CS 520: Supervised Learning on Simple Parametric Models

16:198:520

Instructor: Wes Cowan

1 Formulation and Notation

In supervised learning, we are provided with a set of example data points

$$\text{Data} = \{(\underline{x}_1, \underline{y}_1), (\underline{x}_2, \underline{y}_2), \dots, (\underline{x}_N, \underline{y}_N)\}, \tag{1}$$

where $\underline{x}$ is a sample in some input space X, and the $\underline{y}$ are the output corresponding to the input $\underline{x}$, in some output space Y. We denote by the underscore the fact that the input and corresponding output may be vectors of values. In the case of *regression*, the output $\underline{y}$ generally corresponds to real values, in *classification*, the output $\underline{y}$ usually corresponds to class labels (such as **dog** or **not dog**). The inputs may correspond to anything - for instance, arrays of pixels in images, or sets of weather-related measurements - and the outputs may correspond to anything of interest - for instance class labels for image content, or predictions for weather patterns at some point in the future.

Out goal is to find some function $f : X \mapsto Y$ that accurately models the (input, output) relationship expressed in the data. In general, for some hypothesized f, we can characterize the loss of this model in terms of how well it compares to the desired output $\underline{y}$, summed over each data point:

$$\text{Loss}(f) = \sum_{i=1}^{N} \text{Loss}(f(\underline{x}_i), \underline{y}_i). \tag{2}$$

The goal then is to find, over some function space F, find some map f that minimizes the total loss:

$$f^* = \text{argmin}_{f \in F} \ \text{Loss}(f). \tag{3}$$

In order to facilitate finding this loss-minimizing model, we will frequently take F to be a parameterized family of functions - such as linear, logistic, exponential, etc - so that the total loss is a continuous function of these parameters, and we can try to find the parameters that minimize the loss over all using techniques from calculus.

Common loss functions in the case of regression (taking Y as the real numbers) include the L_1, L_2 errors such as, respectively,

$$\begin{aligned} \text{Loss}(f(\underline{x}), \underline{y}) &= |f(\underline{x}) - y| \\ \text{Loss}(f(\underline{x}), \underline{y}) &= (f(\underline{x}) - y)^2 . \end{aligned} \tag{4}$$

This generalizes naturally to vector space Y, defining the loss in terms of various vector norms.

1.1 Preprocessing

It can frequently be useful to pre-process your data, and structure it in more useful ways. For instance, instead of considering a linear model on inputs of the form $\underline{x} = [x_1, x_2]^{\mathrm{T}}$, we could expand to a quadratic model by considering a linear model on inputs of the form

$$\underline{x}' = [x_1^2, x_1 x_2, x_2^2, x_1, x_2, 1]^{\mathrm{T}}. \tag{5}$$

In general, it can be useful to embed our input space in more interesting feature spaces - in the above case, considering all possible polynomial combinations of the initial bases of degree 2 or fewer. In a more general sense, given a set of feature functions $K_0(\underline{x}), K_1(\underline{x}), \dots, K_m(\underline{x})$, it can be useful to expand to the larger feature space

$$X' = \{[K_0(\underline{x}), K_1(\underline{x}), \dots, K_m(\underline{x})]^{\mathrm{T}} : \underline{x} \in X\}. \tag{6}$$

If the features that you compute capture something interesting and relevant about the data and problem you are interested in solving, training models on the transformed input data may be more fruitful than on the raw data X. This is particularly relevant for things like image processing, when the raw pixel values themselves may not be the most interesting, but computing various features of those pixel values (edge detection, spectral decomposition, etc) may be.

We will generally take $K_0(\underline{x}) = 1$, or assume that the first component of the input data is always 1 - this will simplify a lot of the math to follow.

1.2 Training Models

In general, we will assume that whatever space of functions we are interested in can be parameterized by a vector of parameters $\underline{a}$. We can **train** a model in the following way:

Training a General Model with Gradient Descent:

0) Choose some learning rate $\alpha > 0$.

1) Initialize $\underline{a}_0$ randomly (usually near $\underline{0}$) as an initial guess.

2) Update the parameters based on how poorly the model performs:

$$\underline{a}_{k+1} = \underline{a}_k - \alpha \nabla_{\underline{a}} \text{Loss}(\underline{a}_k). \tag{7}$$

3) Repeat.

The theory here is that for any given model defined by the parameters $\underline{a}_k$, we can try to reduce the total loss (and improve the model) by tweaking the parameters in a direction that reduces the loss - i.e., in the opposite direction of the gradient. This is what's known as **Gradient Descent**. Choosing an appropriate step size may take some experimentation - α too large will often result in overshoot, and poor convergence to a good model. α too small, however, will result in very slow changes in the parameters, and slow convergence to a good model.

Note,

$$\nabla_{\underline{a}} \text{Loss}(\underline{a}) = \sum_{i=1}^{N} \nabla_{\underline{a}} \text{Loss}(f_{\underline{a}}(\underline{x}), \underline{y}). \tag{8}$$

Now, computing the gradient of the loss can be expensive - as it requires iterating over the entire data set, which may be quite large. This motivates the following simplified approach which updates the model based on the loss of a single data point at a time. This is *Stochastic Gradient Descent*.

Training a General Model with Stochastic Gradient Descent:

0) Choose some learning rate $\alpha > 0$.

1) Initialize $\underline{a}_0$ randomly (usually near $\underline{0}$) as an initial guess.

2) Choose some data point $(\underline{x}, \underline{y}) \in$ Data. Update the parameters based on how poorly the model performs:

$$\underline{a}_{k+1} = \underline{a}_k - \alpha \nabla_{\underline{a}} \text{Loss}(f_{\underline{a}_k}(\underline{x}), \underline{y}). \tag{9}$$

3) Repeat.

In both these cases, under some general assumptions on the structure of the loss function (which we will see in more specificity in the following sections), we can guarantee rough notions of convergence to some kind of optimal, loss minimizing $\underline{a}^*$. Generally the convergence of SGD will be worse than true GD because we are only acting based on the error of a specific point in any iteration - but averaged over a long period of time, SGD will usually settle into well-performing models.

2 Linear Regression

In this case, we will take the input space X to be some real vector space (again, taking the 0-component of each vector to be 1), and the output space Y to be the set of real numbers. For linear regression, we take f to be a simple linear combination of the inputs. That is, for some weight vector $\underline{a}$, we have

$$f_{\underline{a}}(\underline{x}) = \sum_{j=0}^{m} a_j x_j. \tag{10}$$

The typical choice of loss function in linear regression is the square loss, this gives:

$$\text{Loss}(f_{\underline{a}}(\underline{x}), y) = \left(f_{\underline{a}}(\underline{x}) - y\right)^2 = \left(\sum_{j=0}^{m} a_j x_j - y\right)^2. \tag{11}$$

We can then compute the gradient in the following way, noting that:

$$\frac{\partial}{\partial a_j}\text{Loss}(f_{\underline{a}}(\underline{x}), y) = 2\left[f_{\underline{a}}(\underline{x}) - y\right] x_j. \tag{12}$$

From this, we may reconstruct the gradient as

$$\nabla_{\underline{a}}\text{Loss}(f_{\underline{a}}(\underline{x}), y) = 2\left[f_{\underline{a}}(\underline{x}) - y\right] \underline{x}. \tag{13}$$

We can drop this immediately into the general form above:

Training a Linear Model with Stochastic Gradient Descent:

0) Choose some learning rate $\alpha > 0$.

1) Initialize $\underline{a}_0$ randomly (usually near $\underline{0}$) as an initial guess.

2) Choose some data point $(\underline{x}, y) \in$ Data. Update the parameters based on how poorly the model performs:

$$\underline{a}_{k+1} = \underline{a}_k - \alpha\left[f_{\underline{a}_k}(\underline{x}) - y\right] \underline{x}. \tag{14}$$

3) Repeat.

(Note, we are essentially 'absorbing' the factor of two into the learning, since it is just a constant.)

This formula has a nice intuition behind it - the amount we change the parameters will increase the larger the error $(f_{\underline{a}_k}(\underline{x}) - y)$ and the more significant the corresponding components of the input $\underline{x}$ are.

3 Logistic Regression

In this case, we will take the input space X to be some real vector space (again, taking the 0-component of each vector to be 1), and the output space Y to be the set of real numbers between 0 and 1, $Y = [0, 1]$. Logistic regression is frequently applied in binary classification: $y = 1$ if $\underline{x}$ belongs to a given class, and $y = 0$ if $\underline{x}$ does not belong to the class. For instance, $\underline{x}$ might be the pixel values of an image, and $y = 1$ or $y = 0$ depending on whether or not the image contains a dog.

For logistic regression, we take f to be a *non-linear* function of a linear combinations of the inputs. In particular, for some weight vector $\underline{a}$:

$$f_{\underline{a}}(\underline{x}) = \frac{1}{1 + e^{-(\underline{a}.\underline{x})}}. \tag{15}$$

*Note, defining the function $g(z) = 1/(1 + e^{-z})$ (the **sigmoid** function), it is convenient to denote $f_{\underline{a}}(\underline{x}) = g(\underline{a}.\underline{x})$.*

The typical choice of loss function in logistic regression is

$$\text{Loss}(f_{\underline{a}}(\underline{x}), y) = -y \ln \left[f_{\underline{a}}(\underline{x}) \right] - (1 - y) \ln \left[1 - f_{\underline{a}}(\underline{x}) \right]. \tag{16}$$

We can then compute the gradient in the following way, noting that:

$$\begin{aligned}
\frac{\partial}{\partial a_j}\text{Loss}(f_{\underline{a}}(\underline{x}), y) &= -y\frac{\frac{\partial}{\partial a_j} f_{\underline{a}}(\underline{x})}{f_{\underline{a}}(\underline{x})} - (1 - y)\frac{\frac{\partial}{\partial a_j}[1 - f_{\underline{a}}(\underline{x})]}{1 - f_{\underline{a}}(\underline{x})} \\
&= -y\frac{g'(\underline{a}.\underline{x})x_j}{f_{\underline{a}}(\underline{x})} - (1 - y)\frac{-g'(\underline{a}.\underline{x})x_j}{1 - f_{\underline{a}}(\underline{x})} \\
&= -\left[\frac{y}{f_{\underline{a}}(\underline{x})} - \frac{1 - y}{1 - f_{\underline{a}}(\underline{x})} \right] g'(\underline{a}.\underline{x})x_j \\
&= -\left[\frac{y(1 - f_{\underline{a}}(\underline{x})) - (1 - y)f_{\underline{a}}(\underline{x})}{f_{\underline{a}}(\underline{x})(1 - f_{\underline{a}}(\underline{x}))} \right] g'(\underline{a}.\underline{x})x_j \\
&= -\left[\frac{y - yf_{\underline{a}}(\underline{x}) - f_{\underline{a}}(\underline{x}) + yf_{\underline{a}}(\underline{x})}{f_{\underline{a}}(\underline{x})(1 - f_{\underline{a}}(\underline{x}))} \right] g'(\underline{a}.\underline{x})x_j \\
&= -\left[\frac{y - f_{\underline{a}}(\underline{x})}{f_{\underline{a}}(\underline{x})(1 - f_{\underline{a}}(\underline{x}))} \right] g'(\underline{a}.\underline{x})x_j \\
&= \left[\frac{f_{\underline{a}}(\underline{x}) - y}{f_{\underline{a}}(\underline{x})(1 - f_{\underline{a}}(\underline{x}))} \right] g'(\underline{a}.\underline{x})x_j
\end{aligned} \tag{17}$$

Noting that $g'(z) = g(z)(1 - g(z))$, (*why?*), we have that $g'(\underline{a}.\underline{x}) = f_{\underline{a}}(\underline{x})(1 - f_{\underline{a}}(\underline{x}))$, or

$$\begin{aligned}
\frac{\partial}{\partial a_j}\text{Loss}(f_{\underline{a}}(\underline{x}), \underline{y}) &= \left[\frac{f_{\underline{a}}(\underline{x}) - y}{f_{\underline{a}}(\underline{x})(1 - f_{\underline{a}}(\underline{x}))} \right] f_{\underline{a}}(\underline{x})(1 - f_{\underline{a}}(\underline{x}))x_j \\
&= \left[f_{\underline{a}}(\underline{x}) - y \right] x_j
\end{aligned} \tag{18}$$

From this, we may reconstruct the gradient as

$$\nabla_{\underline{a}}\text{Loss}(f_{\underline{a}}(\underline{x}), \underline{y}) = \left[f_{\underline{a}}(\underline{x}) - y \right] \underline{x}. \tag{19}$$

We can drop this immediately into the general form above:

Training a Logistic Model with Stochastic Gradient Descent:

0) Choose some learning rate $\alpha > 0$.

1) Initialize $\underline{a}_0$ randomly (usually near $\underline{0}$) as an initial guess.

2) Choose some data point $(\underline{x}, y) \in$ Data. Update the parameters based on how poorly the model performs:

$$\underline{a}_{k+1} = \underline{a}_k - \alpha \left[f_{\underline{a}_k}(\underline{x}) - y \right] \underline{x}. \tag{20}$$

3) Repeat.

The same sort of intuition applies as in the linear regression case - generally tuning the parameters proportionally to the current error, and the significance of the components of $\underline{x}$. It is interesting that the exact same update formula applies in the linear and logistic case. This is not accidental.

3.1 A Probabilistic Interpretation

One way of interpreting the logistic regression model is to take $f(\underline{x})$ as 'the probability that $\underline{x}$ should be labeled with Class 1 (and $1 - f(\underline{x})$ as the probability of being labeled with Class 0)'. In that case, what is the probability that the data you are given received the labeling it did? How *likely* was the data you received?

If each data point is labeled independently, you can argue that the likelihood of the whole labeling is given by

$$\text{likelihood} \propto \prod_{i=1}^{N} f(\underline{x}_i)^{y_i} (1 - f(\underline{x}_i))^{1-y_i} \quad \textit{(why?)}. \tag{21}$$

We are then interested in finding the f that *maximizes the probability of the specific labeling we received*, i.e., for what model f is the above likelihood as large as possible? Attempting to maximize this is the same as attempting to minimize the $-\ln$ likelihood, which results in the exact loss function specified above.

4 General Linear Models

To generalize the logistic model, we might consider f as a general non-linear (though differentiable) function of a linear function of $\underline{x}$. That is,

$$f_{\underline{a}}(\underline{x}) = g(\underline{a}.\underline{x}), \tag{22}$$

for some general differentiable function g. Logistic models simply take g as the sigmoid function. For a general loss function then, we have

$$\frac{\partial}{\partial a_j} \text{Loss}(f_{\underline{a}}(\underline{x}), y) = \left[\text{Loss}^{(1,0)}(f_{\underline{a}}(\underline{x}), y) \right] g'(\underline{a}.\underline{x}) x_j, \tag{23}$$

where $\text{Loss}^{(1,0)}$ refers to the first derivative with respect to the first argument. This gives a full gradient of

$$\nabla_{\underline{a}} \text{Loss}(f_{\underline{a}}(\underline{x}), y) = \left[\text{Loss}^{(1,0)}(f_{\underline{a}}(\underline{x}), y) \right] g'(\underline{a}.\underline{x}) \underline{x}. \tag{24}$$

Dropping this in:

Training a General Linear with Stochastic Gradient Descent:

0) Choose some learning rate $\alpha > 0$.

1) Initialize $\underline{a}_0$ randomly (usually near $\underline{0}$) as an initial guess.

2) Choose some data point $(\underline{x}, y) \in$ Data. Update the parameters based on how poorly the model performs:

$$\underline{a}_{k+1} = \underline{a}_k - \alpha \left[\text{Loss}^{(1,0)}(f_{\underline{a}_k}(\underline{x}), y)\right] g'(\underline{a}_k.\underline{x})\underline{x}. \tag{25}$$

3) Repeat.

This is a little more obscure, but still has a fairly natural interpretation: the $\left[\text{Loss}^{(1,0)}(f_{\underline{a}}(\underline{x}), y)\right]$ factors in the overall loss at the given point, and its relative rate of change; the $g'(\underline{a}.\underline{x})$ determines how sensitive f is to change at the given parameter values - for values of g' near 0, tweaking the parameter values will not change the value of f very much, because f is not particularly sensitive to changes there; the $\underline{x}$ factor then again scales everything by the relative significance of the components of $\underline{x}$.

5 Perceptrons for Binary Classification

Logistic regression sits at an interesting midpoint between regression and classification - the data points are assigned discrete labels (0 or 1), but we try to fit a continuous output function to those values. We can instead try to make a hard assignment of 0 or 1 to the data points. A natural way to do this is to use a general linear model, but take g as a hard threshold function, $g(z) = 1$ if $z \geq 0$, and $g(z) = 0$ else. This model is known as a linear perceptron:

$$f_{\underline{a}}(\underline{x}) = \begin{cases} 1 & \text{if } \underline{a}.\underline{x} \geq 0 \\ 0 & \text{else.} \end{cases} \tag{26}$$

One way of constructing a loss for a given model $f_{\underline{a}}$ then is to simply count the number of misclassified data points:

$$\text{Loss}(\underline{a}) = \sum_{i=1}^{N} \mathbf{1}\left\{f_{\underline{a}}(\underline{x}_i) \neq y_i\right\}. \tag{27}$$

This has a very nice geometric interpretation - the weights $\underline{a}$ can be taken to specify a *hyperplane* through the input space (defined by $\underline{a}.\underline{x} = 0$), and all the points located above this hyperplane are labeled with Class 1, everything below this hyperplane is labeled with Class 0, and the loss is simply the number of misclassified points.

However, we run into the following problem: because g is differentiable (in fact it is mostly differentiable, but the derivative is unhelpfully 0 almost everywhere) and the loss function is not differentiable, the typical training approaches of the previous sections do not apply. However we can extend these training approaches to the so-called **perceptron learning rule** in the following way:

Training a Perceptron:

1) Initialize $\underline{a}_0$ randomly (usually near $\underline{0}$) as an initial guess.

2) Choose some data point $(\underline{x}, y) \in$ Data. Update the parameters based on how poorly the model performs:

$$\underline{a}_{k+1} = \underline{a}_k - \left[f_{\underline{a}_k}(\underline{x} - y\right] \underline{x}. \tag{28}$$

3) Repeat.

It can be shown that if a *linear separator* (i.e., a hyperplane that cleanly divides the two classes) exists, then iterating this rule will discover it. It can be visualized in the sense that in every step, if the data point selected is classified correctly, it does not change the weights (/ location of the hyperplane), but if the data point is misclassified, it

adjusts the weights to correct for this error - in exactly the same way the previous schemes have done: scaling by the size of the error and the significance of the individual components. If a linear separator does not exist, there will be no convergence (as there will always be misclassified points), but if one does exist, this will find it. Interestingly, there is no need for a learning rate in this model - or in other words, a learning rate of $\alpha = 1$ suffices.

Generalizing this notion to non-linear separators and non-linearly separable data is largely the basis for **Support Vector Machines** and **Neural Networks**, respectively. But the general notion of

$$\text{new model} \;=\; \text{old model} \;-\; \text{learning rate} \;*\; \text{error} \;*\; \text{sensitivity}\;, \tag{29}$$

will continue to recur.