Linear Classification with Perceptrons

Ryan P. Adams
COS 324 – Elements of Machine Learning
Princeton University

Our focus so far has been on regression, in which we are trying to predict a real-valued label, but we now turn our attention to the archetypical supervised machine learning problem of classification. In classification, our hypotheses take features in some input space $\mathcal{X}$ and map them into a categorical label space like $\mathcal{Y} = \{\text{cat}, \text{dog}, \text{cow}, \ldots\}$. Our data are sets of input/output tuples as before $\{\boldsymbol{x}_n, y_n\}_{n=1}^N$, with $\boldsymbol{x}_n \in \mathcal{X}$ and $y_n \in \mathcal{Y}$. We will focus here on binary classification, in which there are only two output categories and so we can treat $\mathcal{Y}$ as $\{0, 1\}$ or $\{-1, +1\}$ without loss of generality. In a future note we will talk about how to handle more than two output categories.

Whereas in regression we would talk about fitting a line, in classification we talk about fitting a *decision boundary*. The decision boundary is the hypersurface that partitions the input space $\mathcal{X}$ into the two classes. Figure 1 shows a simple classification problem of determining whether a fruit is an orange or a lemon based on width and height. One possible decision boundary is shown. There are a variety of ways to think about how to specify and model a decision boundary, but the most common way is to construct a function $f : \mathcal{X} \to \mathbb{R}$ and then consider the surface defined by $f(\boldsymbol{x}) = 0$ to be the decision boundary. That is, when $f(\boldsymbol{x}) \geq 0$ we predict class 1 and when $f(\boldsymbol{x}) < 0$ we predict class 0. This also perhaps make it clear why we sometimes make the classes $+1$ and -1 because then the classification becomes $\text{sgn}(f(\boldsymbol{x}))$.

Classification also has a different notion of error. Whereas in regression it seemed like there were many ways to reason about what it means to make an error, with squared error being a particularly convenient one, in classification we almost always care about *accuracy*. So when we think about loss functions for evaluating our models, it is very common to use *zero-one loss*: $\ell(\hat{y}, y) = \mathbf{1}_{\hat{y} \neq y}$. This function is one if you get it wrong and zero if you get it right. In general, however, fitting this loss directly is difficult because it is not differentiable. We usually come up with different loss functions for actually training our models and we hope these training losses relate closely to accuracy; over the next several lectures we will look at several variations.

Here we will focus on the case where we can take $\mathcal{X}$ to be a D-dimensional real space with a constant feature rolled in, as we did in linear regression. We will then model $f(\boldsymbol{x})$ as a linear function, i.e., $f(\boldsymbol{x}) = \boldsymbol{w}^\mathsf{T}\boldsymbol{x}$. In this setup, the decision boundary is a hyperplane defined by $\boldsymbol{w}^\mathsf{T}\boldsymbol{x} = 0$. You can see here that $\boldsymbol{w}$ is proportional to the unit normal vector for the hyperplane. Points in $\mathcal{X}$ are only on the decision boundary if they are orthogonal to $\boldsymbol{w}$. This classifier is based on a simple rule: whether the test point has positive or negative inner product with $\boldsymbol{w}$. So, how do we fit $\boldsymbol{w}$ to data?

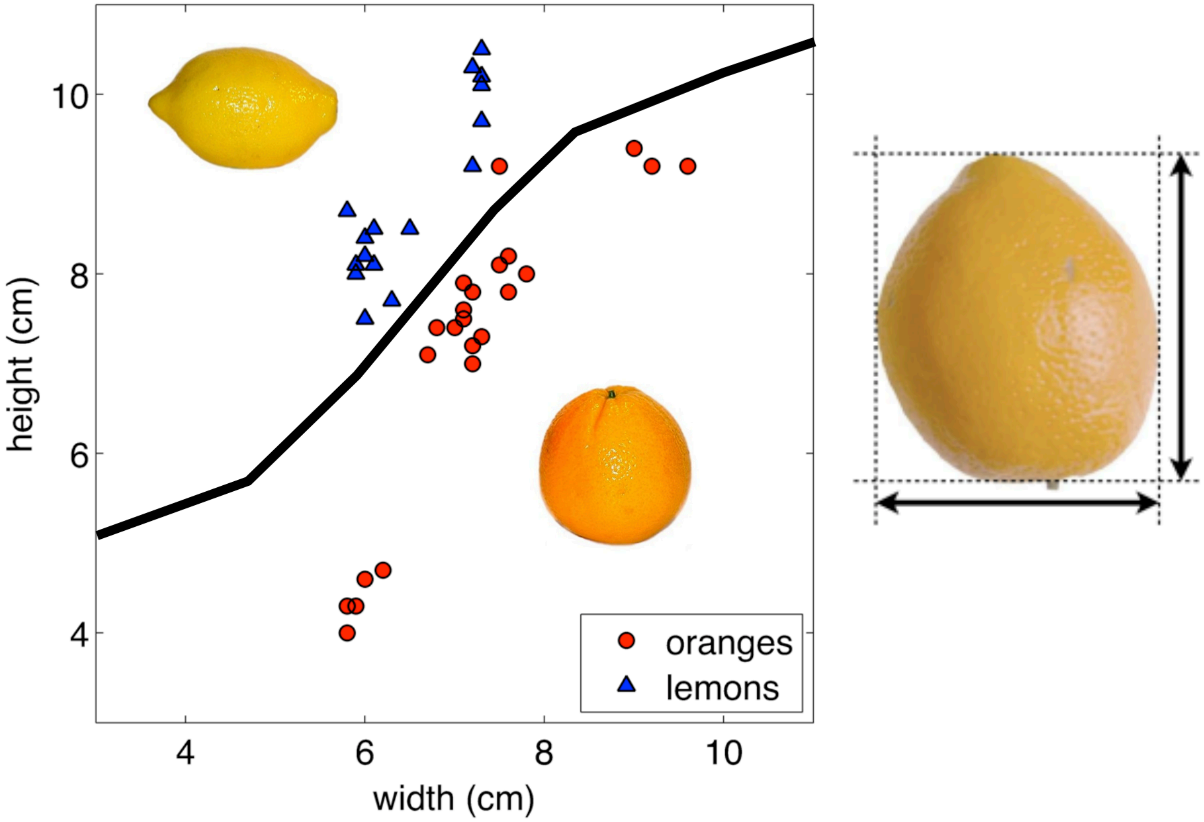
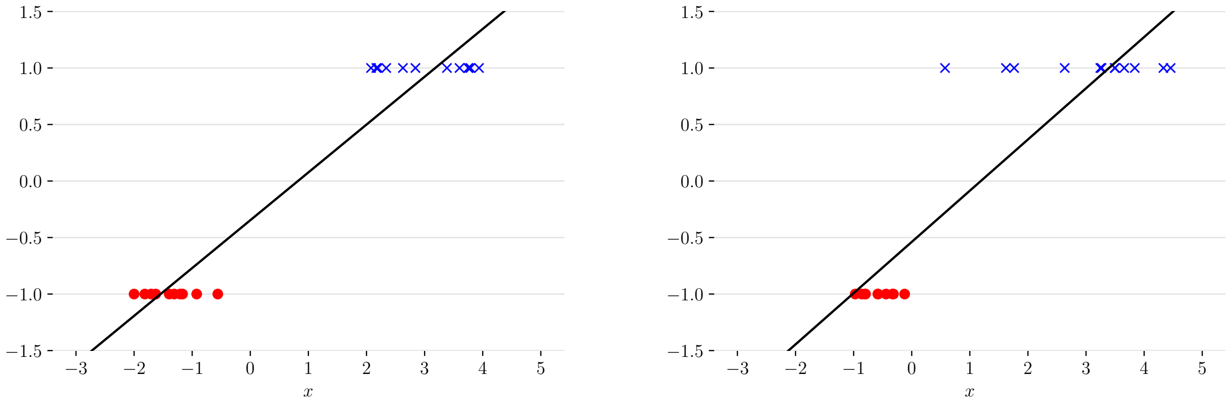


Figure 1: An example of a decision boundary between some measurements of oranges and lemons. These data and figures were created by Iain Murray.

Binary Classification with Least Squares The first thing you might think of when coming up with a way to fit $\boldsymbol{w}$ is to treat it like a regression problem. This is a perfectly valid thing to do: you could take the labels to be $\{-1, +1\}$ and create the design matrices $\boldsymbol{X}$ and $\boldsymbol{y}$ as in regression, but with $\boldsymbol{y}$ being a stack of -1 and $+1$ labels. Then you could find the OLS estimate of $\boldsymbol{w}$ as

$$\boldsymbol{w}^\star = (\boldsymbol{X}^\top \boldsymbol{X})^{-1} \boldsymbol{X}^\top \boldsymbol{y} \,. \tag{1}$$

For many binary classification problems, this is an easy thing to do and the closed-form solution is appealing. However, when you use least-squares for classification, you're ignoring important geometry of the problem that can lead to poor performance. Figure 2a shows a situation where least squares does a reasonable job and finds a sensible decision boundary (the point where the line crosses zero). However, Figure 2b shows one way in which this can go wrong, where the problem should be easy (linearly separable data) but data away from the decision boundary—which should have little effect on the solution—are pulling the decision boundary into a bad place.



(a) Least squares finds a reasonable decision boundary

(b) Least squares finds a poor decision boundary even though the data are linearly separable.

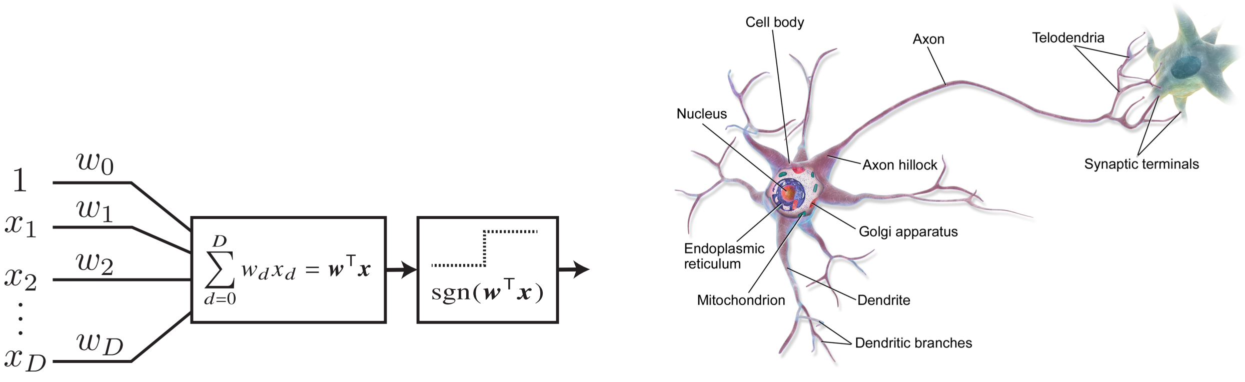
Figure 2: Illustration of how least squares classification can produce an undesirable decision boundary even on an easy problem.

The Perceptron

The perceptron was one of the first neurally-motivated classification models. It's a fancy name to describe the very simple model from above that has a linear weighting of the inputs and a threshold. Figure 3a shows the basic abstraction and perhaps provides some insight into why some people think of this model as relating to biological neurons (Figure 3b). A neuron has dendrites that collect inputs, weighting them with the strength of their synapses. When the dendrites aggregate enough activity to push the cross-membrane voltage over a threshold, then there is an action potential — the neuron "fires" or "spikes" — and it briefly flips its state.

The weights of a perceptron can be learned using a simple algorithm developed by Frank Rosenblatt in 1957. This algorithm depends critically on the data $\{\boldsymbol{x}_n, y_n\}_{n=1}^{N}$ being *linearly separable*. That is, there must exist some $\boldsymbol{w}$ such that the data are perfectly classified by a plane $\boldsymbol{w}^{\mathsf{T}}\boldsymbol{x} = 0$, where we're assuming a constant feature as before.

The *perceptron learning rule* loops and within each loop it iterates over the data one at a time. We're going to use $\{-1, +1\}$ for the labels. As each datum is examined, there are two possible cases: either $y_n \cdot \boldsymbol{w}^{\mathsf{T}}\boldsymbol{x}_n \geq 0$ or $y_n \cdot \boldsymbol{w}^{\mathsf{T}}\boldsymbol{x}_n < 0$. In in the former case, we're classifying datum n correctly, and in the latter case we're getting it wrong. Note that we're multiplying the inner product by the true label to take advantage of the fact that being correct means having the same sign. If we are getting this example right, then there's no reason to update $\boldsymbol{w}$. There are two ways to get it wrong, however. What should we do in each case? If $\text{sgn}(\boldsymbol{w}^{\mathsf{T}}\boldsymbol{x}_n) = +1$ when $y_n = -1$, then we want the inner product to go down. If $\text{sgn}(\boldsymbol{w}^{\mathsf{T}}\boldsymbol{x}_n) = -1$ when $y_n = +1$, then we want the inner product to go up. Of course, to make this inner product go up or down, we need to know about the sign of each entry in $\boldsymbol{x}_n$. For example, if we want $\boldsymbol{w}^{\mathsf{T}}\boldsymbol{x}_n$ to get bigger, and $x_{n,d} < 0$, then we need w_d to get smaller. So at iteration t if we're looking at example n, then the update rule for dimension d of the



(a) Perceptron

(b) Sketch of biological neuron. (Credit: Wikipedia)

Figure 3: Illustraton of perceptron and biological neuron

weights is:

$$w_d^{(t+1)} \leftarrow w_d^{(t)} + \alpha(y_n - \mathrm{sgn}(\boldsymbol{w}^\top \boldsymbol{x}_n))x_{n,d} \,. \tag{2}$$

Here $\alpha > 0$ is a small constant called a *learning rate*. We could write this as a vector update as well:

$$\boldsymbol{w}^{(t+1)} \leftarrow \boldsymbol{w}^{(t)} + \alpha(y_n - \mathrm{sgn}(\boldsymbol{w}^\top \boldsymbol{x}_n)) \cdot \boldsymbol{x}_n \,. \tag{3}$$

Intuitively, this is just trying to nudge the weights in the right direction whenever there is an error, and leave them alone when there isn't an error.

The perceptron learning rule can also be applied using a "batch" of M data within each iteration:

$$w_d^{(t+1)} \leftarrow w_d^{(t)} + \alpha \sum_{m=1}^{M} (y_m - \mathrm{sgn}(\boldsymbol{w}^\top \boldsymbol{x}_m))x_{m,d} \tag{4}$$

or as a vector update:

$$\boldsymbol{w}^{(t+1)} \leftarrow \boldsymbol{w}^{(t)} + \alpha \sum_{m=1}^{M} (y_m - \mathrm{sgn}(\boldsymbol{w}^\top \boldsymbol{x}_m)) \cdot \boldsymbol{x}_m \,. \tag{5}$$

With a small enough learning rate α and linearly separable data, the perceptron learning rule will eventually converge to a solution that classifies all of the training data correctly. It is typical in this setting to decay the learning rate as a function of iteration using something roughly $\alpha_t = O(1/t)$. If the data are not linearly separable, the perceptron learning rule may never converge. Decaying the learning rate will help find a reasonable solution, but it can be slow and noisy.

At about the same time that Rosenblatt introduced the perceptron learning rule, Widrow and Hoff introduced an alternative approach. This rule — called *adaline* for "adaptive linear element"

— converges to low error solutions even when the data are not linearly separable. The adaline rule tries to resolve the non-differentiability of the sgn$(\cdot)$ function by computing the perceptron update rule with $\boldsymbol{w}^\mathsf{T}\boldsymbol{x}_n$ instead of $\mathrm{sgn}(\boldsymbol{w}^\mathsf{T}\boldsymbol{x}_n)$ in Eqs. 2 and 3. This turns out to be exactly the same thing we would do with least squares regression if we tried to solve it with gradient descent rather than solving the linear system directly. To see this, imagine that we looked at the loss function of a single example in OLS and took its gradient:

$$L_n(\boldsymbol{w}) = (\boldsymbol{w}^\mathsf{T}\boldsymbol{x}_n - y_n)^2 \qquad\qquad \nabla_{\boldsymbol{w}} L_n(\boldsymbol{w}) = 2(\boldsymbol{w}^\mathsf{T}\boldsymbol{x}_n - y_n)\boldsymbol{x}_n \,. \tag{6}$$

We would want to go downhill, so in each iteration we could take steps scaled by α in the direction of the negative gradient:

$$\boldsymbol{w}^{(t+1)} \leftarrow \boldsymbol{w}^{(t)} + \alpha 2(y_n - \boldsymbol{w}^\mathsf{T}\boldsymbol{x}_n)\boldsymbol{x}_n \,. \tag{7}$$

Since α is a free parameter, we can just absorb the 2 into it.

Changelog

- 4 October 2018 – Initial version.