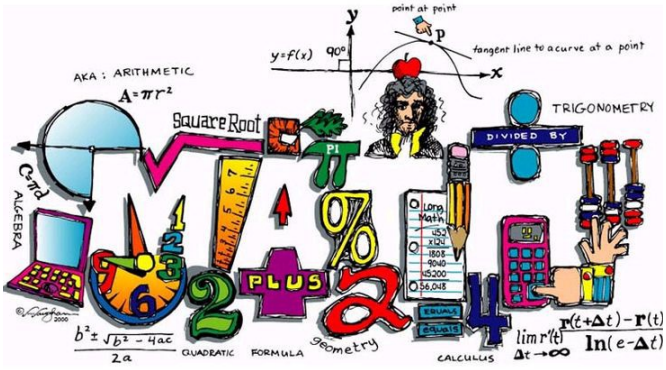
Estimators, Loss Functions, Optimizers —Core of ML Algorithms

Javaid Nabi Follow

May 24 · 13 min read ★

In order to understand how a machine learning algorithm learns from data to predict an outcome, it is essential to understand the underlying concepts involved in training an algorithm.

I assume you have basic machine learning understanding and also basic knowledge of probability and statistics. If not please go through my earlier posts here and here. This post has some theory and maths involved so bear with me and once you read till the end, it will make complete sense as we connect the dots.



source

Estimators

Estimation is a statistical term for finding some estimate of unknown parameter, given some data. Point Estimation is the attempt to provide the single best prediction of some quantity of interest.

Quantity of interest can be:

- A single parameter
- A vector of parameters—e.g., weights in linear regression
- A whole function

Point estimator

To distinguish estimates of parameters from their true value, a point estimate of a parameter `θ` is represented by `θ^`. Let `{x(1) , x (2) ,..x(m)}` be `m` independent and identically distributed data points.Then a point estimator is any function of the data:

$$\hat{\boldsymbol{\theta}}_m = g(\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)})$$

This definition of a point estimator is very general and allows the designer of an estimator great flexibility. While almost any function thus qualifies as an estimator, a good estimator is a function whose output is close to the true underlying θ that generated the training data.

Point estimation can also refer to estimation of relationship between input and target variables referred to as function estimation.

Function Estimation

Here we are trying to predict a variable y given an input vector x. We assume that there is a function `f(x)` that describes the approximate relationship between y and x. For example,

we may assume that `y = f(x) + ε` , where `ε` stands for the part of `y` that is not predictable from `x` . In function estimation, we are interested in approximating `f` with a model or estimate `f^` . Function estimation is really just the same as estimating a parameter `θ` ; the function estimator `f^` is simply a point estimator in function space. Ex: in polynomial regression we are either estimating a parameter `w` or estimating a function mapping from `x` to `y` .

Bias and Variance

Bias and variance measure two different sources of error in an estimator. Bias
measures the expected deviation from the true value of the function or parameter. Variance on the other hand, provides a measure of the deviation from the expected estimator value that any particular sampling of the data is likely to cause.

Bias

The bias of an estimator is defined as:

$$\text{bias}(\hat{\boldsymbol{\theta}}_m) = \mathbb{E}(\hat{\boldsymbol{\theta}}_m) - \boldsymbol{\theta}$$

where the expectation is over the data (seen as samples from a random variable)and `θ` is the true underlying value of `θ` used to define the data generating distribution.

An estimator `θ^m` is said to be unbiased if `bias(θ^m) = 0` , which implies
that `E(θ^m) = θ .`

Variance and Standard Error

The variance of an estimator `Var(θ^)` where the random variable is the training set. Alternately, the square root of the variance is called the standard error, denoted standard error `SE(^θ)` . The variance or the standard error of an estimator provides a measure of how we would expect the estimate we compute from data to vary as we independently re-sample the dataset from the underlying data generating process.

> *Just as we might like an estimator to exhibit low bias we would also like it to have relatively low variance.*

Having discussed the definition of an estimator, let us now discuss some commonly used estimators.

Maximum Likelihood Estimator (MLE)

Maximum Likelihood Estimation can be defined as a method for estimating parameters (such as the mean or variance) from sample data such that the probability (likelihood) of obtaining the observed data is maximized.

Consider a set of `m` examples `X = {x(1), . . . , x(m)}` drawn independently from the true but unknown data generating distribution `Pdata(x)` . Let `Pmodel(x; θ)` be a parametric family of probability distributions over the same space indexed by `θ` . In other words, `Pmodel(x; θ)` maps any configuration `x` to a real number estimating the true probability `Pdata(x) .`
The maximum likelihood estimator for `θ` is then defined as:

$$\boldsymbol{\theta}_{\mathrm{ML}} = \arg\max_{\boldsymbol{\theta}} p_{\mathrm{model}}(\mathbb{X}; \boldsymbol{\theta})$$

Since we assumed the examples to be i.i.d, the above equation can be written in the product form as:

$$\boldsymbol{\theta}_{\mathrm{ML}} = \arg\max_{\boldsymbol{\theta}} \prod_{i=1}^{m} p_{\mathrm{model}}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta})$$

This product over many probabilities can be inconvenient for a variety of reasons. For example, it is prone to numerical underflow. Also, to find the maxima/minima of this function, we can take the derivative of this function w.r.t `θ` and equate it to 0. Since we have terms in product here, we need to apply the chain rule which is quite cumbersome with products. To obtain a more convenient but equivalent optimization problem, we observe that taking the logarithm of the likelihood does not change its arg max but does conveniently transform a product into a sum and since log is a strictly increasing function (natural log function is a monotone transformation), it would not impact the resulting value of `θ` .

So we have:

$$\boldsymbol{\theta}_{\mathrm{ML}} = \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{m} \log p_{\mathrm{model}}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta})$$

Two important properties: Consistency & Efficiency

Consistency: As the number of training examples approaches infinity, the maximum likelihood estimate of a parameter converges to the true value of the parameter.

Efficiency: A way to measure how close we are to the true parameter is by the expected mean squared error, computing the squared difference between the estimated and true parameter values, where the expectation is over `m` training samples from the data generating distribution. That parametric mean squared error decreases as `m` increases, and for `m` large, the Cramér-Rao lower bound shows that no consistent estimator has a lower mean squared error than the maximum likelihood estimator.

> *For the reasons of consistency and efficiency, maximum likelihood is often considered the preferred estimator to use for machine learning.*

When the number of examples is small enough to yield over-fitting behavior, regularization strategies such as weight decay may be used to obtain a biased version of maximum likelihood that has less variance when training data is limited.

Maximum A Posteriori (MAP) Estimation

Following Bayesian approach by allowing the prior to influence the choice of the point estimate. The MAP can be used to obtain a point estimate of an unobserved quantity on the basis of empirical data. The MAP estimate chooses the point of maximal posterior probability (or maximal probability density in the more common case of continuous θ):

$$\boldsymbol{\theta}_{\mathrm{MAP}} = \arg\max_{\boldsymbol{\theta}} p(\boldsymbol{\theta} \mid \boldsymbol{x}) = \arg\max_{\boldsymbol{\theta}} \log p(\boldsymbol{x} \mid \boldsymbol{\theta}) + \log p(\boldsymbol{\theta})$$

Where on the right hand side, `log p(x|θ)` is the standard log likelihood
term, and `log p(θ)` , corresponding to the prior distribution.

As with full Bayesian inference, MAP Bayesian inference has the advantage of
leveraging information that is brought by the prior and cannot be found in the
training data. This additional information helps to reduce the variance in the
MAP point estimate (in comparison to the ML estimate). However, it does so at the price of increased bias.

Loss Functions

In most learning networks, error is calculated as the difference between the actual output `y` and the predicted output `ŷ` . The function that is used to compute this error is known as Loss Function also known as Cost function.

> *Until now our main focus has been on parameter estimating via the MLE or MAP. The reason we discussed it before is that both MLE & MAP provide a mechanism to derive the loss function.*

Let us see some commonly used loss functions.

Mean Squared Error (MSE): Mean Squared Error is one of the most common loss functions. MSE loss function is widely used in **linear regression** as the performance measure. To calculate MSE, you take the difference between your predictions and the ground truth, square it, and average it out across the whole dataset.

$$\mathrm{MSE} = \frac{1}{m}\sum_{i=1}^{m} ||\hat{y}^{(i)} - y^{(i)}||^2$$

where $y(i)$ is the actual expected output and $\hat{y}(i)$ is the model's prediction.

> *Many cost functions used in machine learning, including the MSE, can be derived from the MLE.*

To see how we can derive loss functions from MLE or MAP there is some math involved and you can skip it and move to next section.

Deriving MSE from MLE

Linear regression algorithm learns to take an input x and produce an output value $\hat{y}$. The mapping from x to $\hat{y}$ is chosen to minimize mean squared error. But how did we choose MSE as a criterion for linear regression. Let us arrive at the solution from the point of view of maximum likelihood estimation. Instead of producing a single prediction $\hat{y}$, we now think of the model as producing a conditional distribution $p(y|x)$.

We can model the linear regression problem as:

$$\hat{y} = f(x;\theta)$$
$$y \sim \mathcal{N}(y;\mu = \hat{y},\sigma^2)$$
$$p(y|x;\theta) = \frac{1}{\sigma\sqrt{2\pi}}\exp(\frac{-(y-\hat{y})^2}{2\sigma^2})$$

we are assuming that y as having Normal distribution with $\hat{y}$ as the mean of the distribution and the variance is fixed to some constant σ^2 chosen by the user. Normal distributions are a sensible choice for many applications. In the absence of prior knowledge about what form a distribution over the real numbers should take, the normal distribution is a good default choice.

Now going back to log likelihood defined earlier :

$$
\begin{aligned}
J &= \sum_{i=1}^{m} \log p(y|x;\theta) \\
&= \sum_{i=1}^{m} \log \frac{1}{\sigma\sqrt{2\pi}} \exp\left(\frac{-(y^{(i)} - \hat{y^{(i)}})^2}{2\sigma^2}\right) \\
&= \sum_{i=1}^{m} -\log(\sigma\sqrt{2\pi}) - \log\exp\left(\frac{(y^{(i)} - \hat{y^{(i)}})^2}{2\sigma^2}\right) \\
&= \sum_{i=1}^{m} -\log(\sigma) - \frac{1}{2}\log(2\pi) - \frac{(y^{(i)} - \hat{y^{(i)}})^2}{2\sigma^2} \\
&= -m\log(\sigma) - \frac{m}{2}\log(2\pi) - \sum_{i=1}^{m} \frac{(y^{(i)} - \hat{y^{(i)}})^2}{2\sigma^2} \\
&= -m\log(\sigma) - \frac{m}{2}\log(2\pi) - \sum_{i=1}^{m} \frac{\|y^{(i)} - \hat{y^{(i)}}\|^2}{2\sigma^2}
\end{aligned}
$$

where `ŷ^(i)` is the output of the linear regression on the i-th input `x^(i)` and `m` is the number of the training examples. We see that first two terms are constant so basically maximizing the log-likelihood means minimizing the MSE as:

$$
\nabla_\theta J = -\nabla_\theta \sum_{i=1}^{m} \frac{\|y^{(i)} - \hat{y^{(i)}}\|^2}{2\sigma^2}
$$

We immediately see that maximizing the log-likelihood with respect to `θ` yields the same estimate of the parameters `θ` as does minimizing the mean squared error. The two criteria have different values but the same location of the optimum. This justifies the use of the MSE as a maximum likelihood estimation procedure.

Cross-Entropy Loss (or Log Loss): Cross entropy measures the divergence between two probability distribution, if the cross entropy is large, which means that the difference between two distribution is large, while if the cross entropy is small, which means that two distribution is similar to each other.

Cross entropy is defined as:

$$
H(P, Q) = -\sum_{x} P(x) \log Q(x)
$$

where *P* is the distribution of the true labels, and *Q* is the probability distribution of the predictions from the model. *It can be also shown that*

cross entropy loss can be as well derived from MLE, I will not bore you with more math.

Let us further simplify this for our model with:

- N—number of observations
- M—number of possible class labels (dog, cat, fish)
- y—a binary indicator (0 or 1) of whether class label `c` is the correct classification for observation `o`
- p—the model's predicted probability that observation

Binary Classification

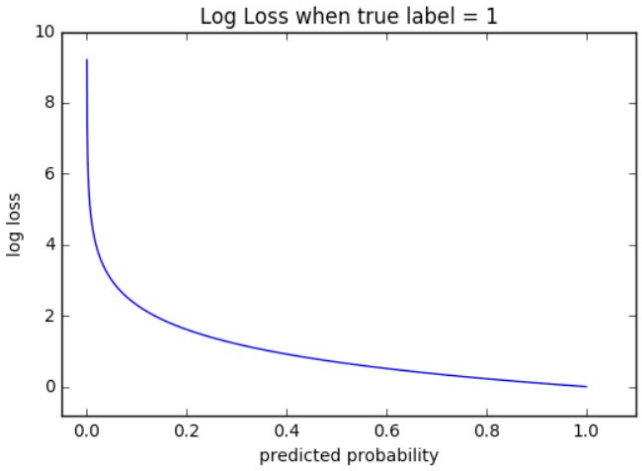
In binary classification (M=2), the formula equals:

$$CE = -(y\log(p) + (1-y)\log(1-p))$$

In case of a binary classification each predicted probability is compared to the actual class output value (0 or 1) and a score is calculated that penalizes the probability based on the distance from the expected value.

Visualization

The graph below shows the range of possible log loss values given a true observation (y= 1). As the predicted probability approaches 1, log loss slowly decreases. As the predicted probability decreases, however, the log loss increases rapidly.



Log loss penalizes both types of errors, but especially those predictions that are confident and wrong!

Multi-class Classification

In multi-class classification (M>2), we take the sum of log loss values for each class prediction in the observation.

$$CE = -\sum_{c=1}^{M} y_{o,c} \log(p_{o,c})$$

Cross-entropy for a binary or two class prediction problem is actually calculated as the average cross entropy across all examples. Log Loss uses negative log to provide an easy metric for comparison. It takes this approach because the positive log of numbers < 1 returns negative values, which is confusing to work with when comparing the performance of two models. See this post for detailed discussion on cross-entropy loss.

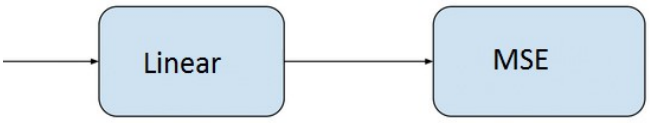
ML problems and corresponding Loss functions

Let us see what are commonly used output layers and loss function in machine learning models:

Regression Problem

A problem where you predict a real-value quantity.

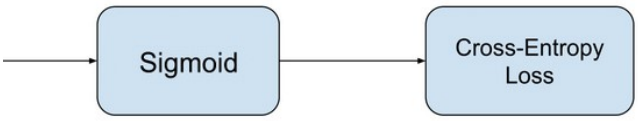
- **Output Layer Configuration**: One node with a linear activation unit.
- **Loss Function**: Mean Squared Error (MSE).



Binary Classification Problem

A problem where you classify an example as belonging to one of two classes. The problem is framed as predicting the likelihood of an example belonging to class one, e.g. the class that you assign the integer value 1, whereas the other class is assigned the value 0.

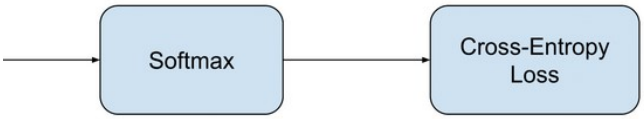
- **Output Layer Configuration**: One node with a sigmoid activation unit.
- **Loss Function**: Cross-Entropy, also referred to as Logarithmic loss.



Multi-Class Classification Problem

A problem where you classify an example as belonging to one of more than two classes. The problem is framed as predicting the likelihood of an example belonging to each class.

- **Output Layer Configuration**: One node for each class using the softmax activation function.
- **Loss Function**: Cross-Entropy, also referred to as Logarithmic loss.



Having discussed estimator and various loss functions let us understand the role of optimizers in ML algorithms.

Optimizers

To minimize the prediction *error or loss*, the model while experiencing the examples of the training set, updates the model parameters w. These error calculations when plotted against the w is also called **cost function plot** $J(w)$, since it determines the cost/penalty of the model. So minimizing the error is also called as minimization the cost function.

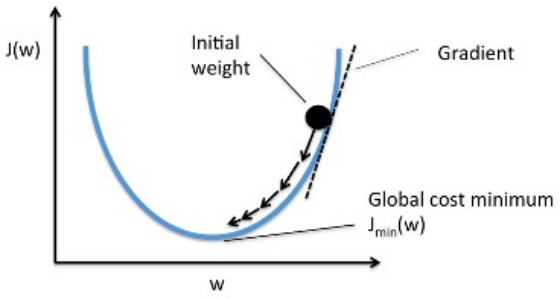
But how exactly do you do that? Using **optimizers**.

> *Optimizers are used to update weights and biases i.e. the internal parameters of a model to reduce the error.*

The most important technique and the foundation of how we train and optimize our model is using **Gradient Descent**.

Gradient Descent :

When we plot the cost function $J(w)$ *vs* w. It is represented as below:



As we see from the curve, there exists a value of parameters `w` which has the minimum cost `Jmin` . Now we need to find a way to reach this minimum cost.

In the gradient descent algorithm, we start with random model parameters and calculate the error for each learning iteration, keep updating the model parameters to move closer to the values that results in minimum cost.

repeat until minimum cost: {

$$w_j = w_j - \alpha \partial / \partial w_j J(W)$$

}

In the above equation we are updating the model parameters after each iteration. The second term of the equation calculates the slope or gradient of the curve at each iteration.

The gradient of the cost function is calculated as partial derivative of cost function `J` with respect to each model parameter `Wj` , `j` takes value of number of features `[1 to n]` . `α` , *alpha*, is the learning rate, or how quickly we want to move towards the minimum. If α is too large, we can overshoot. If `α` is too small, means small steps of learning hence the overall time taken by the model to observe all examples will be more.

There are three ways of doing gradient descent:

Batch gradient descent: Uses all of the training instances to update the model parameters in each iteration.

Mini-batch Gradient Descent: Instead of using all examples, Mini-batch Gradient Descent divides the training set into smaller size called batch denoted by 'b'. Thus a mini-batch 'b' is used to update the model parameters in each iteration.

Stochastic Gradient Descent (SGD): updates the parameters using only a single training instance in each iteration. The training instance is usually selected randomly. Stochastic gradient descent is often preferred to optimize cost functions when there are hundreds of

thousands of training instances or more, as it will converge more quickly than batch gradient descent .

Some other commonly used optimizers:

Adagrad

Adagrad adapts the learning rate specifically to individual features: that means that some of the weights in your dataset will have different learning rates than others. This works really well for sparse datasets where a lot of input examples are missing. Adagrad has a major issue though: the adaptive learning rate tends to get really small over time. Some other optimizers below seek to eliminate this problem.

RMSprop

RMSprop is a special version of Adagrad developed by Professor Geoffrey Hinton in his neural nets class. Instead of letting all of the gradients accumulate for momentum, it only accumulates gradients in a fixed window. RMSprop is similar to Adaprop, which is another optimizer that seeks to solve some of the issues that Adagrad leaves open.

Adam

Adam stands for adaptive moment estimation, and is another way of using past gradients to calculate current gradients. Adam also utilizes the concept of momentum by adding fractions of previous gradients to the current one. This optimizer has become pretty widespread, and is practically accepted for use in training neural nets.

I have just presented brief overview of the these optimizers, please refer to this post for detailed analysis on various optimizers.

I hope now you understand what happens underneath when you write:

```
# loss function: Binary Cross-entropy and optimizer: Adam
model.compile(loss='binary_crossentropy', optimizer='adam')
```

or

```
# loss function: MSE and optimizer: stochastic gradient
descent
model.compile(loss='mean_squared_error', optimizer='sgd')
```

Thanks for reading.

References:

[1] https://www.deeplearningbook.org/contents/ml.html

[2] https://machinelearningmastery.com/loss-and-loss-functions-for-training-deep-learning-neural-networks/

[3] https://blog.algorithmia.com/introduction-to-optimizers/

[4] https://jhui.github.io/2017/01/05/Deep-learning-Information-theory/

[5] https://blog.algorithmia.com/introduction-to-loss-functions/

[6] https://gombru.github.io/2018/05/23/cross_entropy_loss/

[7] https://www.kdnuggets.com/2018/04/right-metric-evaluating-machine-learning-models-1.html

[8] https://rohanvarma.me/Loss-Functions/

[9] http://blog.christianperone.com/2019/01/mle/