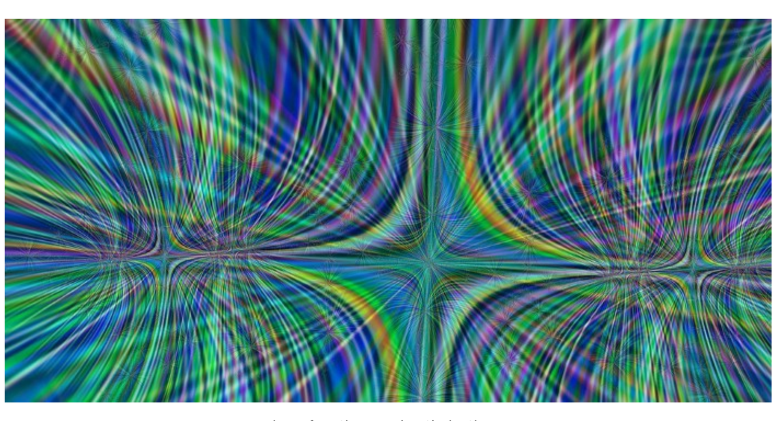
# Common Loss functions in machine learning

Ravindra Parmar Follow
Sep 2, 2018 · 5 min read



Loss functions and optimizations

Machines learn by means of a loss function. It's a method of evaluating how well specific algorithm models the given data. If predictions deviates too much from actual results, loss function would cough up a very large number. Gradually, with the help of some optimization function, loss function learns to reduce the error in prediction. In this article we will go through several loss functions and their applications in the domain of machine/deep learning.

There's no one-size-fits-all loss function to algorithms in machine learning. There are various factors involved in choosing a loss function for specific problem such as type of machine learning algorithm chosen, ease of calculating the derivatives and to some degree the percentage of outliers in the data set.

Top highlight

Broadly, loss functions can be classified into two major categories depending upon the type of learning task we are dealing with — **Regression losses** and **Classification losses**. In classification, we are trying to predict output from set of finite categorical values i.e Given large data set of images of hand written digits, categorizing them into one of 0–9 digits. Regression, on the other hand, deals with predicting a continuous value for example given floor area, number of rooms, size of rooms, predict the price of room.

```
NOTE
        n        - Number of training examples.
        i        - ith training example in a data set.
        y(i)     - Ground truth label for ith training example.
        y_hat(i) - Prediction for ith training example.
```

## Regression Losses

### Mean Square Error/Quadratic Loss/L2 Loss

***Mathematical formulation*** :-

$$MSE = \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{n}$$

Mean Squared Error

As the name suggests, *Mean square error* is measured as the average of squared difference between predictions and actual observations. It's only concerned with the average magnitude of error irrespective of their direction. However, due to squaring, predictions which are far away from actual values are penalized heavily in comparison to less deviated predictions. Plus MSE has nice mathematical properties which makes it easier to calculate gradients.

```
import numpy as np

y_hat = np.array([0.000, 0.166, 0.333])
y_true = np.array([0.000, 0.254, 0.998])

def rmse(predictions, targets):
    differences = predictions - targets
    differences_squared = differences ** 2
    mean_of_differences_squared = differences_squared.mean()
    rmse_val = np.sqrt(mean_of_differences_squared)
    return rmse_val

print("d is: " + str(["%.8f" % elem for elem in y_hat]))
print("p is: " + str(["%.8f" % elem for elem in y_true]))

rmse_val = rmse(y_hat, y_true)
print("rms error is: " + str(rmse_val))
```

### Mean Absolute Error/L1 Loss

***Mathematical formulation*** :-

$$MAE = \frac{\sum_{i=1}^{n}\mid y_i - \hat{y}_i \mid}{n}$$

Mean absolute error

*Mean absolute error*, on the other hand, is measured as the average of sum of absolute differences between predictions and actual observations. Like MSE, this as well measures the magnitude of error without considering their direction. Unlike MSE, MAE needs more complicated tools such as linear programming to compute the gradients. Plus MAE is more robust to outliers since it does not make use of square.

```
import numpy as np

y_hat = np.array([0.000, 0.166, 0.333])
y_true = np.array([0.000, 0.254, 0.998])

print("d is: " + str(["%.8f" % elem for elem in y_hat]))
print("p is: " + str(["%.8f" % elem for elem in y_true]))

def mae(predictions, targets):
    differences = predictions - targets
    absolute_differences = np.absolute(differences)
    mean_absolute_differences = absolute_differences.mean()
    return mean_absolute_differences

mae_val = mae(y_hat, y_true)
print ("mae error is: " + str(mae_val))
```

### Mean Bias Error

This is much less common in machine learning domain as compared to it's counterpart. This is same as MSE with the only difference that we don't take absolute values. Clearly there's a need for caution as positive and negative errors could cancel each other out. Although less accurate in practice, it could determine if the model has positive bias or negative bias.

***Mathematical formulation*** :-

$$MBE = \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)}{n}$$

Mean bias error

## Classification Losses

### Hinge Loss/Multi class SVM Loss

In simple terms, the score of correct category should be greater than sum of scores of all incorrect categories by some safety margin (usually one). And hence hinge loss is used for maximum-margin classification, most notably for support vector machines. Although not differentiable, it's a convex function which makes it easy to work with usual convex optimizers used in machine learning domain.

***Mathematical formulation*** :-

$$SVMLoss = \sum_{j \neq y_i} max(0, s_j - s_{y_i} + 1)$$

SVM Loss or Hinge Loss

Consider an example where we have three training examples and three classes to predict — Dog, cat and horse. Below the values predicted by our algorithm for each of the classes :-

| | Image #1 | Image #2 | Image #3 |
|---|---|---|---|
| Dog | -0.39 | -4.61 | 1.03 |
| Cat | 1.49 | 3.28 | -2.37 |
| Horse | 4.21 | 1.46 | -2.27 |

Hinge loss/ Multi class SVM loss

Computing hinge losses for all 3 training examples :-

```
## 1st training example
max(0, (1.49) - (-0.39) + 1) + max(0, (4.21) - (-0.39) + 1)
max(0, 2.88) + max(0, 5.6)
2.88 + 5.6
8.48 (High loss as very wrong prediction)

## 2nd training example
max(0, (-4.61) - (3.28)+ 1) + max(0, (1.46) - (3.28)+ 1)
max(0, -6.89) + max(0, -0.82)
0 + 0
0 (Zero loss as correct prediction)

## 3rd training example
max(0, (1.03) - (-2.27)+ 1) + max(0, (-2.37) - (-2.27)+ 1)
max(0, 4.3) + max(0, 0.9)
4.3 + 0.9
5.2 (High loss as very wrong prediction)
```

### Cross Entropy Loss/Negative Log Likelihood

This is the most common setting for classification problems. Cross-entropy loss increases as the predicted probability diverges from the actual label.

***Mathematical formulation*** :-

$$CrossEntropyLoss = -(y_i log(\hat{y}_i) + (1 - y_i) log(1 - \hat{y}_i))$$

Cross entropy loss

Notice that when actual label is 1 (y(i) = 1), second half of function disappears whereas in case actual label is 0 (y(i) = 0) first half is dropped off. In short, we are just multiplying the log of the actual predicted probability for the ground truth class. An important aspect of this is that cross entropy loss penalizes heavily the predictions that are *confident but wrong*.

```
import numpy as np

predictions = np.array([[0.25,0.25,0.25,0.25],
                        [0.01,0.01,0.01,0.96]])
targets = np.array([[0,0,0,1],
                   [0,0,0,1]])

def cross_entropy(predictions, targets, epsilon=1e-10):
    predictions = np.clip(predictions, epsilon, 1. - epsilon)
    N = predictions.shape[0]
    ce_loss = -np.sum(np.sum(targets * np.log(predictions + 1e-
5)))/N
    return ce_loss

cross_entropy_loss = cross_entropy(predictions, targets)
print ("Cross entropy loss is: " + str(cross_entropy_loss))
```



WRITTEN BY
Ravindra Parmar
Follow
Software professional. Machine/Deep Learning Enthusiast. An active member of https://stackoverflow.com/users/4047092/ravi