Machine Learning

Matthew Cocci

1 Machine Learning Problems

There are two main types of learning in the field of machine learning:

1. Supervised Learning: In this case, we know the right answer to our problem. For example, we might have data on house prices and a number of covariates, including bedrooms, age, location, etc. From there, we try to fit a model that best predicts price using the covariates, and we can check the fit of our model against the observed values.

2. Unsupervised Learning: Here, you don't know the right answer. For example, you might want to group a set of customers into clusters; however, there's no known right answer that you can use to check the quality of your clustering model.

These learning methods are applied to serve two main types of learning problems:

1. Regression Problems: The predicted or response variable takes on continuous values.

2. Classification Problems: Each observation or training example belongs to some category, and must be classified. This might be supervised (like a binary dead-alive type problem) or unsupervised (like clustering).

2 Terminology

In order to calibrate an algorithm, we use a set of **training examples** (the data), denoted $(\mathbf{x}^{(1)}, y^{(1)}), (\mathbf{x}^{(2)}, y^{(2)}), \dots, (\mathbf{x}^{(m)}, y^{(m)})$, where $(\mathbf{x}^{(i)}, y^{(i)})$ is the ith training example. Each training example consists of a vector of n **features** (covariates)

$$\mathbf{x}^{(i)} = \left(x_1^{(i)}, x_2^{(i)}, \dots, x_n^{(i)}\right)^T$$

We use these training examples to fit the parameters of a **hypothesis** (model), denoted $h_\theta(\mathbf{x})$. If we consider a simple linear model and that might look like

$$h_\theta(x) = \theta_0 + \theta_1 x$$

If we have a multivariate linear model, that would look like

$$\begin{aligned} h_\theta(\mathbf{x}) &= \theta^T\mathbf{x} \\ \mathbf{x} &= (1, x_1, x_2, \dots, x_n)^T \in \mathbb{R}^{n+1}, \qquad x_0 = 1 \\ \theta &= (\theta_0, \theta_1, \theta_2, \dots, \theta_n)^T \in \mathbb{R}^{n+1} \end{aligned}$$

3 Gradient Descent

Again, suppose we want to find point estimates of the components in a vector $\theta = (\theta_1, \theta_2, \dots, \theta_n)$ by minimizing some cost function, $\mathcal{J}(\theta)$.[1] Then we update our parameters all at once as follows:

$$\begin{aligned} \theta_1^{(t+1)} &= \theta_1^{(t)} - \alpha \frac{\partial}{\partial \theta_1^{(t)}} [\mathcal{J}(\theta)] \\ \vdots \qquad & \qquad \vdots \\ \theta_n^{(t+1)} &= \theta_n^{(t)} - \alpha \frac{\partial}{\partial \theta_n^{(t)}} [\mathcal{J}(\theta)] \end{aligned}$$

where α is the learning rate. This parameter controls the size of our steps and influences the aggressiveness of our descent. So use this process to keep updating and iterating until convergence.

Note At each iteration, we update all the components of θ *simultaneously.* That is, the equations above are a package deal. We don't update one parameter at a time as we did with Gibbs Sampling. Instead, we take all the partial derivatives of the cost function, $\mathcal{J}(\theta)$, then plug into the RHS of the above expressions.

Feature Scaling If we have multiple features, the gradient of descent may need some help to converge more quickly. In particular, if the features can take on ranges of values that are very different—i.e. $x_1^{(i)} \in (1,3)$ and $x_2^{(i)} \in (500, 10000)$—then our contours will appear very elongated and the gradient descent algorithm may converge slowly. To fix this, we might want to normalize the data and make each feature into a z-score or apply some other reasonable normoalization.

Choice of α: First, notice how this process self-adjusts in the magnitude of the update. In particular, as you get close to the minimum, the derivatives approach zero. So even with α constant, the shrinking derivative means that the gradient descent algorithm will take smaller steps, which is desireable. To choose a particular value of α, you face a tradeoff. Very small values of α will always allow the gradient descent algorithm to converge, but it will do so very slowly. Larger values of α risk overshooting the local minimum and may cause the algorithm to diverge.

[1]We can easily make the necessary changes if we want to maximize. In particular, we can minimize the negative of some target function. (The name "cost function" isn't really appropriate if we're trying to maximize.)

4 Multiple Regression

Recall the *normal equation* for multiple regression, which in matrix notation, says that parameter estimates should be

$$\theta = (X^T X)^{-1} X^T y$$

where X will be an $m \times n + 1$ matrix for m training examples and n features. The first feature of our $n + 1$ features in X is a 1 for each training example to include a constant in our regression. **Note** that we must have $m \geq n$—that is the number of training examples greater than the number of features—in order to invert the matrix in parentheses. Otherwise, the matrix is *singular* (non-invertible).

The typical cost function that we want to minimize (over m training examples) is:

$$\mathcal{J}(\theta) = \frac{1}{2} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2 \tag{1}$$

5 Logistic Regression

5.1 Single-Class Problems

Recall that the response variable can take on one of two values—typically positive or negative, or $y \in \{0, 1\}$. This is a classification problem where we want to assign observations (or training examples) to a group based on features. We set our hypothesis as follows:

$$h_\theta(x) = \frac{1}{1 + e^{-\theta^T x}}$$

where θ^T is a vector of $n+1$ parameters and x is a vector of $n+1$ features. The term on the right is called the *logistic* or *sigmoid* function, and it is guaranteed to lie between 0 and 1. The resulting output from $h_\theta(x)$ is a conditional probability that an observation is in the 1 group given the observed features.

Cost Function: It turns out that the cost function we used for multiple regression in Equation 1 will be *non-convex*, so the gradient descent algorithm will get stuck at local modes. This leads us to modify our cost function in line with maximum likelihood so that the cost for a particular training example is

$$\begin{aligned} \text{Cost}(h_\theta(x),\ y) &= \begin{cases} -\ln h_\theta(x) & \text{if } y = 1 \\ -\ln(1 - h_\theta(x)) & \text{if } y = 0 \end{cases} \\ &= -y \ln h_\theta(x) - (1 - y) \ln(1 - h_\theta(x)) \qquad (2) \\ \Rightarrow \quad \mathcal{J}(\theta) &= -\frac{1}{m} \sum_{i=1}^{m} \left[y^{(i)} \ln h_\theta(x^{(i)}) + (1 - y^{(i)}) \ln(1 - h_\theta(x^{(i)})) \right] \qquad (3) \end{aligned}$$

5.2 Multi-Class Problems

Suppose that we want to generalize single-class logistic regression to the case of multiple groups. Specifically, suppose that each $y^{(i)}$ can belong to groups $\{1, 2, \ldots, k\}$. Then the *one-vs-all* approach says

1. Run k different logistic regressions, where you take one group, i, and treat it as the posive case. The remaining $k-1$ groups are treated as one big negative group.
2. Using your training examples to fit a hypothesis for each such logistic regression so that there are k hypotheses in total: $h_\theta^{(i)}(x), \ldots, h_\theta^{(k)}(x)$.
3. Then, given some covariates, x, you predict the group for this observation to be

$$\max_i\{h_\theta^{(1)}(x), \ldots, h_\theta^{(k)}(x)\}$$

   i.e. group i, the group that has the maximum chance of containing the observation.

6 Regularization

In order to avoid the problem of overfitting, we typically apply *regularization*, where we penalize large values of the parameters we're trying to fit by adding a term to the cost function. This means we want to find

$$\arg\min_\theta \mathcal{J}(\theta) = f\left(h_\theta(x), y\right) + g(\theta) \tag{4}$$

where $\theta = (\theta_1, \ldots, \theta_n)$ and where x and y are a vector of m training examples. Note that θ_0—the conventional constant parameter—is not included in the regularization.

As an example of a standard form for Equation 4, we might choose

$$\arg\min_\theta \mathcal{J}(\theta) = \arg\min \frac{1}{2m}\left[\sum_{i=1}^m \left(h_\theta(x^{(i)}) - y^{(i)}\right)^2 + \lambda \sum_{j=1}^n \theta_j^2\right] \tag{5}$$

6.1 Implementation by Normal Equations, Multiple Regression

Suppose we consider the problem of multiple regression, and we need to choose the regression parameters with the regularization and cost function in Equation 5. Then the normal equations are modified to

$$\theta = \left(X^T X + \lambda R\right)^{-1} X^T y \tag{6}$$

where R is nearly the identity matrix I_n, but with entry $R_{1,1} = 0$ instead of 1.

Note that *without* regularization we must have $m \geq n$—that is the number of training examples greater than the number of features. Otherwise the matrix in parentheses is *singular* (non-invertible). **But** if we choose $\lambda > 0$ and apply regularization, then the matrix in parentheses is *guaranteed* to be invertible, even if $m < n$.

6.2 Implementation by Gradient Descent, Multiple Regression

If this were used in an implementation of gradient descent, this would yield an updating rule of:

$$
\begin{aligned}
\theta_0 &:= \theta_0 - \alpha \, \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right) x_0^{(i)} \\
\theta_j &:= \theta_j - \alpha \left[\frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right) x_j^{(i)} + \frac{\lambda}{m} \theta_j \right] \\
&:= \theta_j \left(1 - \alpha \frac{\lambda}{m} \right) - \alpha \, \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right) x_j^{(i)}
\end{aligned}
$$

where $j = 1, \ldots, n$ and $h_\theta(x)$ is the standard multiple regression form. It's clear that the first term is θ_j times a constant less than 1, so we're *shrinking* the parameter towards zero, then the second term updates as before the shrunken parameter.

6.3 Implementation by Gradient Descent, Logistic Regression

If we want to implement a regularized logistic regression, we'll take the cost function to be

$$
\mathcal{J}(\theta) = - \left[\frac{1}{m} \sum_{i=1}^{m} y^{(i)} \ln h_\theta(x^{(i)}) + (1 - y^{(i)}) \ln \left(1 - h_\theta(x^{(i)}) \right) \right] + \frac{\lambda}{2m} \sum_{j=1}^{n} \theta_j^2 \tag{7}
$$

Given this cost function, the update rule for gradient descent becomes

$$
\begin{aligned}
\theta_0 &:= \theta_0 - \alpha \, \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right) x_0^{(i)} \\
\theta_j &:= \theta_j - \alpha \left[\frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right) x_j^{(i)} + \frac{\lambda}{m} \theta_j \right]
\end{aligned}
$$

where $j = 1, \ldots, n$. Now this looks very similar to the update rule for multiple regression, except it doesn't make explicit the different hypothesis functions. In particular, for logistic regression, we have

$$
h_\theta(x) = \frac{1}{1 + e^{-\theta^T x}}
$$

7 Neural Networks

This is useful for classification problems involving a high number of features. We saw with logistic regression that including different powers and cross-products of the features can give a better fit for the classification problem, fitting circles and ellipses to the data; *however*, while this works well for, say, two features, the parameter optimization problem grows very quickly with more features.

Neural networks are also very useful because they can represent highly non-linear functions and classifications. In particular, they can even be used to construct logical decision-making and classification.

7.1 Model Representation

Here's the basic idea: between the *input layer* (the features) and the *output layer* (the hypothesis), we introduce at least one *hidden layer* transforming the inputs into new variables that will be used in the hypothesis. This forms the *architecture* of the network.

We introduce the following notation to be used in computing/evaluating layers,[2]

$$\begin{aligned} a_i^{(j)} &= \text{the ``activation'' of unit } i \text{ in layer } j \\ \Theta^{(j)} &= \text{matrix of weights controlling function} \\ &\qquad \text{mapping from layer } j \text{ to } j+1 \end{aligned}$$

Now suppose there are n features, L hidden layers, and s_j units in layer j. Note that in each layer, we'll typically introduce a constant term, x_0 (for the features) or $a_0^{(j)}$ in the intermediate layers. This is commonly called the "bias unit."

$$a_i^{(2)} = f_i^{(1)}\left(x_0, \ldots, x_n, \Theta_{i0}^{(1)}, \ldots, \Theta_{in}^{(1)}\right), \qquad i = 1, \ldots, n$$

$$a_i^{(j+1)} = f_i^{(j)}\left(a_0^{(j)}, \ldots, a_{s_j}^{(j)},\ \Theta_{i0}^{(j)}, \ldots, \Theta_{is_j}^{(j)}\right), \qquad \begin{cases} i = 1, \ldots, s_{j+1} \\ j = 1, \ldots, L-1 \end{cases}$$

$$h_\Theta(x) = a_1^{(L)} = f_1^{(L-1)}\left(a_0^{(L-1)}, \ldots, a_{s_{L-1}}^{(L-1)},\ \Theta_{i0}^{(L-1)}, \ldots, \Theta_{is_{L-1}}^{(L-1)}\right)$$

It follows, by our definition of s_j and $\Theta^{(j)}$, that $\Theta^{(j)}$ is of dimension $s_{j+1} \times (s_j + 1)$. I know; there's a lot of notation. But basically, it says each unit in the first layer is some function of the features and parameters. Then the activiation of all units in subsequent layers are functions of the activations of units in the immediately previous layer. This goes on until you get to your hypothesis, and this process is called *forward propogation.*

[2] Don't take "activation" and "weights" too seriously. "Activation" just indicates the *value* computed and output by a specific layer or unit in the layer. In addition, "weights" are just the name for parameters in the neural networks literature.

7.2 Vectorized Implementation

Here, we simplify notation a bit and take the special case where the function of the attributes, $g(\cdot)$, is the sigmoid function. So we define the stage 2 attributes as follows

$$\begin{aligned} a_1^{(2)} &= g\left(\Theta_{10}^{(1)} x_0 + \Theta_{11}^{(1)} x_1 + \cdots + \Theta_{1n}^{(1)} x_n\right) = g\left(z_1^{(2)}\right) \\ \vdots \quad & \quad \vdots \\ a_{s_2}^{(2)} &= g\left(\Theta_{s_2 0}^{(1)} x_0 + \Theta_{s_2 1}^{(1)} x_1 + \cdots + \Theta_{s_2 n}^{(1)} x_n\right) = g\left(z_{s_2}^{(2)}\right) \end{aligned}$$

Now let's vectorize, redefining the vector x of dimension $(n+1) \times 1$ (with one more row than the number of features for the extra constant term) as follows to make notation simpler: $x = a^{(1)}$. Next, we define $z^{(j)}$ as a vector of dimension $s_j \times 1$, where s_j is the number of attributes in layer j. We do the following steps to compute the hypothesis:

1. Compute $z^{(2)} = \Theta^{(1)} x = \Theta^{(1)} a^{(1)}$, then use $z^{(2)}$ to compute $a^{(2)} = g(z^{(2)})$.
2. Prepend $a_0^{(2)} = 1$ to $a^{(2)}$ to allow for a constant/bias term, then compute $z^{(3)} = \Theta^{(2)} a^{(2)}$.
3. Continue this process, prepending $a_0^{(j)} = 1$ to $a^{(j)}$, computing $z^{(j+1)} = \Theta^{(j)} a^{(j)}$ for all j number of layers until you reach $h_\Theta(x)$, a process called *forward propagation.*

7.3 Cost Function for a Classfication Neural Network

Binary Classification: In this special case, $y \in \{0, 1\}$ and there is 1 output unit that computes $h_\Theta(x) \in \mathbb{R}$. So $s_L = 1$, where s_ℓ is the number of units in layer ℓ.

Multi-Class Classification: In this more general case, $y \in \mathbb{R}^K$ where there are K distinct classes. For ex., y could be one of four classes (pedestrian, car, motorcycle, truck), denoted

$$y = \begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix}, \qquad \begin{bmatrix} 0 & 1 & 0 & 0 \end{bmatrix}, \qquad \begin{bmatrix} 0 & 0 & 1 & 0 \end{bmatrix}, \qquad \text{or} \qquad \begin{bmatrix} 0 & 0 & 0 & 1 \end{bmatrix}$$

So there are K output units. Typically, we only use this setup when $K \geq 3$.

Cost Function: This is a generalization of the cost function for logistic regression:

$$\begin{aligned} & h_\Theta(x) \in \mathbb{R}^K \qquad (h_\Theta(x))_i = i\text{th output} \\ & J(\Theta) = -\frac{1}{m}\left[\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \ln\left(h_\Theta(x^{(i)})\right)_k + \left(1 - y_k^{(i)}\right) \ln\left(1 - \left(h_\Theta(x^{(i)})\right)_k\right)\right] \\ & \qquad\qquad + \frac{\lambda}{2m} \sum_{\ell=1}^{L-1} \sum_{i=1}^{s_\ell} \sum_{j=1}^{s_{\ell+1}} \left(\Theta_{ji}^{(\ell)}\right)^2 \end{aligned} \tag{8}$$

The first term is a simple generalization of the regular logistic regression cost function—we just sum over the K different classes. Note that the next term in $J(\Theta)$ is a summation over weights *not* including the constant bias terms (so we don't include $i = 0$).

7.4 Backpropogation Algorithm

In order to use gradient descent, we need the value of the cost function, $J(\Theta)$, along with the partial derivatives, $\frac{\partial J(\Theta)}{\partial \Theta_{ij}^{(\ell)}}$.

To compute the derivatives, we use the *backpropogation* algorithm. Here's the intuition and the impliementation for multi-class logistic classification:

1. At each node, we compute an "error" term,
$$\delta_j^{(\ell)} = \text{“error” of node } j \text{ in layer } \ell.$$
2. We start with the final node:
$$\delta_j^{(L)} = a_j^{(L)} - y_j = \left(h_\Theta(x)\right)_j - y_j$$
3. We then move to earlier nodes:
$$\delta_j^{(\ell)} = \left(\Theta^{(\ell)}\right)^T \delta^{(\ell+1)} .* \, g'\left(z^{(\ell)}\right), \qquad z^{(\ell+1)} = \Theta^{(\ell)} a^{(\ell)}$$
where $.*$ is elementwise multiplication.
4. It then follows from a lengthy proof that
$$\frac{\partial J(\Theta)}{\partial \Theta_{ij}^{(\ell)}} = a_j^{(\ell)} \delta_i^{(\ell+1)}$$