Connections between Perceptron and Logistic Regression (and SVM)

This lecture note is intended to expand on the in-class discussion of perceptron, logistic regression, and their similarities. Note that this handles the binary classification case, but the same core similarities underlie the multiclass versions of these algorithms as well.

Preliminaries Following the Eisenstein notation, we have a dataset $\mathcal{D} = \left\{(\mathbf{x}^{(i)}, y^{(i)}\right\}_{i=1}^{N}$ of labeled examples. A feature vector $\mathbf{f}(\mathbf{x}) \in \mathbb{R}^d$ is extracted from each input $\mathbf{x}$. Each $y^{(i)}$ is either +1 or -1, to denote positive and negative examples, respectively. Let $\mathbf{w} \in \mathbb{R}^d$ denote the weight vector to be learned for our classifier.

1 Perceptron

The perceptron algorithm[1] is as follows:

Algorithm 1 Perceptron

Initialize $\mathbf{w} = \mathbf{0}$
for $t = 1$ to $|T|$ **do** ▷ Loop over T epochs, or until convergence (an epoch passes with no update)
  for $i = 1$ to $|N|$ **do** ▷ Loop over N examples
    $y_{\text{pred}} = \text{sign}(\mathbf{w}^\top \mathbf{f}(\mathbf{x}^{(i)}))$ ▷ Make a prediction of +1 or -1 based on the current weights
    $\mathbf{w} \leftarrow \mathbf{w} + \frac{1}{2}\left(y^{(i)} - y_{\text{pred}}\right)\mathbf{f}(\mathbf{x}^{(i)})$ ▷ Update weights if an error was made
  end for
end for
Return $\mathbf{w}$

Note that we have compressed three update cases into one line here. If $y_{\text{pred}} = y^{(i)}$, the example is correct and no update is made. If $y_{\text{pred}} = -1$ and $y^{(i)} = +1$, then we should add $\mathbf{f}(\mathbf{x}^{(i)})$ to $\mathbf{w}$ to make the dot product more positive. If $y_{\text{pred}} = +1$ and $y^{(i)} = -1$, then we should subtract off $\mathbf{f}(\mathbf{x}^{(i)})$ from $\mathbf{w}$. Taking the difference of $y^{(i)}$ and y_{pred} and multiplying by $\frac{1}{2}$ encodes these three cases into a single equation.

2 Logistic Regression

Under logistic regression,

$$P(y = +1|\mathbf{x}) = \frac{\exp\left(\mathbf{w}^\top \mathbf{f}(\mathbf{x})\right)}{1 + \exp\left(\mathbf{w}^\top \mathbf{f}(\mathbf{x})\right)}, \qquad P(y = -1|\mathbf{x}) = \frac{1}{1 + \exp\left(\mathbf{w}^\top \mathbf{f}(\mathbf{x})\right)} \tag{1}$$

The positive class probability is obtained by applying the logistic function to the value $\mathbf{w}^\top \mathbf{f}(\mathbf{x})$, and the negative class probability is one minus the positive probability. When the weight-feature dot product is 0, the positive probability is exactly 0.5 (equal probability of positive and negative class). When this dot product is positive, the exponential term is greater than 1, so the probability is greater than 0.5; conversely, when the dot product is negative, the exponential term is less than 1, so the probability is smaller than 0.5. This equation is a very convenient way to map from the space of reals $(-\infty, \infty)$ to the space of probabilities $(0, 1)$, as shown in Figure 1.

[1] This is a basic version of the algorithm that doesn't change the update step size at all.

Binary logistic regression can be thought of as a special case of multiclass logistic regression where the negative class has no associated features. The multiclass case, discussed in the Eisenstein book, expresses the denominator as a sum over the output space $\mathcal{Y}$ of possible labels. We can view the binary case as a sum over two terms, one of which has a zero feature vector, giving a zero dot product and a 1 when exponentiated.

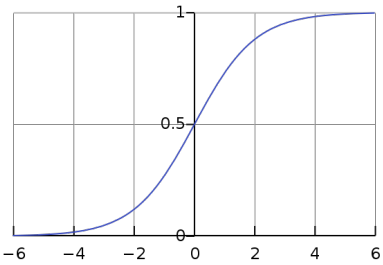

Figure 1: The logistic function

Learning To learn our weights $\mathbf{w}$ from training data, we want to maximize the probability of the observed data. Maximizing the *product* of probabilities over the training set is equivalent to maximizing the *sum* of *log* probabilities, a quantity which is easier to deal with mathematically. Following Eisenstein Equations 2.57-2.58 in the binary case with the notation used in lecture ($\mathbf{w}$ instead of θ), we have:

$$\mathcal{L}\left(\mathbf{x}^{(1:N)}, y^{(1:N)}\right) = \sum_{i=1}^{N} \log P(y^{(i)}|\mathbf{x}^{(i)}) \tag{2}$$

$$= \sum_{i=1}^{N} \begin{cases} \mathbf{w}^\top \mathbf{f}(\mathbf{x}^{(i)}) - \log\left(1 + \exp\left(\mathbf{w}^\top \mathbf{f}(\mathbf{x}^{(i)})\right)\right) & \text{if } y^{(i)} = +1 \text{ (positive)} \\ -\log\left(1 + \exp\left(\mathbf{w}^\top \mathbf{f}(\mathbf{x}^{(i)})\right)\right) & \text{if } y^{(i)} = -1 \text{ (negative)} \end{cases} \tag{3}$$

$$= \sum_{i=1}^{N} \left[\frac{1}{2}\left(1 + y^{(i)}\right)\mathbf{w}^\top \mathbf{f}(\mathbf{x}^{(i)}) - \log\left(1 + \exp\left(\mathbf{w}^\top \mathbf{f}(\mathbf{x}^{(i)})\right)\right)\right] \tag{4}$$

The first line to the second line is achieved by expanding the definition and applying the log. The second line to the last line encodes these two cases in a single equation: note that $1 + y^{(i)}$ is 2 in the positive case and 0 in the negative case.

In order to optimize a continuous, differentiable function like this, we typically use an algorithm like gradient descent/ascent. In this case, we want to *maximize* the log probability, so we want to use gradient *ascent*. The gradient of this function with respect to $\mathbf{w}$ is:

$$\frac{\partial}{\partial \mathbf{w}}\mathcal{L}\left(\mathbf{x}^{(1:N)}, y^{(1:N)}\right) = \sum_{i=1}^{N}\left[\frac{1}{2}(1 + y^{(i)})\mathbf{f}(\mathbf{x}^{(i)}) - \frac{\exp(\mathbf{w}^\top \mathbf{f}(\mathbf{x}^{(i)}))}{1 + \exp(\mathbf{w}^\top \mathbf{f}(\mathbf{x}^{(i)}))}\mathbf{f}(\mathbf{x}^{(i)})\right] \tag{5}$$

$$= \sum_{i=1}^{N}\left[\frac{1}{2}(1 + y^{(i)})\mathbf{f}(\mathbf{x}^{(i)}) - P(y = +1|\mathbf{x}^{(i)})\mathbf{f}(\mathbf{x}^{(i)})\right] \tag{6}$$

$$= \sum_{i=1}^{N}\mathbf{f}(\mathbf{x}^{(i)})\left(\frac{1}{2}(1 + y^{(i)}) - P(y = +1|\mathbf{x}^{(i)})\right) \tag{7}$$

Again, we have compressed two update cases into one. When $y^{(i)} = +1$, we update towards $\mathbf{f}(\mathbf{x}^{(i)})$, but the higher the quantity $P(y = +1|\mathbf{x}^{(i)})$, the less of an update we make. When $y^{(i)} = -1$, we update away from $\mathbf{f}(\mathbf{x}^{(i)})$, and as $P(y = +1|\mathbf{x}^{(i)})$ grows larger, we make more and more of an update. Intuitively, we can think of this as a soft version of the perceptron, where instead of a 1/-1 or 0 as the coefficient for $\mathbf{f}(\mathbf{x}^{(i)})$, we have a real-valued coefficient that depends on how "off" the probability is from the correct value.

3 Connections between Perceptron and Logistic Regression

These two algorithms are motivated from two very different directions. Perceptron is essentially defined by its update rule. It can be proven that, if the data are linearly separable, perceptron is guaranteed to converge; the proof relies on showing that the perceptron makes non-zero (and non-vanishing) progress towards a separating solution on every update. By contrast, logistic regression is instead motivated from a probabilistic perspective, coming from a rich statistical tradition of exponential family models,[2] and the update comes from taking the gradient. This is a convex objective and so gradient-based optimization techniques are guaranteed to converge, but the likelihood can never truly be maximized with a finite $\mathbf{w}$ vector.

However, **we can cast both of these algorithms as minimization of different losses**. For simplicity, we will only discuss losses of a single $(\mathbf{x}, y)$ pair. For both algorithms, we will express the loss in terms of $y\mathbf{w}^\top \mathbf{f}(\mathbf{x})$. This is the standard weight-feature dot product multiplied by the true y. We always want this value to be positive. We can define the perceptron loss as follows:

$$\mathcal{L}_{\text{perc}}(\mathbf{x}, y) = \begin{cases} 0 & \text{if } y\mathbf{w}^\top \mathbf{f}(\mathbf{x}) > 0 \\ -y\mathbf{w}^\top \mathbf{f}(\mathbf{x}) & \text{if } y\mathbf{w}^\top \mathbf{f}(\mathbf{x}) \leq 0 \end{cases} \tag{8}$$

This says that the model incurs 0 loss on correct examples, and otherwise loss is proportional to how "wrong" the classification is.

Differentiating this update with respect to $\mathbf{w}$ yields $-y\mathbf{f}(\mathbf{x})$ when that quantity is negative; this is a constant with respect to a particular training example. However, recall that this value is a loss that we are attempting to minimize, so we want to use gradient *descent* which will involve subtracting the gradient from the weights. We therefore recover the standard update rule: add $\mathbf{f}(\mathbf{x})$ when y (the true label) is positive, and subtract it when y is negative.

Figure 2 shows this perceptron loss plotted graphically. Note that it is zero for $y\mathbf{w}^\top \mathbf{f}(\mathbf{x}) > 0$. It is non-differentiable at the origin, but this is rarely an issue in practice.

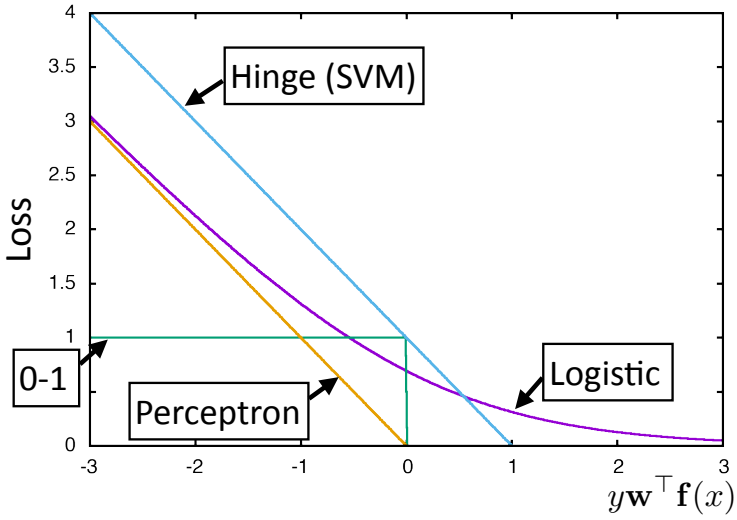

Figure 2: Loss functions for perceptron, logistic regression, and SVM (the hinge loss). 0-1 loss, the "ideal" classification loss, is shown for comparison.

Logistic regression We rewrite Equation 3 and make two changes: we express it using the same quantity $y\mathbf{w}^\top \mathbf{f}(\mathbf{x})$, and we negate the likelihood so it becomes a loss to minimize:

$$\mathcal{L}_{\text{lr}}(\mathbf{x}, y) = \begin{cases} -y\mathbf{w}^\top \mathbf{f}(\mathbf{x}) + \log\left(1 + \exp\left(y\mathbf{w}^\top \mathbf{f}(\mathbf{x})\right)\right) & \text{if } y = +1 \text{ (positive)} \\ \log\left(1 + \exp\left(-y\mathbf{w}^\top \mathbf{f}(\mathbf{x})\right)\right) & \text{if } y = -1 \text{ (negative)} \end{cases}$$

[2] Discussion of these models is outside the scope of this class. However, these models arise naturally from answering the question of what is the distribution with the maximum entropy that fits certain constraints on expectations imposed by the training data.

$-x + \log(1 + \exp(x))$ and $\log(1 + \exp(-x))$ are actually two ways of writing the same function (can you convince yourself of this?), so this loss is actually a single equation when considered in terms of $y\mathbf{w}^\top \mathbf{f}(\mathbf{x})$.

Figure 2 shows this loss in purple. Surprisingly, this doesn't look that different from the perceptron loss! They both asymptote to a line with slope -1 as $y\mathbf{w}^\top \mathbf{f}(\mathbf{x})$ becomes negative and asymptote to $y = 0$ as it becomes positive. The logistic function is smoother and nonzero at the origin, meaning that it prefers examples to be classified correctly by a larger margin. Because it never becomes exactly 0, continuing to train a logistic regression classifier will lead to larger and larger weights.

Another motivation for these two algorithms is that their loss functions are different approximations to the 0-1 loss. The 0-1 loss is not really useful for learning, as its derivative is zero almost everywhere, but it reflects our true objective: minimize the number of errors our classifier makes. Log loss (the logistic regression loss), perceptron loss, and hinge loss are all *surrogate* losses that approximate the 0-1 loss and are easier to optimize.

4 Bonus: SVM

Support vector machines are typically defined as type of optimization problem called a quadratic program (QP): we are minimizing the norm of a weight vector (which is quadratic) subject to the (linear) constraint that every example is classified correctly by a "margin" of 1. Since these constraints are unsatisfiable when the data are inseparable, we introduce a notion of "slack", or a soft penalty that can be paid when an example is misclassified or classified correctly but with too small a margin. Like perceptron, these models are not probabilistic.

Taking the gradient of the SVM objective, we recover an update similar to perceptron: no update is made on examples which are classified correctly, and the *same update as the perceptron* is made when the constraint is violated. The SVM can therefore be defined by a "hinge loss" that is the same as the perceptron loss, but shifted over by 1, reflecting the fact that examples have to be classified correctly by a margin of 1 in order not to incur loss from slack. This leads to much more stable learning in practice compared to the perceptron and is a straightforward extension to implement, as it simply requires changing the criterion for when the update is made.