- Know the difference between loss function and cost function.
- Learn how to implement different loss functions in Python.

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?

Loss functions, also known as objective functions, are pivotal in machine learning algorithms. They guide decision-making by mapping choices to associated costs. Imagine being atop a hill and needing to descend. Rejecting uphill paths, you'd opt for the steepest downhill slope. Similarly, a loss function aids in minimizing errors during training by quantifying the discrepancy between predicted and actual values. This optimization process aims to achieve an optimum, where the loss function yields the lowest possible value. Common loss functions include quadratic loss (e.g., Mean Squared Error) and absolute value (e.g., Mean Absolute Error). Understanding and leveraging loss functions enhance comprehension of machine learning algorithms.

Also Read: Dimensionality Reduction Techniques | Python

Difference Between Loss Function and Cost Function

I want to emphasize this: 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.

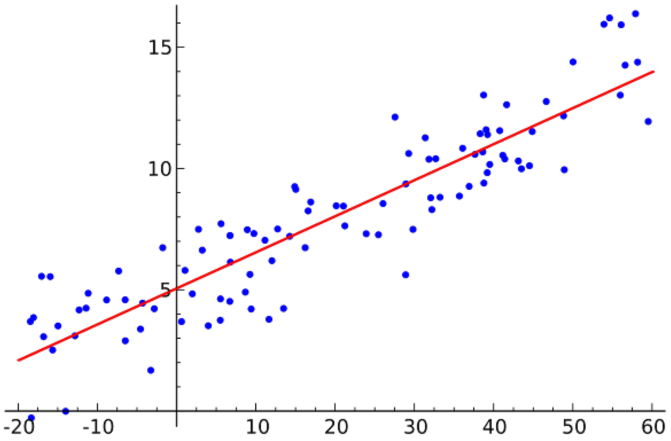
Also Read: Regularization in Machine Learning

What Are 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_i's. Thus, we essentially fit a line in space on these variables.

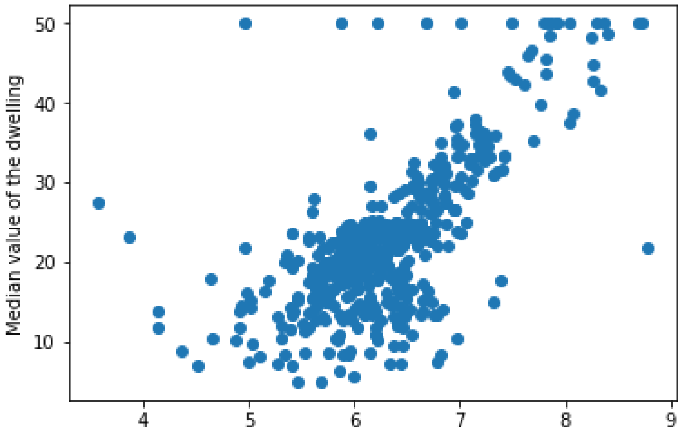
```
Y = a0 + a1 * X1 + a2 * X2 + ....+ an * Xn
```

We will use the given data points to find the coefficients a0, a1, ..., an.



We will use the famous Boston Housing Dataset to understand 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 – the *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:



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:

$$\text{Repeat until convergence } \{$$

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

$$\}$$

Here, *θj* represents the weight to be updated, *α* denotes the learning rate, and *J* symbolizes the cost function. The cost function, parameterized by *θ*, aims to minimize the overall cost. For an in-depth explanation of Gradient Descent and its workings, refer here.

Steps for Loss Functions

1. Define the predictor function *f*(*X*), and identify the parameters to find.
2. Determine the loss for each training example.
3. Derive the expression for the Cost Function, representing the average loss across all examples.
4. Compute the gradient of the Cost Function concerning each unknown parameter.
5. Select the learning rate and execute the weight update rule for a fixed number of iterations.

These steps guide the optimization process, aiding in the determination of optimal model parameters. Below, you'll find descriptions of various regression loss functions, including least squares, which quantify the deviation between predicted and actual values.

Also Read: [15 Most Important Features of Scikit-Learn!]

Mean Squared Error Loss

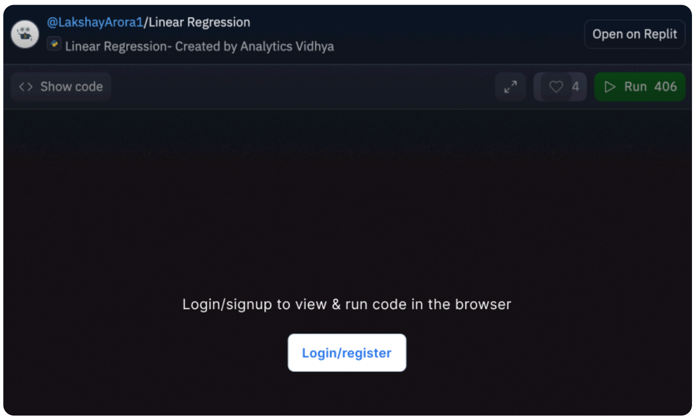
Squared Error loss for each training example, also known as **L2 Loss**, is the square of the difference between the actual and the predicted values:

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

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

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

Python Code:

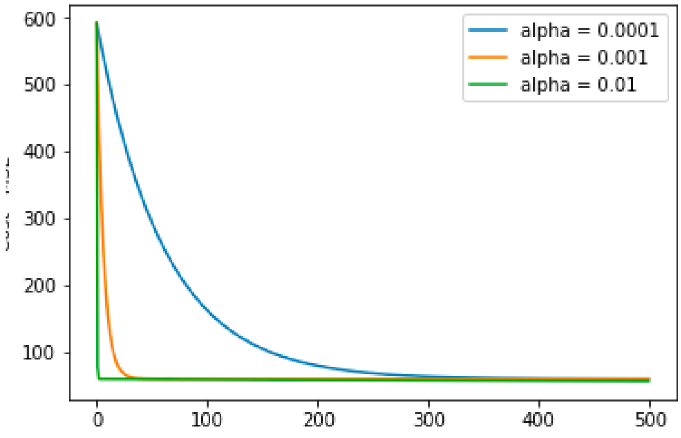


@LakshayArora1/Linear Regression

Linear Regression- Created by Analytics Vidhya

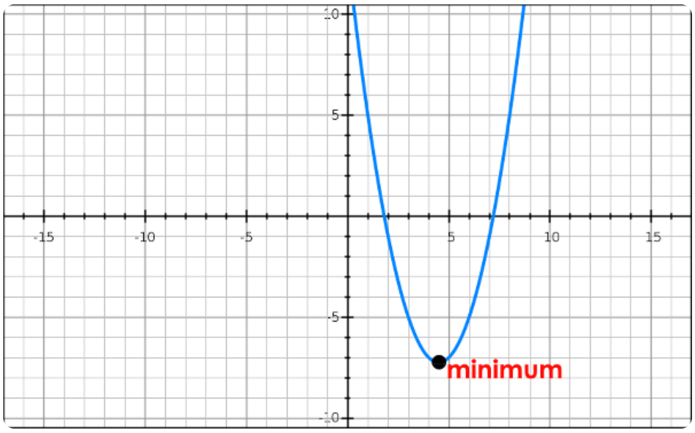
Login/signup to view & run code in the browser

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.***

Also Read: [Data Science Subjects and Syllabus [Latest Topics Included]]

Mean Absolute Error Loss

Absolute Error for each training example is the distance between the predicted and the actual values, irrespective of the sign, i.e., it is the absolute difference between the actual and predicted values. 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,** which is the Mean Absolute Error (MAE).

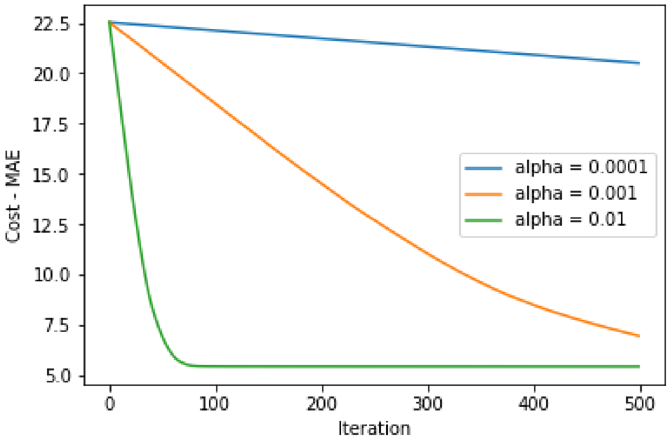
The MAE cost is more robust to outliers as compared to MSE. However, handling the absolute or modulus operator in mathematical equations is not easy. I'm sure a lot of you must agree with this! We can consider this as a disadvantage of MAE.

Here is the code for the *update_weight* function with MAE cost:

```python
def update_weights_MAE(m, b, X, Y, learning_rate):
    m_deriv = 0
    b_deriv = 0
    N = len(X)
    for i in range(N):
        # Calculate partial derivatives
        # -x(y - (mx + b)) / |mx + b|
        m_deriv += - X[i] * (Y[i] - (m*X[i] + b)) / abs(Y[i] - (m*X[

        # -(y - (mx + b)) / |mx + b|
        b_deriv += -(Y[i] - (m*X[i] + b)) / abs(Y[i] - (m*X[i] + b))

    # We subtract because the derivatives point in direction of stee
    m -= (m_deriv / float(N)) * learning_rate
    b -= (b_deriv / float(N)) * learning_rate

    return m, b
```

weight_update_MAE.py hosted with ❤ by GitHub view raw

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



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:

Quadratic

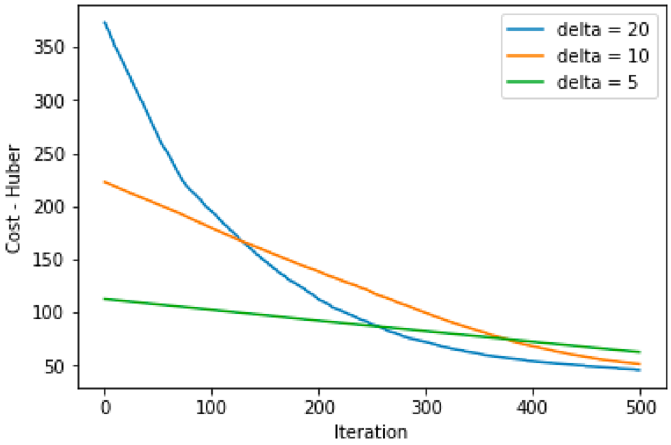
$$L_{\delta} = \begin{cases} \frac{1}{2}(y - f(x))^{2}, & if\ |y - f(x)| \le \delta \\ \delta|y - f(x)| - \frac{1}{2}\delta^{2}, & otherwise \end{cases}$$

Linear

```python
def update_weights_Huber(m, b, X, Y, delta, learning_rate):
    m_deriv = 0
    b_deriv = 0
    N = len(X)
    for i in range(N):
        # derivative of quadratic for small values and of linear for
        if abs(Y[i] - m*X[i] - b) <= delta:
          m_deriv += -X[i] * (Y[i] - (m*X[i] + b))
          b_deriv += - (Y[i] - (m*X[i] + b))
        else:
          m_deriv += delta * X[i] * ((m*X[i] + b) - Y[i]) / abs((m*X
          b_deriv += delta * ((m*X[i] + b) - Y[i]) / abs((m*X[i] + b

    # We subtract because the derivatives point in direction of stee
    m -= (m_deriv / float(N)) * learning_rate
    b -= (b_deriv / float(N)) * learning_rate

    return m, b
```

weight_update_Huber.py hosted with ❤ by GitHub view raw

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

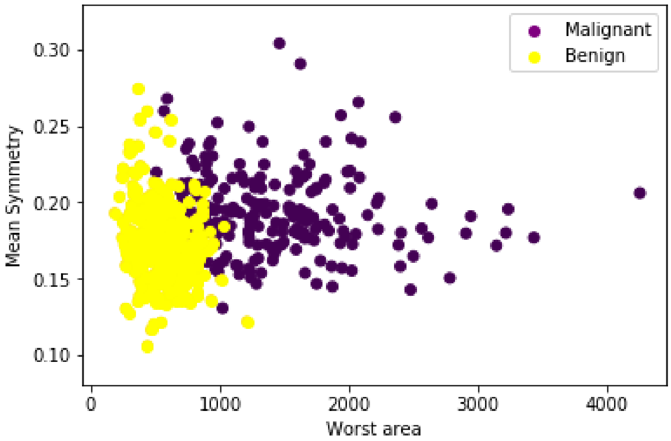
*We want to classify a tumor as **'Malignant'** or **'Benign'** based on features like average radius, area, perimeter, etc. For simplification, we will use only two input features*

*(X_1 and X_2), namely **‘worst area’** and **‘mean symmetry,’** for classification. The target value Y can be 0 (Malignant) or 1 (Benign).*

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

*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:



Here are the different types of binary classification loss functions.

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):

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

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 function for classification models, which output a probability *p*.

```
Probability that the element belongs to class 1 (or positive cl
Then, the probability that the element belongs to class 0 (or r
```

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

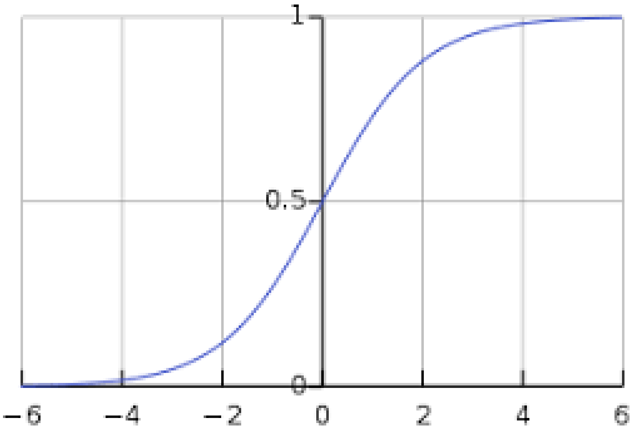
$$L = -y * \log(p) - (1-y) * \log(1-p) = \begin{cases} -\log(1-p), & if\ y = 0 \\ -\log(p), & if\ y = 1 \end{cases}$$

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.

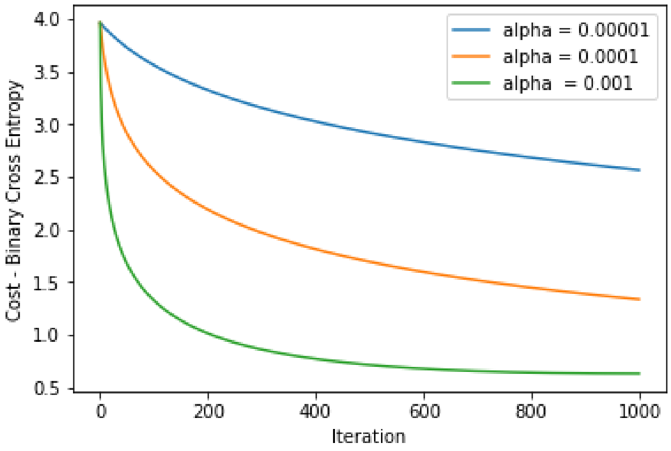


Try to find the gradient yourself and then look at the code for the *update_weight* function below:

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

        # Calculate partial derivatives
        m1_deriv += -X1[i] * (s - Y[i])
        m2_deriv += -X2[i] * (s - Y[i])
        b_deriv += -(s - Y[i])

    # We subtract because the derivatives point in direction of stee
    m1 -= (m1_deriv / float(N)) * learning_rate
    m2 -= (m2_deriv / float(N)) * learning_rate
    b -= (b_deriv / float(N)) * learning_rate

    return m1, m2, b
```

weight_update_BCE.py hosted with ❤ by GitHub view raw

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



Also Read: Everything you need to Know about Linear Regression

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.

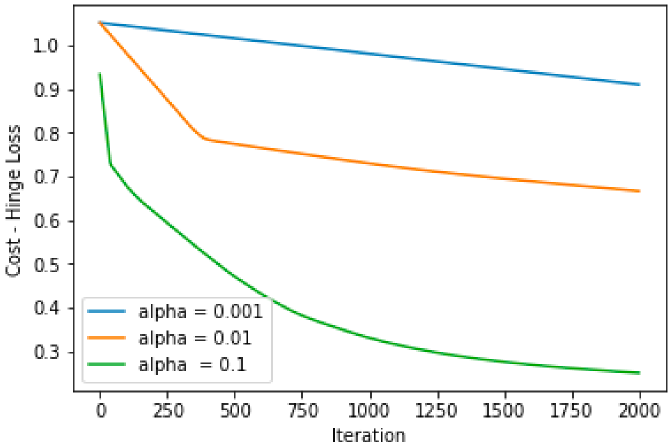
Hinge loss for an input-output pair (x, y) is given as:

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

```
def update_weights_Hinge(m1, m2, b, X1, X2, Y, learning_rate):
    m1_deriv = 0
    m2_deriv = 0
    b_deriv = 0
    N = len(X1)
    for i in range(N):
        # Calculate partial derivatives
        if Y[i]*(m1*X1[i] + m2*X2[i] + b) <= 1:
          m1_deriv += -X1[i] * Y[i]
          m2_deriv += -X2[i] * Y[i]
          b_deriv += -Y[i]
        # else derivatives are zero

    # We subtract because the derivatives point in direction of stee
    m1 -= (m1_deriv / float(N)) * learning_rate
    m2 -= (m2_deriv / float(N)) * learning_rate
    b -= (b_deriv / float(N)) * learning_rate

    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

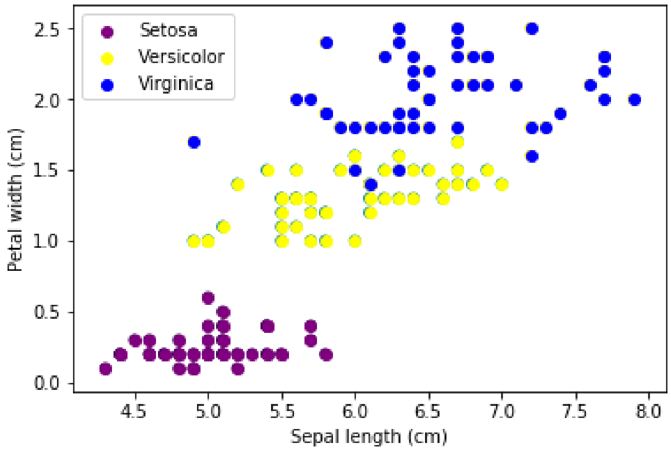
Emails are not just classified as spam or not spam (this isn't the 90s anymore!). They are classified into various other categories – Work, Home, Social, Promotions, etc. This is a Multi-Class Classification use case.

We'll use the Iris Dataset to understand the remaining two loss functions. We will use 2 features X_1, **Sepal length,** and feature X_2, **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:

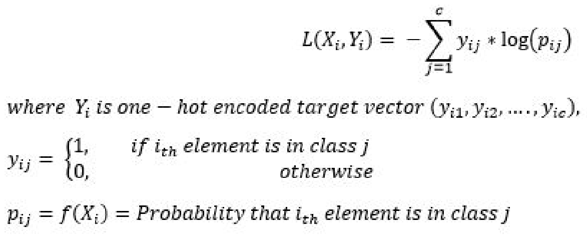


Here are the different types of multi-class classification loss functions.

Also Read: A Comprehensive Guide on Hyperparameter Tuning and its Techniques

Multi-Class Cross Entropy Loss

The multi-class cross-entropy loss function 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:



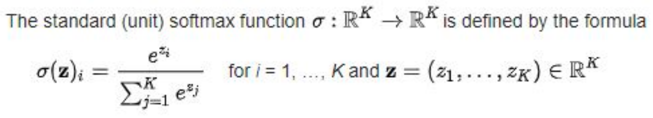
$$L(X_i, Y_i) = -\sum_{j=1}^{c} y_{ij} * \log(p_{ij})$$

$$where\ \ Y_i\ is\ one - hot\ encoded\ target\ vector\ (y_{i1}, y_{i2}, \ldots., y_{ic}),$$

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

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

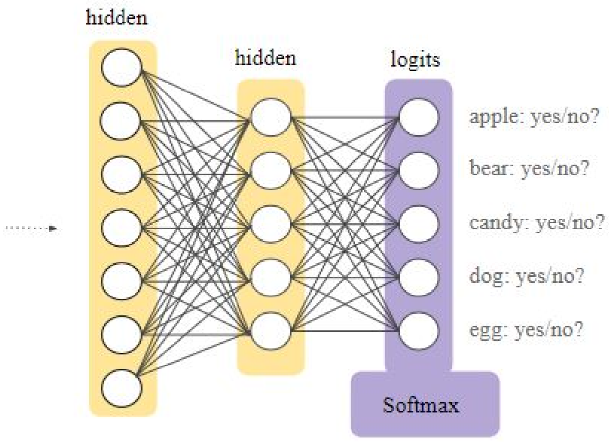
We use the softmax function to find the probabilities p_ij:



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

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

“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



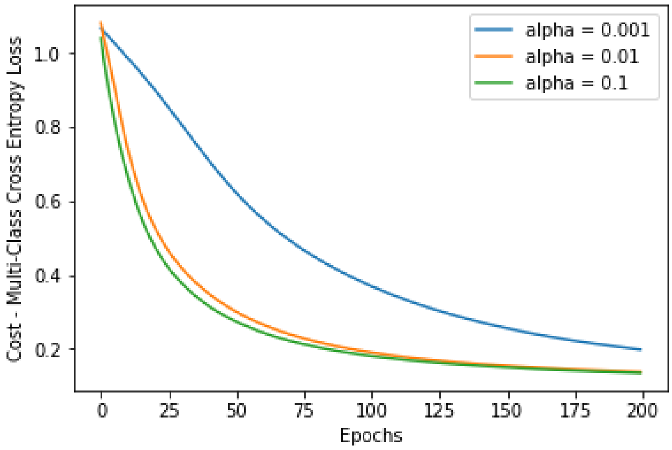
Finally, our output is the class with the maximum probability for the given input.

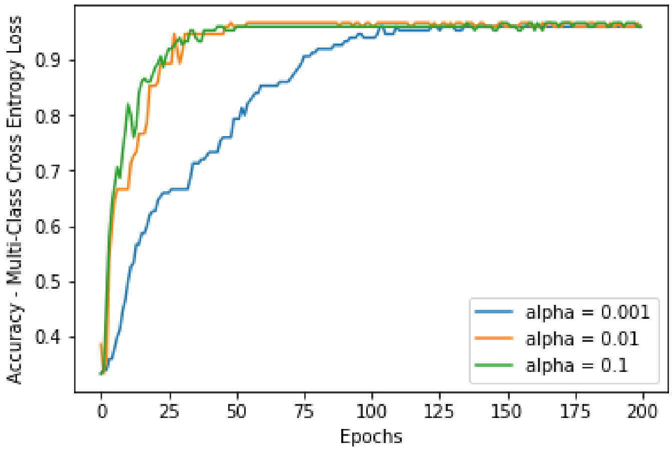
We build a model using an input layer and an output layer and compile it with different learning rates. Specify the loss parameter as **‘categorical_crossentropy’** in the model.compile() statement:

```python
# importing requirements
from keras.layers import Dense
from keras.models import Sequential
from keras.optimizers import adam

# alpha = 0.001 as given in the lr parameter in adam() optimizer

# build the model
model_alpha1 = Sequential()
model_alpha1.add(Dense(50, input_dim=2, activation='relu'))
model_alpha1.add(Dense(3, activation='softmax'))

# compile the model
opt_alpha1 = adam(lr=0.001)
model_alpha1.compile(loss='categorical_crossentropy', optimizer=opt_

# fit the model
# dummy_Y is the one-hot encoded
# history_alpha1 is used to score the validation and accuracy scores
history_alpha1 = model_alpha1.fit(dataX, dummy_Y, validation_data=(d
```

model_MCE.py hosted with ❤ by **GitHub** view raw

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





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

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

$= \text{Expectation of logaritmic difference between } P \text{ and } Q \text{ with respect to } P$

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, and 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:

$$D_{KL}(P||Q) = -\sum_{x}\big(P(x).logQ(x) - P(x).logP(x)\big) = H(P,Q) - H(P,P)$$

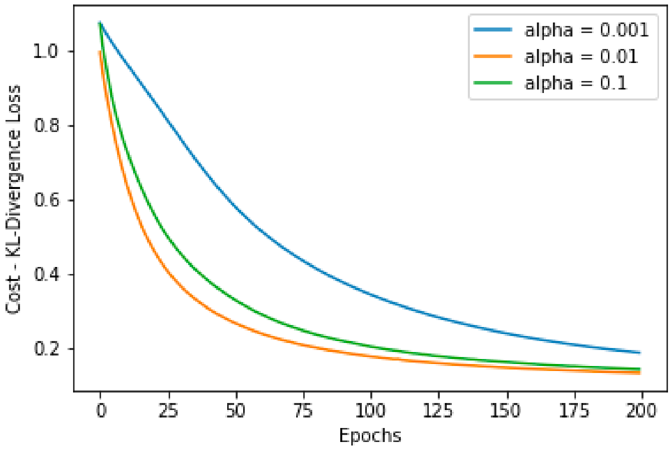
$H(P,P)\ is\ the\ entropy\ of\ P$

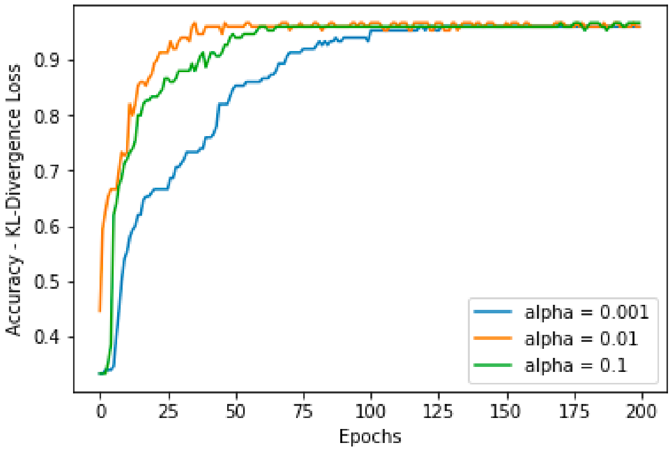
$H(P,Q)\ is\ the\ cross-entropy\ of\ P\ and\ Q$

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.

```
# importing requirements
from keras.layers import Dense
from keras.models import Sequential
from keras.optimizers import adam

# alpha = 0.001 as given in the lr parameter in adam() optimizer

# build the model
model_alpha1 = Sequential()
model_alpha1.add(Dense(50, input_dim=2, activation='relu'))
model_alpha1.add(Dense(3, activation='softmax'))

# compile the model
opt_alpha1 = adam(lr=0.001)
model_alpha1.compile(loss='kullback_leibler_divergence', optimizer=o

# fit the model
# dummy_Y is the one-hot encoded
# history_alpha1 is used to score the validation and accuracy scores
history_alpha1 = model_alpha1.fit(dataX, dummy_Y, validation_data=(d
```

model_KL.py hosted with ❤ by GitHub   view raw





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).

Conclusion

Woah! Through this tutorial, 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 more times 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.

Meanwhile, you can also check out our comprehensive beginner-level machine learning course: Applied Machine Learning – Beginner to Professional