12. Machine Learning

Chris Piech and Mehran Sahami

May 2017

Machine Learning is the subfield of computer science that gives computers the ability to perform tasks without being explicitly programmed. There are several different tasks that fall under the domain of machine learning and several different algorithms for "learning". In this chapter we are going to focus on Classification and two classic Classification algorithms: Naïve Bayes and Logistic Regression.

1 Classification

In classification tasks, your job is to use training data with feature/label pairs (**x**, y) in order to estimate a function $\hat{y} = g(\mathbf{x})$. This function can then be used to make a prediction. In classification the value of y takes on one of a discrete number of values. As such we often chose $g(\mathbf{x}) = \underset{y}{\text{argmax}}\ \hat{P}(Y = y|\mathbf{X})$.

In the classification task you are given N training pairs: $(\mathbf{x}^{(1)}, y^{(1)}), (\mathbf{x}^{(2)}, y^{(2)}), \ldots, (\mathbf{x}^{(N)}, y^{(N)})$ Where $\mathbf{x}^{(i)}$ is a vector of m discrete features for the ith training example and $y^{(i)}$ is the discrete label for the ith training example.

In this chapter we are going to assume that all values in our training data-set are binary. While this is not a necessary assumption (both naive bayes and logistic regression can work for non-binary data), it makes it much easier to learn the core concepts. Specifically we assume that all labels are binary $y^{(i)} \in \{0, 1\}\ \forall i$ and all features are binary $x_j^{(i)} \in \{0, 1\}\ \forall i, j$.

2 Naïve Bayes algorithm

Here is the Naïve Bayes algorithm. After presenting the algorithm I am going to show the theory behind it.

Training

The objective in training is to estimate the probabilities $P(Y)$ and $P(X_i|Y)$ for all $0 < i \leq m$ features.

Using an MLE estimate:

$$\hat{p}(X_i = x_i|Y = y) = \frac{(\text{\# training examples where } X_i = x_i \text{ and } Y = y)}{(\text{training examples where } Y = y)}$$

Using a Laplace MAP estimate:

$$\hat{p}(X_i = x_i|Y = y) = \frac{(\text{\# training examples where } X_i = x_i \text{ and } Y = y) + 1}{(\text{training examples where } Y = y) + 2}$$

Prediction

For an example with $\mathbf{x} = [x_1, x_2, \ldots, x_m]$, estimate the value of y as:

$$\begin{aligned}
\hat{y} = g(\mathbf{x}) &= \underset{y}{\text{argmax}}\ \hat{P}(\mathbf{X}|Y)\hat{P}(Y) && \text{This is equal to argmax } \hat{P}(Y=y|\mathbf{X}) \\
&= \underset{y}{\text{argmax}} \prod_{i=1}^{m} \hat{p}(X_i = x_i|Y=y)\hat{p}(Y=y) && \text{Naïve Bayes assumption} \\
&= \underset{y}{\text{argmax}} \sum_{i=1}^{m} \log \hat{p}(X_i = x_i|Y=y) + \log \hat{p}(Y=y) && \text{Log version for numerical stability}
\end{aligned}$$

Note that for small enough datasets you may not need to use the log version of the argmax.

Theory

We can solve the classification task using a brute force solution. To do so we will learn the full joint distribution $\hat{P}(Y, \mathbf{X})$. In the world of classification, when we make a prediction, we want to chose the value of y that maximizes: $g(\mathbf{x}) = \underset{y}{\text{argmax}}\ \hat{P}(Y=y|\mathbf{X})$.

$$\begin{aligned}
\hat{y} = g(\mathbf{x}) &= \underset{y}{\text{argmax}}\ \hat{P}(Y|\mathbf{X}) = \underset{y}{\text{argmax}}\ \frac{\hat{P}(\mathbf{X}, Y)}{\hat{P}(\mathbf{X})} && \text{By definition of conditional probability} \\
&= \underset{y}{\text{argmax}}\ \hat{P}(\mathbf{X}, Y) && \text{Since } \hat{P}(\mathbf{X}) \text{ is constant with respect to Y}
\end{aligned}$$

Using our training data we could interpret the joint distribution of $\mathbf{X}$ and Y as one giant multinomial with a different parameter for every combination of $\mathbf{X} = \mathbf{x}$ and $Y = y$. If for example, the input vectors are only length one. In other words $|\mathbf{x}| = 1$ and the number of values that x and y can take on are small, say binary, this is a totally reasonable approach. We could estimate the multinomial using MLE or MAP estimators and then calculate argmax over a few lookups in our table.

The bad times hit when the number of features becomes large. Recall that our multinomial needs to estimate a parameter for every unique combination of assignments to the vector $\mathbf{x}$ and the value y. If there are $|\mathbf{x}| = n$ binary features then this strategy is going to take order $\mathcal{O}(2^n)$ space and there will likely be many parameters that are estimated without any training data that matches the corresponding assignment.

Naïve Bayes Assumption

The Naïve Bayes Assumption is that each feature of $\mathbf{x}$ is independent of one another given y. That assumption is wrong, but useful. This assumption allows us to make predictions using space and data which is linear with respect to the size of the features: $\mathcal{O}(n)$ if $|\mathbf{x}| = n$. That allows us to train and make predictions for huge feature spaces such as one which has an indicator for every word on the internet. Using this assumption the prediction algorithm can be simplified.

$$\begin{aligned}
\hat{y} = g(\mathbf{x}) &= \underset{y}{\text{argmax}}\ \hat{P}(\mathbf{X}, Y) && \text{As we last left off} \\
&= \underset{y}{\text{argmax}}\ \hat{P}(\mathbf{X}|Y)\hat{P}(Y) && \text{By chain rule} \\
&= \underset{y}{\text{argmax}} \prod_{i=1}^{n} \hat{p}(X_i|Y)\hat{P}(Y) && \text{Using the naïve bayes assumption} \\
&= \underset{y}{\text{argmax}} \sum_{i=1}^{m} \log \hat{p}(X_i = x_i|Y=y) + \log \hat{p}(Y=y) && \text{Log version for numerical stability}
\end{aligned}$$

This algorithm is both fast and stable both when training and making predictions. If we think of each X_i, y pair as a multinomial we can find MLE and MAP estimations for the values. See the "algorithm" section for the optimal values of each p in the multinomial.

Naïve Bayes is a simple form of a field of machine learning called Probabilistic Graphical Models. In that field you make a graph of how your variables are related to one another and you come up with conditional independence assumptions that make it computationally tractable to estimate the joint distribution.

Example

Say we have thirty examples of people's preferences (like or not) for Star Wars, Harry Potter and Lord of the Rings. Each training example has x_1, x_2 and y where x_1 is whether or not the user liked Star Wars, x_2 is whether or not the user liked Harry Potter and y is whether or not the user liked Lord of the Rings.

For the 30 training examples the MAP and MLE estimates are as follows:

Y \ X_1	0	1	MLE estimates	
0	3	10	0.10	0.33
1	4	13	0.13	0.43

Y \ X_2	0	1	MLE estimates	
0	5	8	0.17	0.27
1	7	10	0.23	0.33

Y	#	MLE est.
0	13	0.43
1	17	0.57

For a new user who likes Star Wars ($x_1 = 1$) but not Harry Potter ($x_2 = 0$) do you predict that they will like Lord of the Rings? See the lecture slides for the answer.

3 Logistic Regression

Logistic Regression is a classification algorithm (I know, terrible name. Perhaps Logistic Classification would have been better) that works by trying to learn a function that approximates $P(Y|X)$. It makes the central assumption that $P(Y|X)$ can be approximated as a sigmoid function applied to a linear combination of input features. It is particularly important to learn because logistic regression is the basic building block of artificial neural networks.

Mathematically, for a single training datapoint $(\mathbf{x}, y)$ Logistic Regression assumes:

$$P(Y = 1|\mathbf{X} = \mathbf{x}) = \sigma(z) \text{ where } z = \theta_0 + \sum_{i=1}^{m} \theta_i x_i$$

This assumption is often written in the equivalent forms:

$$P(Y = 1|\mathbf{X} = \mathbf{x}) = \sigma(\theta^T \mathbf{x}) \qquad \text{where we always set } x_0 \text{ to be 1}$$
$$P(Y = 0|\mathbf{X} = \mathbf{x}) = 1 - \sigma(\theta^T \mathbf{x}) \qquad \text{by total law of probability}$$

Using these equations for probability of $Y|X$ we can create an algorithm that selects values of theta that maximize that probability for all data. I am first going to state the log probability function and partial derivatives with respect to theta. Then later we will (a) show an algorithm that can chose optimal values of theta and (b) show how the equations were derived.

An important thing to realize is that: given the best values for the parameters (θ), logistic regression often can do a great job of estimating the probability of different class labels. However, given bad , or even random, values of θ it does a poor job. The amount of "intelligence" that you logistic regression machine learning algorithm has is dependent on having good values of θ.

Notation

Before we get started I want to make sure that we are all on the same page with respect to notation. In logistic regression, θ is a vector of parameters of length m and we are going to learn the values of those parameters

based off of n training examples. The number of parameters should be equal to the number of features of each datapoint (see section 1).

Two pieces of notation that we use often in logistic regression that you may not be familiar with are:

$$\theta^T \mathbf{x} = \sum_{i=1}^{m} \theta_i x_i = \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_m x_m \qquad \text{dot product, aka weighted sum}$$

$$\sigma(z) = \frac{1}{1+e^{-z}} \qquad \text{sigmoid function}$$

Log Likelihood

In order to chose values for the parameters of logistic regression we use Maximum Likelihood Estimation (MLE). As such we are going to have two steps: (1) write the log-likelihood function and (2) find the values of θ that maximize the log-likelihood function.

The labels that we are predicting are binary, and the output of our logistic regression function is supposed to be the probability that the label is one. This means that we can (and should) interpret the each label as a Bernoulli random variable: $Y \sim \text{Bern}(p)$ where $p = \sigma(\theta^T \mathbf{x})$.

To start, here is a super slick way of writing the probability of one datapoint (recall this is the equation form of the probability mass function of a Bernoulli):

$$P(Y = y | X = \mathbf{x}) = \sigma(\theta^T \mathbf{x})^y \cdot \left[1 - \sigma(\theta^T \mathbf{x})\right]^{(1-y)}$$

Now that we know the probability mass function, we can write the likelihood of all the data:

$$L(\theta) = \prod_{i=1}^{n} P(Y = y^{(i)} | X = \mathbf{x}^{(i)}) \qquad \text{The likelihood of independent training labels}$$

$$= \prod_{i=1}^{n} \sigma(\theta^T \mathbf{x}^{(i)})^{y^{(i)}} \cdot \left[1 - \sigma(\theta^T \mathbf{x}^{(i)})\right]^{(1-y^{(i)})} \qquad \text{Substituting the likelihood of a Bernoulli}$$

And if you take the log of this function, you get the reported Log Likelihood for Logistic Regression. The log likelihood equation is:

$$LL(\theta) = \sum_{i=1}^{n} y^{(i)} \log \sigma(\theta^T \mathbf{x}^{(i)}) + (1 - y^{(i)}) \log[1 - \sigma(\theta^T \mathbf{x}^{(i)})]$$

Recall that in MLE the only remaining step is to chose parameters (θ) that maximize log likelihood.

Gradient of Log Likelihood

Now that we have a function for log-likelihood, we simply need to chose the values of theta that maximize it. We can find the best values of theta by using an optimization algorithm. However, in order to use an optimization algorithm, we first need to know the partial derivative of log likelihood with respect to each parameter. First I am going to give you the partial derivative (so you can see how it is used). Then I am going to show you how to derive it:

$$\frac{\partial LL(\theta)}{\partial \theta_j} = \sum_{i=1}^{n} \left[y^{(i)} - \sigma(\theta^T \mathbf{x}^{(i)}) \right] x_j^{(i)}$$

Gradient Ascent Optimization

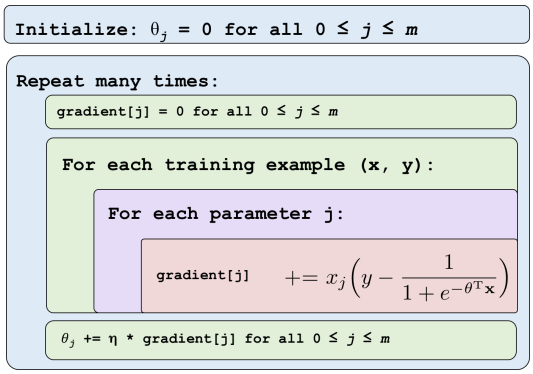
Our goal is to choosing parameters (θ) that maximize likelihood, and we know the partial derivative of log likelihood with respect to each parameter. We are ready for our optimization algorithm.

In the case of logistic regression we can't solve for θ mathematically. Instead we use a computer to chose θ. To do so we employ an algorithm called gradient ascent (a classic in optimization theory). The idea behind gradient ascent is that if you continuously take small steps in the direction of your gradient, you will eventually make it to a local maxima. In the case of logistic regression you can prove that the result will always be a global maxima.

The update to our parameters that results in each small step can be calculated as:

$$\begin{aligned}\theta_j^{\text{new}} &= \theta_j^{\text{old}} + \eta \cdot \frac{\partial LL(\theta^{\text{old}})}{\partial \theta_j^{\text{old}}} \\ &= \theta_j^{\text{old}} + \eta \cdot \sum_{i=1}^{n} \left[y^{(i)} - \sigma(\theta^T \mathbf{x}^{(i)}) \right] x_j^{(i)}\end{aligned}$$

Where η is the magnitude of the step size that we take. If you keep updating θ using the equation above you will converge on the best values of θ. You now have an intelligent model. Here is the gradient ascent algorithm for logistic regression in pseudo-code:



Pro-tip: Don't forget that in order to learn the value of θ_0 you can simply define $\mathbf{x}_0$ to always be 1.

Derivations

In this section we provide the mathematical derivations for the gradient of log-likelihood. The derivations are worth knowing because these ideas are heavily used in Artificial Neural Networks.

Our goal is to calculate the derivative of the log likelihood with respect to each theta. To start, here is the definition for the derivative of a sigmoid function with respect to its inputs:

$$\frac{\partial}{\partial z}\sigma(z) = \sigma(z)[1-\sigma(z)] \qquad \text{to get the derivative with respect to } \theta\text{, use the chain rule}$$

Take a moment and appreciate the beauty of the derivative of the sigmoid function. The reason that sigmoid has such a simple derivative stems from the natural exponent in the sigmoid denominator.

Since the likelihood function is a sum over all of the data, and in calculus the derivative of a sum is the sum of derivatives, we can focus on computing the derivative of one example. The gradient of theta is simply the sum of this term for each training datapoint.

First I am going to show you how to compute the derivative the hard way. Then we are going to look at an

easier method. The derivative of gradient for one datapoint $(\mathbf{x}, y)$:

$$\begin{aligned}
\frac{\partial LL(\theta)}{\partial \theta_j} &= \frac{\partial}{\partial \theta_j} y \log \sigma(\theta^T \mathbf{x}) + \frac{\partial}{\partial \theta_j} (1-y) \log[1 - \sigma(\theta^T \mathbf{x}] && \text{derivative of sum of terms} \\
&= \left[\frac{y}{\sigma(\theta^T \mathbf{x})} - \frac{1-y}{1-\sigma(\theta^T \mathbf{x})} \right] \frac{\partial}{\partial \theta_j} \sigma(\theta^T \mathbf{x}) && \text{derivative of } \log f(x) \\
&= \left[\frac{y}{\sigma(\theta^T \mathbf{x})} - \frac{1-y}{1-\sigma(\theta^T \mathbf{x})} \right] \sigma(\theta^T \mathbf{x})[1 - \sigma(\theta^T \mathbf{x})] \mathbf{x}_j && \text{chain rule + derivative of sigma} \\
&= \left[\frac{y - \sigma(\theta^T \mathbf{x})}{\sigma(\theta^T \mathbf{x})[1 - \sigma(\theta^T \mathbf{x})]} \right] \sigma(\theta^T \mathbf{x})[1 - \sigma(\theta^T \mathbf{x})] \mathbf{x}_j && \text{algebraic manipulation} \\
&= \left[y - \sigma(\theta^T \mathbf{x}) \right] \mathbf{x}_j && \text{cancelling terms}
\end{aligned}$$

Derivatives Without Tears

That was the hard way. Logistic regression is the building block of Artificial Neural Networks. If we want to scale up, we are going to have to get used to an easier way of calculating derivatives. For that we are going to have to welcome back our old friend the chain rule. By the chain rule:

$$\begin{aligned}
\frac{\partial LL(\theta)}{\partial \theta_j} &= \frac{\partial LL(\theta)}{\partial p} \cdot \frac{\partial p}{\partial \theta_j} && \text{Where } p = \sigma(\theta^T \mathbf{x}) \\
&= \frac{\partial LL(\theta)}{\partial p} \cdot \frac{\partial p}{\partial z} \cdot \frac{\partial z}{\partial \theta_j} && \text{Where } z = \theta^T \mathbf{x}
\end{aligned}$$

Chain rule is the decomposition mechanism of calculus. It allows us to calculate a complicated partial derivative ($\frac{\partial LL(\theta)}{\partial \theta_j}$) by breaking it down into smaller pieces.

$$\begin{aligned}
LL(\theta) &= y \log p + (1-y) \log(1-p) && \text{Where } p = \sigma(\theta^T \mathbf{x}) \\
\frac{\partial LL(\theta)}{\partial p} &= \frac{y}{p} - \frac{1-y}{1-p} && \text{By taking the derivative} \\
\hline
p &= \sigma(z) && \text{Where } z = \theta^T \mathbf{x} \\
\frac{\partial p}{\partial z} &= \sigma(z)[1 - \sigma(z)] && \text{By taking the derivative of the sigmoid} \\
\hline
z &= \theta^T \mathbf{x} && \text{As previously defined} \\
\frac{\partial z}{\partial \theta_j} &= \mathbf{x}_j && \text{Only } \mathbf{x}_j \text{ interacts with } \theta_j
\end{aligned}$$

Each of those derivatives was much easier to calculate. Now we simply multiply them together.

$$\begin{aligned}
\frac{\partial LL(\theta)}{\partial \theta_j} &= \frac{\partial LL(\theta)}{\partial p} \cdot \frac{\partial p}{\partial z} \cdot \frac{\partial z}{\partial \theta_j} \\
&= \left[\frac{y}{p} - \frac{1-y}{1-p} \right] \cdot \sigma(z)[1 - \sigma(z)] \cdot \mathbf{x}_j && \text{By substituting in for each term} \\
&= \left[\frac{y}{p} - \frac{1-y}{1-p} \right] \cdot p[1-p] \cdot \mathbf{x}_j && \text{Since } p = \sigma(z) \\
&= [y(1-p) - p(1-y)] \cdot \mathbf{x}_j && \text{Multiplying in} \\
&= [y - p] \mathbf{x}_j && \text{Expanding} \\
&= [y - \sigma(\theta^T \mathbf{x})] \mathbf{x}_j && \text{Since } p = \sigma(\theta^T \mathbf{x})
\end{aligned}$$