Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

6.S064 Introduction to Machine Learning

Phase 1: from classification to regression (lecture 3)

Linear classification: non-separable case

Over the last two lectures, we have talked about linear classifiers, i.e., classifiers that can be written as

$$h(x; \theta, \theta_0) = \text{sign}(\, \theta \cdot x + \theta_0 \,) \tag{1}$$

with parameters $\theta \in \mathcal{R}^d$ and $\theta_0 \in \mathcal{R}$. Whenever the training examples $S_n = \{(x^{(t)}, y^{(t)}), t = 1, \ldots, n\}$ are linearly separable, we can use the perceptron algorithm to find parameters that do indeed separate the training examples, i.e., achieve zero training error. In other words, the algorithm finds $\hat{\theta}$, $\hat{\theta}_0$ such that

$$\mathcal{E}_n(\hat{\theta}, \hat{\theta}_0) = \frac{1}{n} \sum_{t=1}^{n} [\![\, y^{(t)}(\hat{\theta} \cdot x^{(t)} + \hat{\theta}_0) \leq 0 \,]\!] = 0 \tag{2}$$

What happens if the training examples are not linearly separable? The perceptron algorithm will not converge, nor finds the classifier with the smallest training error. In fact, the problem of minimizing the training error in the *non-realizable case* is computationally hard. We will try to address these problems here. In order to keep the discussion simpler, we will drop θ_0 as a parameter. It will be easy to add it later.

We will first change how the errors are measured on the training set. What we minimize on the training set is completely in our control. The goal of minimizing the test or generalization error, measured by counting mistakes, remains the same. But it may be helpful to use a slightly different metric on the training set. Loosely speaking, the idea is to give the classifier a little more feedback in terms of how close its predictions are to the training labels. We will call the resulting training error *empirical risk*. It is defined as the average loss on the training examples or

$$R_n(\theta) = \frac{1}{n} \sum_{t=1}^{n} \text{Loss}(\, y^{(t)}(\theta \cdot x^{(t)}) \,) \tag{3}$$

Note that the loss function here takes as an argument the "agreement" $y^{(t)}(\theta \cdot x^{(t)})$ between the linear predictions $\theta \cdot x^{(t)}$ (that we would threshold) and the training labels $y^{(t)}$. If we wanted to go back to the fraction of mistakes, we would simply define

$$\text{Loss}_{0/1}(z) = [\![\, z \leq 0 \,]\!] = \begin{cases} 1, & \text{if } z \leq 0 \\ 0, & \text{o.w.} \end{cases} \tag{4}$$

This is known as the *zero-one loss*. We will explore here a different loss function known as the *hinge loss*:

$$\text{Loss}_h(z) = \max\{1 - z, 0\} \tag{5}$$

The advantage of defining the loss in this way is that it forces the classifier predictions to be more than just correct. In other words, the agreement $z = y^{(t)}(\theta \cdot x^{(t)})$ should be ≥ 1 before we attain zero loss. If we reduce the agreement below 1 (or ≤ 0 for actually misclassified points), the loss increases linearly, and continues to increase the more incorrect our predictions are. In other words, when $y^{(t)}(\theta \cdot x^{(t)}) < 1$, the loss is simply $1 - y^{(t)}(\theta \cdot x^{(t)})$.

It remains to come up with an algorithm that minimizes

$$R_n(\theta) = \frac{1}{n}\sum_{t=1}^{n} \text{Loss}_h(\, y^{(t)}(\theta \cdot x^{(t)})\,) = \frac{1}{n}\sum_{t=1}^{n} \max\big\{\, 1 - y^{(t)}(\theta \cdot x^{(t)}),\, 0\big\} \tag{6}$$

with respect to the parameters θ. The hope here is that by minimizing this empirical risk we actually obtain a better classifier, one that generalizes better. We will address this question later on in the course. Moreover, it should be possible to find $\hat{\theta}$ that minimizes $R_n(\theta)$ even when the examples are not linearly separable. Fortunately, $R_n(\theta)$ is a *convex* function of the parameters[1]. While there may be multiple parameter settings that all attain the minimum, convexity guarantees here that a simple algorithm will be able to find one of them.

(Sub-)gradient descent

We will use gradient descent to minimize $R_n(\theta)$. To this end, note that the gradient

$$\nabla_\theta R_n(\theta) = \left[\frac{\partial R_n(\theta)}{\partial \theta_1}, \ldots, \frac{\partial R_n(\theta)}{\partial \theta_d}\right]^T \tag{7}$$

points in the direction where $R_n(\theta)$ increases. In order to minimize the empirical risk, we should therefore take a small step in the opposite direction. Indeed, the gradient descent algorithm simply iteratively updates the parameters according to

$$\theta^{(k+1)} = \theta^{(k)} - \eta_k \nabla_\theta R_n(\theta)_{|\theta=\theta^{(k)}} \tag{8}$$

where η_k is known as the *step-size* or *learning rate*. We will discuss below how the step-size should be chosen for a stochastic version of this algorithm.

There is one technical issue that we have to address. The function $R_n(\theta)$ is not everywhere differentiable. The hinge loss functions were defined as piece-wise linear

[1] A convex function $f(\theta)$ is any function such that $f(\lambda\theta + (1-\lambda)\theta') \leq \lambda f(\theta) + (1-\lambda)f(\theta')$ for any θ, θ' and $\lambda \in (0, 1)$. Convex functions have a characteristic "bowl" shape.

functions. Adding them together, creating $R_n(\theta)$, also results in a piece-wise linear function. So, there will be "kinks", i.e., points where $R_n(\theta)$ is not differentiable. In fact, at those points, we have several possible gradients and these, collectively, define what is called the *sub-differential*. When the function is differentiable, the sub-differential simply reduces to the typical gradient. For our purposes here, i.e., from the point of view of minimizing $R_n(\theta)$, we only need to select one possible gradient at any point regardless of how many choices there are. This is easy for us to do since the empirical risk is a piece-wise linear function. We can just take any gradient right around a "kink".

Stochastic (sub-)gradient descent

It is often advantageous to use a stochastic version of the gradient descent update. This is true, e.g., when there are many training examples. So, we select a training example at random, and perform a parameter update on the basis of the corresponding loss alone (recall the perceptron rule). If we take small enough steps, these stochastic updates, in aggregate, move the parameters roughly in the same direction as the gradient descent rule. Specifically, the algorithm looks like

$$\begin{array}{l} \theta^{(0)} = 0 \text{ (vector)}, \\ \text{select } t \in \{1, \ldots, n\} \text{ at random}, \\ \quad \theta^{(k+1)} = \theta^{(k)} - \eta_k \nabla_\theta \text{Loss}_h(\, y^{(t)} \theta \cdot x^{(t)} \,)_{|\theta=\theta^{(k)}} \end{array} \tag{9}$$

If the agreement $y^{(t)}(\theta^{(k)} \cdot x^{(t)}) > 1$ then the loss is identically zero and so is the gradient. No update is made in this case. When $y^{(t)}(\theta^{(k)} \cdot x^{(t)}) \leq 1$, on the other hand,

$$\nabla_\theta \text{Loss}_h(\, y^{(t)} \theta \cdot x^{(t)} \,)_{|\theta=\theta^{(k)}} = \nabla_\theta \big(1 - y^{(t)} \theta \cdot x^{(t)} \, \big)_{|\theta=\theta^{(k)}} = -y^{(t)} x^{(t)} \tag{10}$$

We can therefore rewrite the stochastic sub-gradient descent algorithm in a form that looks very much like the perceptron algorithm:

$$\begin{array}{l} \theta^{(0)} = 0 \text{ (vector)}, \\ \text{select } t \in \{1, \ldots, n\} \text{ at random}, \\ \quad \text{if } y^{(t)}(\theta^{(k)} \cdot x^{(t)}) \leq 1, \text{ then} \\ \quad\quad \theta^{(k+1)} = \theta^{(k)} + \eta_k y^{(t)} x^{(t)} \end{array} \tag{11}$$

There are several differences with the perceptron algorithm. First, we use a decreasing learning rate η_k (later updates will be smaller) rather than set it to one. Adjusting the learning rate is essential for the algorithm to converge. Second, the "mistake", i.e., when the update is made, is now defined in terms of the agreement being less than one (as opposed to less than zero for the perceptron). Finally, we select the training example to consider at random rather than cycling through them in order. The random selection prevents the gradient descent updates from oscillating.

So how do we set the learning rate η_k? In our context, $\eta_k = 1/(k+1)$ would ensure that our stochastic sub-gradient descent algorithm converges to the minimum of $R_n(\theta)$. Other choices are possible as well. In fact, any positive learning rate that satisfies

$$\sum_{k=1}^{\infty} \eta_k^2 < \infty, \quad \sum_{k=1}^{\infty} \eta_k = \infty \tag{12}$$

would permit the algorithm to converge though the speed may vary.

For stochastic gradient descent algorithms, it is often quite beneficial to keep track of the best solution obtained so far in addition to the current $\theta^{(k)}$. In other words, we would also keep $\theta^{(i_k)}$, where $i_k = \text{argmin}_{i=1,\ldots,k} R_n(\theta^{(i)})$. The empirical risk of the current solution $\theta^{(k)}$, i.e., $R_n(\theta^{(k)})$, goes down in a noisy fashion while $R_n(\theta^{(i_k)})$ is monotonically decreasing (by definition!). If the algorithm is stopped at any point, we should report $\theta^{(i_k)}$ (the best solution so far) rather than $\theta^{(k)}$. Note that $\lim_{k\to\infty} R_n(\theta^{(i_k)})$ is not necessarily zero as the points may not be linearly separable.

Regression

We have so far discussed only learning problems where we try to predict $y \in \{-1, 1\}$ labels for given feature vectors $x \in \mathcal{R}^d$ (binary classification). Regression problems are exactly like classification problems except that we try to predict real valued responses $y \in \mathcal{R}$. For example, we could try to predict the future value of a property based on a feature representation that includes the size (square footage), location, and so on.

We are looking for functions that map feature vectors $x \in \mathcal{R}^d$ to real valued responses $y \in \mathcal{R}$. In other words, a *regression function* is any $f : \mathcal{R}^d \to \mathcal{R}$, and $f(x)$ specifies the predicted response to x. As in the classification setting, since the number of training examples is finite, we must restrict the *set of functions* $\mathcal{F}$ that we entertain. Permitting all possible regression functions, and selecting one in response to training data, would again be liable to solutions that generalize poorly.

Linear regression

We will start with linear regression. A linear regression function is simply a linear function of the feature vectors: $f(x; \theta, \theta_0) = \theta \cdot x + \theta_0$. Note that a linear classifier differs only in that we threshold the resulting value: $h(x; \theta, \theta_0) = \text{sign}(f(x; \theta, \theta_0))$. Here we are interested in the real value $f(x; \theta, \theta_0)$ directly.

Our learning task is to choose the parameters $\hat{\theta}$ and $\hat{\theta}_0$ in response to a training set $S_n = \{(x^{(t)}, y^{(t)}), t = 1, \ldots, n\}$, where $y^{(t)} \in \mathcal{R}$, such that the resulting function $f(x; \hat{\theta}, \hat{\theta}_0)$ would yield good real valued predictions on yet unseen examples.