

# A Detailed Guide to 7 Loss Functions for Machine Learning Algorithms with Python Code

[INTERMEDIATE](#)[LISTICLE](#)[MACHINE LEARNING](#)[PYTHON](#)[STATISTICS](#)

## Overview

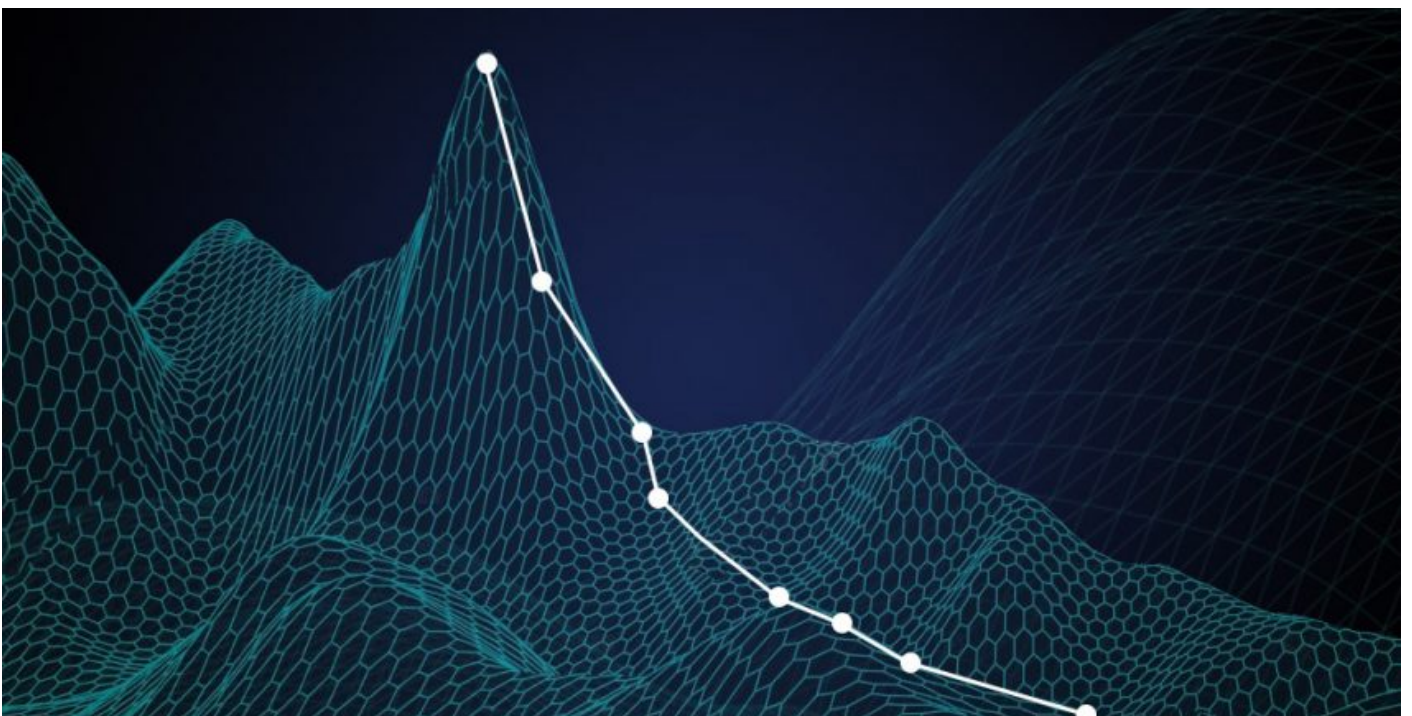
- What are loss functions? And how do they work in machine learning algorithms? Find out in this article
- Loss functions are actually at the heart of these techniques that we regularly use
- This article covers multiple loss functions, where they work, and how you can code them in Python

## Introduction

Picture this – you’ve trained a machine learning model on a given dataset and are ready to put it in front of your client. But how can you be sure that this model will give the optimum result? Is there a metric or a technique that will help you quickly evaluate your model on the dataset?

Yes – and that, in a nutshell, is where loss functions come into play in machine learning.

Loss functions are at the heart of the [machine learning algorithms](#) we love to use. But I’ve seen the majority of beginners and enthusiasts become quite confused regarding how and where to use them.



They're not difficult to understand and will enhance your understanding of machine learning algorithms infinitely. So, what are loss functions and how can you grasp their meaning?

In this article, I will discuss 7 common loss functions used in [machine learning](#) and explain where each of them is used. We have a lot to cover in this article so let's begin!

*Loss functions are one part of the entire machine learning journey you will take. Here's the perfect course to help you get started and make you industry-ready:*

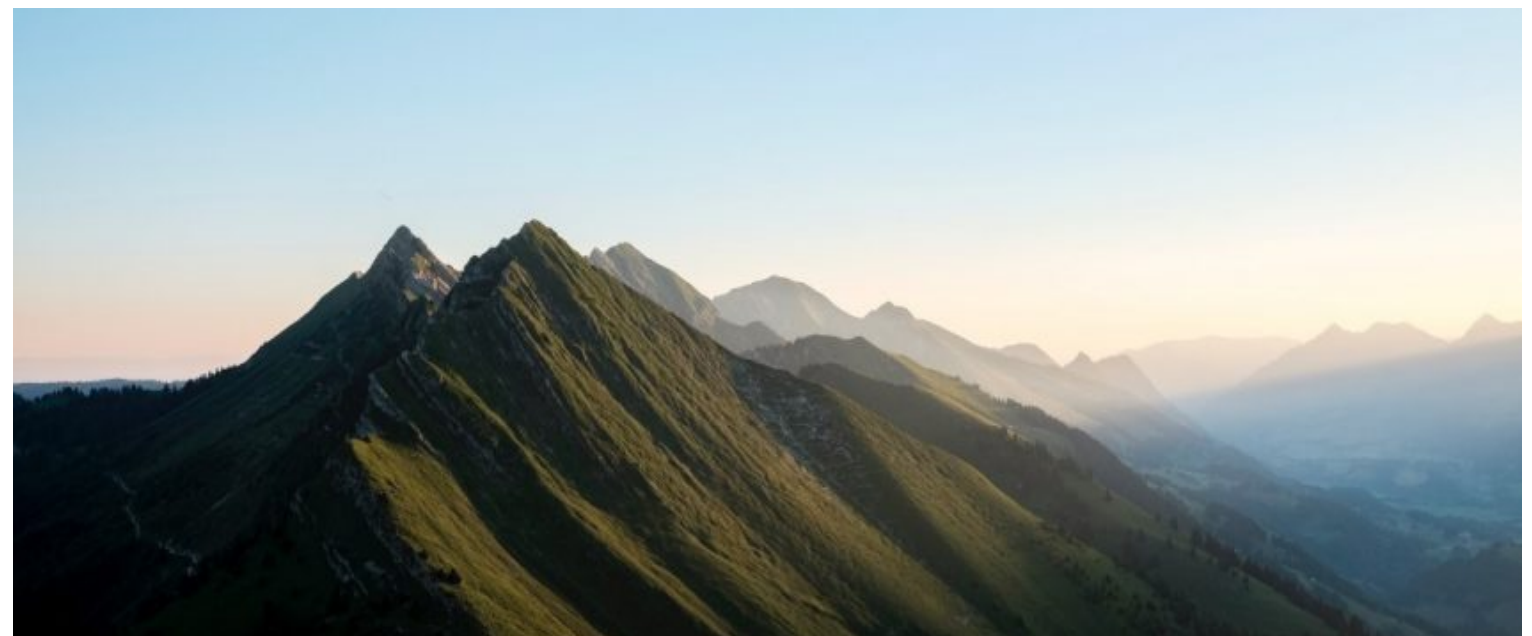
- [Applied Machine Learning – Beginner to Professional](#)

## Table of Contents

- What are Loss Functions?
- Regression Loss Functions
  - Squared Error Loss
  - Absolute Error Loss
  - Huber Loss
- Binary Classification Loss Functions
  - Binary Cross-Entropy
  - Hinge Loss
- Multi-class Classification Loss Functions
  - Multi-class Cross Entropy Loss
  - Kullback Leibler Divergence Loss

## What are Loss Functions?

Let's say you are on the top of a hill and need to climb down. How do you decide where to walk towards?



Here's what I would do:

- Look around to see all the possible paths
- Reject the ones going up. This is because these paths would actually cost me more energy and make my task even more difficult
- Finally, take the path that I **think** has the most slope downhill

This intuition that I just judged my decisions against? This is exactly what a loss function provides.

A loss function maps decisions to their associated costs.

Deciding to go up the slope will cost us energy and time. Deciding to go down will benefit us. Therefore, it has a negative cost.

In [supervised machine learning](#) algorithms, we want to **minimize** the error for each training example during the learning process. This is done using some optimization strategies like [gradient descent](#). *And this error comes from the loss function.*

## What's the Difference between a Loss Function and a Cost Function?

I want to emphasize this here – although **cost function** and **loss function** are synonymous and used interchangeably, they are different.

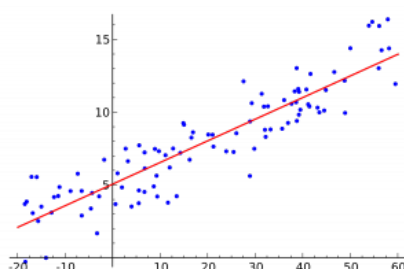
A loss function is for a single training example. It is also sometimes called an **error function**. A cost function, on the other hand, is the **average loss** over the entire training dataset. The optimization strategies aim at minimizing the cost function.

## Regression Loss Functions

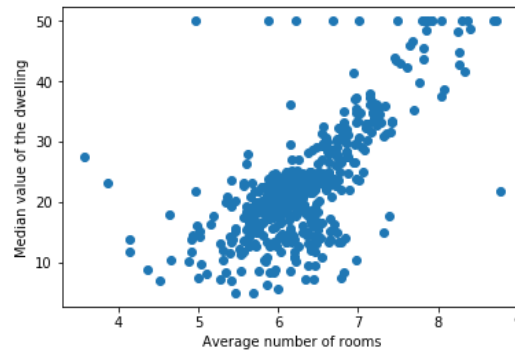
You must be quite familiar with linear regression at this point. It deals with modeling a linear relationship between a **dependent variable**, Y, and several **independent variables**, X<sub>i</sub>'s. Thus, we essentially fit a line in space on these variables.

$$Y = a_0 + a_1 * X_1 + a_2 * X_2 + \dots + a_n * X_n$$

We will use the given data points to find the coefficients a<sub>0</sub>, a<sub>1</sub>, ..., a<sub>n</sub>.



We will use the famous [Boston Housing Dataset](#) for understanding this concept. And to keep things simple, we will use only one feature – the *Average number of rooms per dwelling* (X) – to predict the dependent variable – *Median Value* (Y) of houses in '\$1000's'.



We will use **Gradient Descent** as an optimization strategy to find the regression line. I will not go into the intricate details about Gradient Descent, but here is a reminder of the Weight Update Rule:

Repeat until convergence {

$$\theta_j \leftarrow \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

}

Source: Hackernoon.com

Here,  $\theta_j$  is the weight to be updated,  $\alpha$  is the learning rate and  $J$  is the cost function. The cost function is parameterized by  $\theta$ . **Our aim is to find the value of  $\theta$  which yields minimum overall cost.**

You can get an in-depth explanation of Gradient Descent and how it works [here](#).

I have defined the steps that we will follow for each loss function below:

1. Write the expression for our predictor function,  $f(X)$  and identify the parameters that we need to find
2. Identify the loss to use for each training example
3. Find the expression for the Cost Function – the average loss on all examples
4. Find the gradient of the Cost Function with respect to each unknown parameter
5. Decide on the learning rate and run the weight update rule for a fixed number of iterations

## 1. Squared Error Loss

Squared Error loss for each training example, also known as **L2 Loss**, is the square of the difference between the actual and the predicted values:

$$L = (y - f(x))^2$$

The corresponding cost function is the **Mean** of these **Squared Errors (MSE)**.

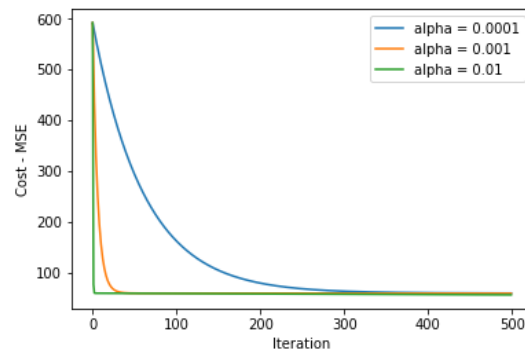
I encourage you to try and find the gradient for gradient descent yourself before referring to the code below.

```
1 def update_weights_MSE(m, b, X, Y, learning_rate):
2     m_deriv = 0
3     b_deriv = 0
4     N = len(X)
5     for i in range(N):
6         # Calculate partial derivatives
7         # -2x(y - (mx + b))
8         m_deriv += -2*X[i] * (Y[i] - (m*X[i] + b))
9
10        # -2(y - (mx + b))
11        b_deriv += -2*(Y[i] - (m*X[i] + b))
12
13    # We subtract because the derivatives point in direction of steepest ascent
14    m -= (m_deriv / float(N)) * learning_rate
15    b -= (b_deriv / float(N)) * learning_rate
16
17    return m, b
```

[view raw](#)

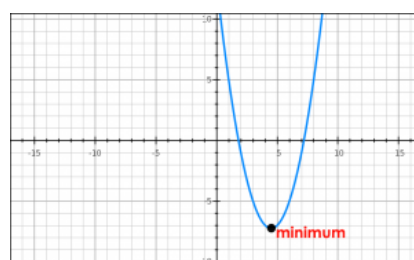
weight\_update\_MSE.py hosted with ♥ by GitHub

I used this code on the Boston data for different values of the learning rate for 500 iterations each:



*Here's a task for you. Try running the code for a learning rate of 0.1 again for 500 iterations. Let me know your observations and any possible explanations in the comments section.*

Let's talk a bit more about the MSE loss function. It is a positive quadratic function (of the form  $ax^2 + bx + c$  where  $a > 0$ ). Remember how it looks graphically?



A quadratic function only has a global minimum. Since there are no local minima, we will never get stuck in one. Hence, it is always guaranteed that Gradient Descent will converge (*if it converges at all*) to the global minimum.

The MSE loss function penalizes the model for making large errors by squaring them. Squaring a large quantity makes it even larger, right? But there's a caveat. This property makes the MSE cost function less robust to outliers. Therefore, ***it should not be used if our data is prone to many outliers.***

## 2. Absolute Error Loss

Absolute Error for each training example is the distance between the predicted and the actual values, irrespective of the sign. Absolute Error is also known as the **L1 loss**:

$$L = |y - f(x)|$$

As I mentioned before, the cost is the **Mean** of these **Absolute Errors (MAE)**.

***The MAE cost is more robust to outliers as compared to MSE.*** However, handling the absolute or modulus operator in mathematical equations is not easy. I'm sure a lot of you must agree with this! We can consider this as a disadvantage of MAE.

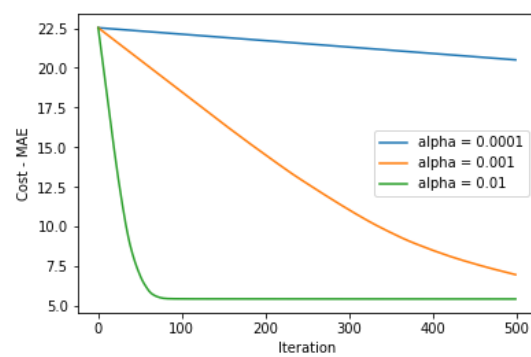
Here is the code for the *update\_weight* function with MAE cost:

```
1 def update_weights_MAE(m, b, X, Y, learning_rate):
2     m_deriv = 0
3     b_deriv = 0
4     N = len(X)
5     for i in range(N):
6         # Calculate partial derivatives
7         # -x(y - (mx + b)) / |mx + b|
8         m_deriv += - X[i] * (Y[i] - (m*X[i] + b)) / abs(Y[i] - (m*X[i] + b))
9
10        # -(y - (mx + b)) / |mx + b|
11        b_deriv += -(Y[i] - (m*X[i] + b)) / abs(Y[i] - (m*X[i] + b))
12
13    # We subtract because the derivatives point in direction of steepest ascent
14    m -= (m_deriv / float(N)) * learning_rate
15    b -= (b_deriv / float(N)) * learning_rate
16
17    return m, b
```

[view raw](#)

weight\_update\_MAE.py hosted with ♥ by GitHub

We get the below plot after running the code for 500 iterations with different learning rates:



### 3. Huber Loss

The Huber loss combines the best properties of MSE and MAE. It is quadratic for smaller errors and is linear otherwise (and similarly for its gradient). It is identified by its *delta* parameter:

$$L_{\delta} = \begin{cases} \frac{1}{2}(y - f(x))^2, & \text{if } |y - f(x)| \leq \delta \\ \delta|y - f(x)| - \frac{1}{2}\delta^2, & \text{otherwise} \end{cases}$$

← Quadratic  
← Linear

```

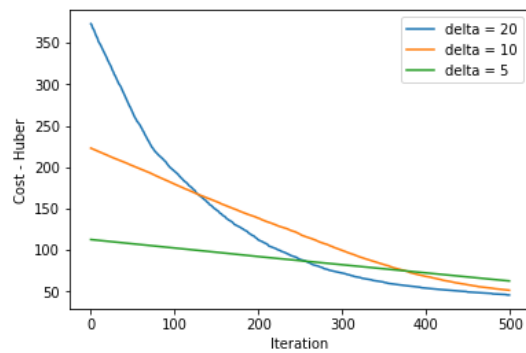
1  def update_weights_Huber(m, b, X, Y, delta, learning_rate):
2      m_deriv = 0
3      b_deriv = 0
4      N = len(X)
5      for i in range(N):
6          # derivative of quadratic for small values and of linear for large values
7          if abs(Y[i] - m*X[i] - b) <= delta:
8              m_deriv += -X[i] * (Y[i] - (m*X[i] + b))
9              b_deriv += - (Y[i] - (m*X[i] + b))
10         else:
11             m_deriv += delta * X[i] * ((m*X[i] + b) - Y[i]) / abs((m*X[i] + b) - Y[i])
12             b_deriv += delta * ((m*X[i] + b) - Y[i]) / abs((m*X[i] + b) - Y[i])
13
14     # We subtract because the derivatives point in direction of steepest ascent
15     m -= (m_deriv / float(N)) * learning_rate
16     b -= (b_deriv / float(N)) * learning_rate
17
18     return m, b

```

[view raw](#)

weight\_update\_Huber.py hosted with ♥ by GitHub

We obtain the below plot for 500 iterations of weight update at a learning rate of 0.0001 for different values of the *delta* parameter:



Huber loss is more robust to outliers than MSE. *It is used in [Robust Regression](#), [M-estimation](#) and [Additive Modelling](#). A variant of Huber Loss is also used in classification.*

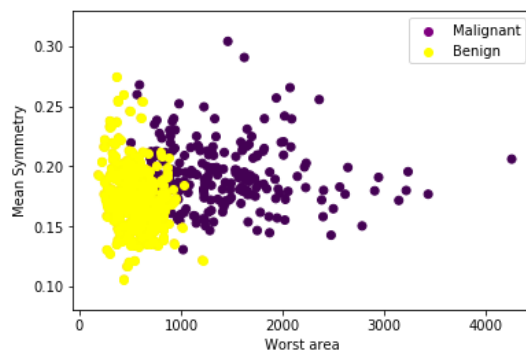
## Binary Classification Loss Functions

The name is pretty self-explanatory. Binary Classification refers to assigning an object into one of two classes. This classification is based on a rule applied to the input feature vector. For example, classifying an email as *spam* or *not spam* based on, say its subject line, is binary classification.

I will illustrate these binary classification loss functions on the [Breast Cancer dataset](#).

*We want to classify a tumor as ‘**Malignant**’ or ‘**Benign**’ based on features like average radius, area, perimeter, etc. For simplification, we will use only two input features ( $X_1$  and  $X_2$ ) namely ‘**worst area**’ and ‘**mean symmetry**’ for classification. The target value  $Y$  can be 0 (Malignant) or 1 (Benign).*

Here is a scatter plot for our data:



## 1. Binary Cross Entropy Loss

**Let us start by understanding the term ‘entropy’.** Generally, we use entropy to indicate disorder or uncertainty. It is measured for a random variable  $X$  with probability distribution  $p(X)$ :



$$S = \begin{cases} - \int p(x) \cdot \log p(x) \cdot dx, & \text{if } x \text{ is continuous} \\ - \sum_x p(x) \cdot \log p(x), & \text{if } x \text{ is discrete} \end{cases}$$

The negative sign is used to make the overall quantity positive.

A greater value of entropy for a probability distribution indicates a greater uncertainty in the distribution. Likewise, a smaller value indicates a more certain distribution.

This makes binary cross-entropy suitable as a loss function – **you want to minimize its value**. We use **binary cross-entropy** loss for classification models which output a probability  $p$ .

Probability that the element belongs to class 1 (or positive class) =  $p$  Then, the probability that the element belongs to class 0 (or negative class) =  $1 - p$

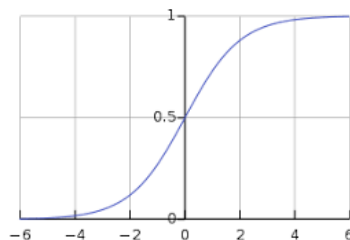
Then, the cross-entropy loss for output label  $y$  (can take values 0 and 1) and predicted probability  $p$  is defined as:

$$L = -y * \log(p) - (1 - y) * \log(1 - p) = \begin{cases} -\log(1 - p), & \text{if } y = 0 \\ -\log(p), & \text{if } y = 1 \end{cases}$$

This is also called **Log-Loss**. To calculate the probability  $p$ , we can use the sigmoid function. Here,  $z$  is a function of our input features:

$$S(z) = \frac{1}{1 + e^{-z}}$$

The range of the sigmoid function is  $[0, 1]$  which makes it suitable for calculating probability.



Try to find the gradient yourself and then look at the code for the *update\_weight* function below.

```
1 def update_weights_BCE(m1, m2, b, X1, X2, Y, learning_rate):
2     m1_deriv = 0
3     m2_deriv = 0
4     b_deriv = 0
5     N = len(X1)
6     for i in range(N):
7         s = 1 / (1 + (1 / (1 + math.exp(-m1*X1[i] - m2*X2[i] - b))))
```

```

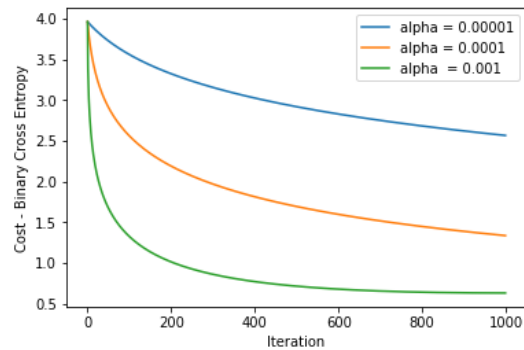
9         # Calculate partial derivatives
10        m1_deriv += -X1[i] * (s - Y[i])
11        m2_deriv += -X2[i] * (s - Y[i])
12        b_deriv += -(s - Y[i])
13
14        # We subtract because the derivatives point in direction of steepest ascent
15        m1 -= (m1_deriv / float(N)) * learning_rate
16        m2 -= (m2_deriv / float(N)) * learning_rate
17        b -= (b_deriv / float(N)) * learning_rate
18
19    return m1, m2, b

```

[view raw](#)

weight\_update\_BCE.py hosted with ♥ by GitHub

I got the below plot on using the weight update rule for 1000 iterations with different values of alpha:



## 2. Hinge Loss

**Hinge loss is primarily used with [Support Vector Machine \(SVM\)](#) Classifiers with class labels -1 and 1.** So make sure you change the label of the 'Malignant' class in the dataset from 0 to -1.

Hinge Loss not only penalizes the wrong predictions but also the right predictions that are not confident.

Hinge loss for an input-output pair (x, y) is given as:

$$L = \max(0, 1 - y * f(x))$$

```

1  def update_weights_Hinge(m1, m2, b, X1, X2, Y, learning_rate):
2      m1_deriv = 0
3      m2_deriv = 0
4      b_deriv = 0
5      N = len(X1)
6      for i in range(N):
7          # Calculate partial derivatives
8          if Y[i]*(m1*X1[i] + m2*X2[i] + b) <= 1:
9              m1_deriv += -X1[i] * Y[i]
10             m2_deriv += -X2[i] * Y[i]
11             b_deriv += -Y[i]
12             # else derivatives are zero
13

```

```

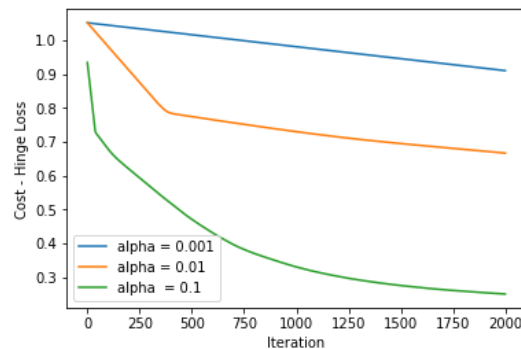
14 # We subtract because the derivatives point in direction of steepest ascent
15 m1 -= (m1_deriv / float(N)) * learning_rate
16 m2 -= (m2_deriv / float(N)) * learning_rate
17 b -= (b_deriv / float(N)) * learning_rate
18
19 return m1, m2, b

```

weight\_update\_Hinge.py hosted with ♥ by GitHub

[view raw](#)

After running the update function for 2000 iterations with three different values of alpha, we obtain this plot:



**Hinge Loss simplifies the mathematics for SVM while maximizing the loss (as compared to Log-Loss). It is used when we want to make real-time decisions with not a laser-sharp focus on accuracy.**

## Multi-Class Classification Loss Functions

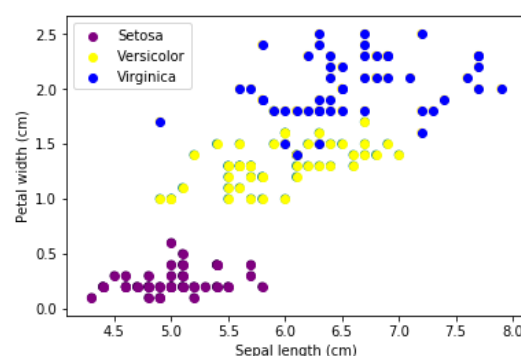
Emails are not just classified as spam or not spam (this isn't the 90s anymore!). They are classified into various other categories – Work, Home, Social, Promotions, etc. This is a [Multi-Class Classification](#) use case.

We'll use the [Iris Dataset](#) for understanding the remaining two loss functions. We will use 2 features X<sub>1</sub>, **Sepal length** and feature X<sub>2</sub>, **Petal width**, to predict the class (Y) of the Iris flower – **Setosa, Versicolor or Virginica**

Our task is to implement the classifier using a neural network model and the in-built Adam optimizer in Keras. This is because as the number of parameters increases, the math, as well as the code, will become difficult to comprehend.

If you are new to Neural Networks, I highly recommend reading [this article](#) first.

Here is the scatter plot for our data:



# 1. Multi-Class Cross Entropy Loss

The multi-class cross-entropy loss is a generalization of the Binary Cross Entropy loss. The loss for input vector  $X_i$  and the corresponding one-hot encoded target vector  $Y_i$  is:

$$L(X_i, Y_i) = - \sum_{j=1}^c y_{ij} * \log(p_{ij})$$

where  $Y_i$  is one – hot encoded target vector  $(y_{i1}, y_{i2}, \dots, y_{ic})$ ,

$$y_{ij} = \begin{cases} 1, & \text{if } i_{th} \text{ element is in class } j \\ 0, & \text{otherwise} \end{cases}$$

$p_{ij} = f(X_i) = \text{Probability that } i_{th} \text{ element is in class } j$

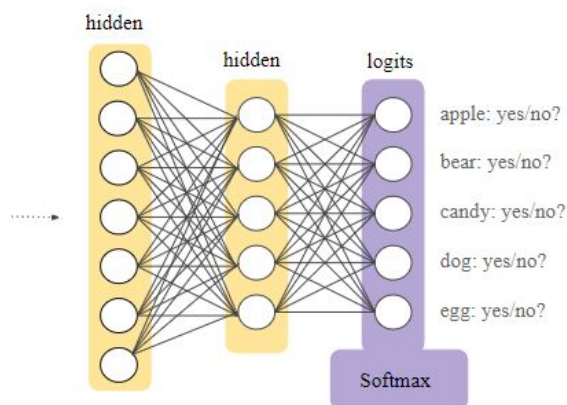
We use the softmax function to find the probabilities  $p_{ij}$ :

The standard (unit) softmax function  $\sigma : \mathbb{R}^K \rightarrow \mathbb{R}^K$  is defined by the formula

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^K e^{z_j}} \quad \text{for } i = 1, \dots, K \text{ and } \mathbf{z} = (z_1, \dots, z_K) \in \mathbb{R}^K$$

Source: Wikipedia

“Softmax is implemented through a neural network layer just before the output layer. The Softmax layer must have the same number of nodes as the output layer.” Google Developer’s Blog



Finally, our output is the class with the maximum probability for the given input.

We build a model using an input layer and an output layer and compile it with different learning rates. Specify the loss parameter as **'categorical\_crossentropy'** in the model.compile() statement:

```
1 # importing requirements
2 from keras.layers import Dense
3 from keras.models import Sequential
4 from keras.optimizers import adam
5
```

```

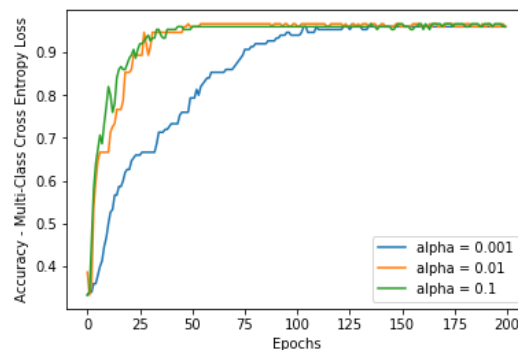
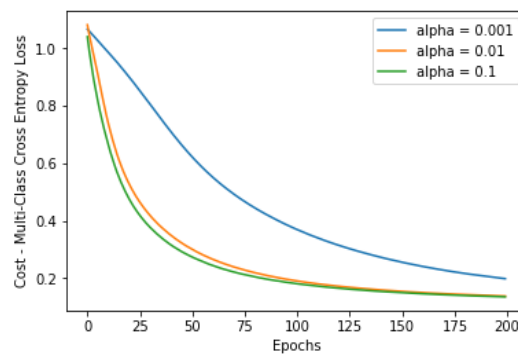
6 # alpha = 0.001 as given in the lr parameter in adam() optimizer
7
8 # build the model
9 model_alpha1 = Sequential()
10 model_alpha1.add(Dense(50, input_dim=2, activation='relu'))
11 model_alpha1.add(Dense(3, activation='softmax'))
12
13 # compile the model
14 opt_alpha1 = adam(lr=0.001)
15 model_alpha1.compile(loss='categorical_crossentropy', optimizer=opt_alpha1, metrics=['accuracy'])
16
17 # fit the model
18 # dummy_Y is the one-hot encoded
19 # history_alpha1 is used to score the validation and accuracy scores for plotting
20 history_alpha1 = model_alpha1.fit(dataX, dummy_Y, validation_data=(dataX, dummy_Y), epochs=200, verbose=0)

```

[view raw](#)

model\_MCE.py hosted with ❤ by GitHub

Here are the plots for cost and accuracy respectively after training for 200 epochs:



## 2. KL-Divergence

The **Kullback-Liebler Divergence** is a measure of how a probability distribution differs from another distribution. A KL-divergence of zero indicates that the distributions are identical.

For probability distributions  $P$  and  $Q$ ,  
 $KL$  – Divergence of  $P$  from  $Q$  is given by

$$D_{KL}(P||Q) = \begin{cases} -\sum_x P(x) \cdot \log \frac{Q(x)}{P(x)} = \sum_x P(x) \cdot \log \frac{P(x)}{Q(x)}, & \text{for discrete distributions} \\ -\int P(x) \cdot \log \frac{Q(x)}{P(x)} \cdot dx = \int P(x) \cdot \log \frac{P(x)}{Q(x)} \cdot dx, & \text{for continuous distributions} \end{cases}$$

= Expectation of logarithmic difference between  $P$  and  $Q$  with respect to  $P$

Notice that the divergence function is not symmetric.

$$D_{KL}(P||Q) \neq D_{KL}(Q||P)$$

This is why  $KL$ -Divergence cannot be used as a distance metric.

I will describe the basic approach of using  $KL$ -Divergence as a loss function without getting into its math. We want to approximate the **true** probability distribution  $P$  of our target variables with respect to the input features, given some **approximate** distribution  $Q$ . Since  $KL$ -Divergence is not symmetric, we can do this in two ways:

1. Minimizing the **forward KL**,  $D_{KL}(P||Q)$
2. Minimizing the **backward KL**,  $D_{KL}(Q||P)$

The first approach is used in Supervised learning, the second in Reinforcement Learning.  $KL$ -Divergence is functionally similar to multi-class cross-entropy and is also called relative entropy of  $P$  with respect to  $Q$ :

$$D_{KL}(P||Q) = -\sum_x (P(x) \cdot \log Q(x) - P(x) \cdot \log P(x)) = H(P, Q) - H(P, P)$$

$H(P, P)$  is the entropy of  $P$

$H(P, Q)$  is the cross – entropy of  $P$  and  $Q$

We specify the '**kullback\_leibler\_divergence**' as the value of the loss parameter in the `compile()` function as we did before with the multi-class cross-entropy loss.

```
1 # importing requirements
2 from keras.layers import Dense
3 from keras.models import Sequential
4 from keras.optimizers import adam
5
6 # alpha = 0.001 as given in the lr parameter in adam() optimizer
7
8 # build the model
9 model_alpha1 = Sequential()
```

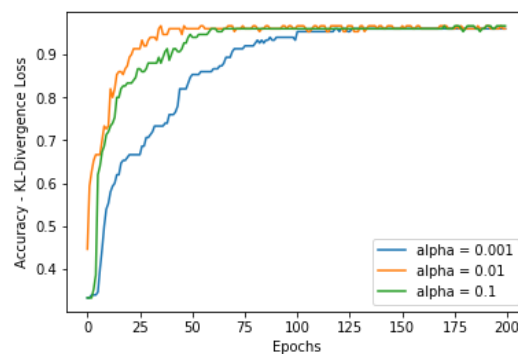
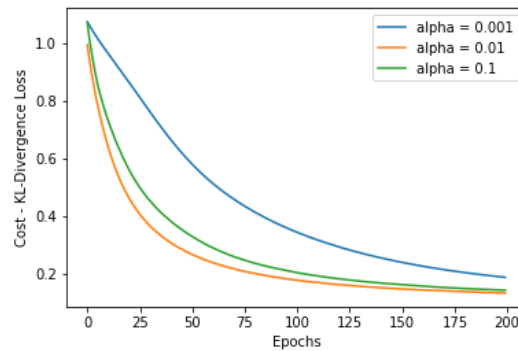
```

10 model_alpha1.add(Dense(50, input_dim=2, activation='relu'))
11 model_alpha1.add(Dense(3, activation='softmax'))
12
13 # compile the model
14 opt_alpha1 = adam(lr=0.001)
15 model_alpha1.compile(loss='kullback_leibler_divergence', optimizer=opt_alpha1, metrics=['accuracy'])
16
17 # fit the model
18 # dummy_Y is the one-hot encoded
19 # history_alpha1 is used to score the validation and accuracy scores for plotting
20 history_alpha1 = model_alpha1.fit(dataX, dummy_Y, validation_data=(dataX, dummy_Y), epochs=200, verbose=0)

```

[view raw](#)

model\_KL.py hosted with ♥ by GitHub



***KL-Divergence is used more commonly to approximate complex functions than in multi-class classification. We come across KL-Divergence frequently while playing with deep-generative models like Variational Autoencoders (VAEs).***

## End Notes

Woah! We have covered a lot of ground here. Give yourself a pat on your back for making it all the way to the end. This was quite a comprehensive list of loss functions we typically use in machine learning.

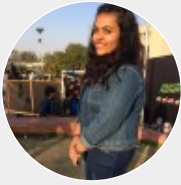
I would suggest going through this article a couple of times more as you proceed with your machine learning journey. This isn't a one-time effort. It will take a few readings and experience to understand how and where these loss functions work.

Make sure to experiment with these loss functions and let me know your observations down in the comments. Also, let me know other topics that you would like to read about. I will do my best to cover them in future articles.

Meanwhile, make sure you check out our comprehensive beginner-level machine learning course:

- [Applied Machine Learning – Beginner to Professional](#)
- 

Article Url - <https://www.analyticsvidhya.com/blog/2019/08/detailed-guide-7-loss-functions-machine-learning-python-code/>



## **Khyati Mahendru**

As a student of B.Tech in Mathematics and Computing, I look at everything through a lens of numbers. A story-teller by nature and a problem-solver at the core, I am gaining practical experience in ML and DS as an intern at Analytics Vidhya.