or whether $\ell \neq k$. When $\ell = k$, we pick up an additional -1 term in this derivative, which ends up resulting in another $-\vec{x}_i^\top$ term in the derivative with respect to $\vec{w}_k$. However, this additional term contributes nothing to the second derivative of (4) with respect to any of the $\vec{w}_k$ since it is effectively a constant. As a result, the second derivative doesn't care if $\ell = k$ or not. This is a computationally nice feature of this particular loss function.

[10]You'll see a HW problem where you use linearization to better understand the origins of gain in circuits.

