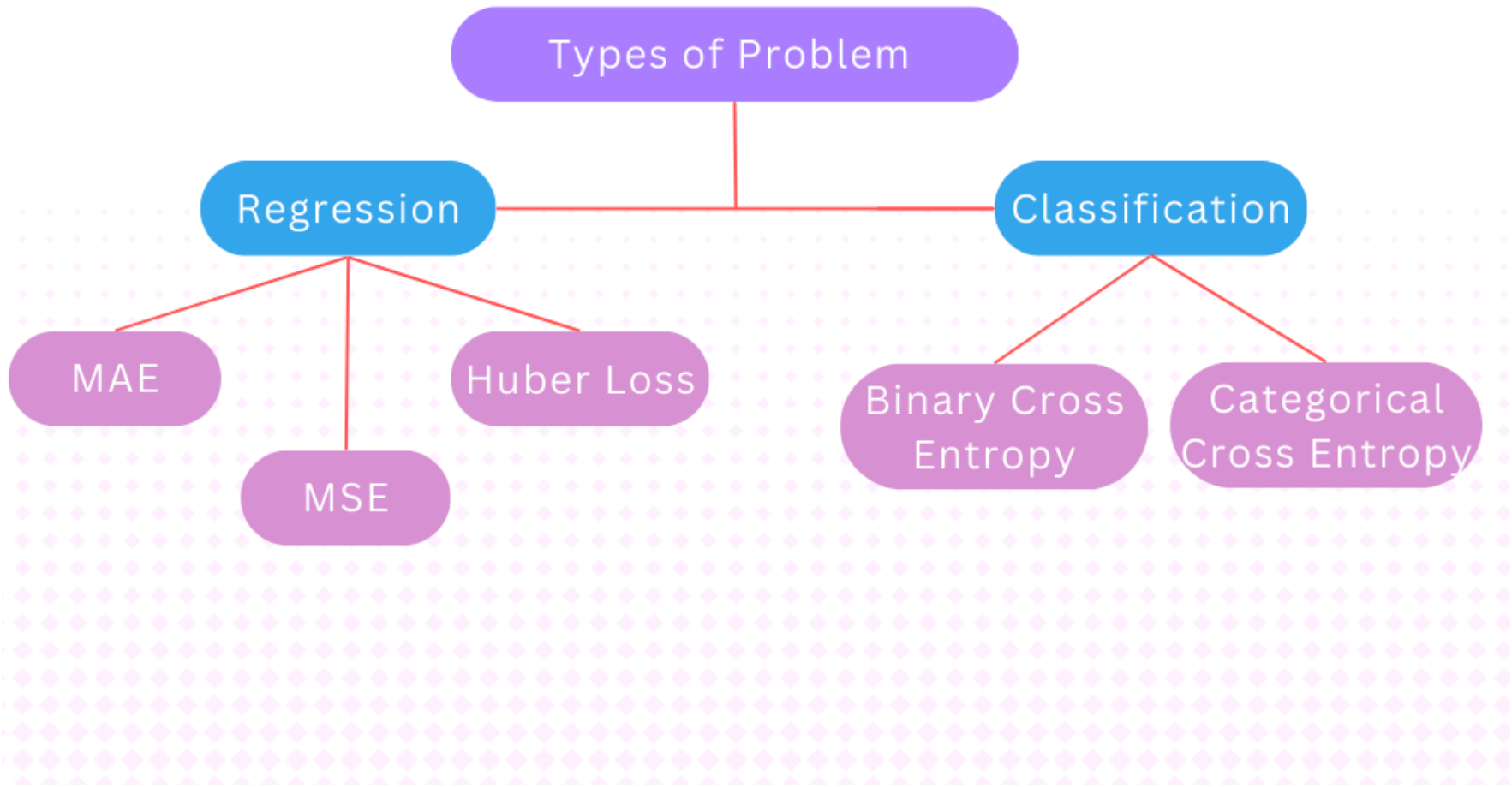


# Loss Function

A loss function (or cost function) is a measure of how well a model's predictions match the actual desired outcomes. It quantifies the difference between predicted values and the actual target values, providing a concrete number that represents the "cost" associated with the errors made by the model.

## Loss Function in ML



### Loss Function for Regression Machine Learning Problem

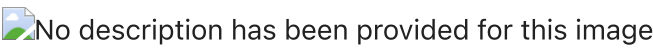
```
In [32]: import numpy as np

In [33]: y_pred = np.array([1,1,1,0,0,1])
         y_true = np.array([0.30,0.70,1,0,0.5,0.6])
```

#### 1. Mean Absolute Error (MAE)

**Mean Absolute Error (MAE)** is a common metric used to measure the average magnitude of errors in a set of predictions, without considering their direction. In other words, it measures the average absolute difference between predicted values and actual values.

##### Formula



##### Use Case

MAE is particularly useful in regression problems where you need to evaluate the performance of predictive models. It's often used in scenarios where outliers are not predominant because MAE treats all errors equally, without giving extra weight to larger errors.

##### Advantages

- 1. **Simplicity:** Easy to understand and interpret.
- 2. **Non-Sensitivity to Outliers:** Each error contributes proportionally to the total error, making it less sensitive to outliers compared to metrics like Mean Squared Error (MSE).
- 3. **Direct Interpretability:** The error is in the same unit as the target variable, making it easy to understand the magnitude of the error.

##### Disadvantages

1. **Equal Weight to All Errors:** Does not account for the severity of larger errors, which might be important in some contexts.
2. **Gradient Smoothness:** The MAE function is not differentiable at zero, which can be problematic for certain optimization algorithms during model training.
3. **Less Emphasis on Large Errors:** Might not be suitable for applications where large errors are particularly undesirable and need to be penalized more heavily.

## Example

Suppose you have a machine learning model that predicts house prices. You compare the predicted prices with the actual prices for four houses:

`y_predicted = [250000, 150000, 200000, 300000]`

`y_true = [240000, 160000, 210000, 290000]`

To calculate the MAE:

1. Calculate the absolute errors for each prediction:

- $|250000 - 240000| = 10000$
- $|150000 - 160000| = 10000$
- $|200000 - 210000| = 10000$
- $|300000 - 290000| = 10000$

2. Sum these absolute errors:

- $10000 + 10000 + 10000 + 10000 = 40000$

3. Divide by the number of observations (4):

- $MAE = 40000 / 4 = 10000$

So, the MAE is 10000.

1. **MAE is a useful metric for many regression problems due to its simplicity and interpretability.**
2. **However, it may not always be the best choice when outliers are a significant concern or when larger errors need to be penalized more.**
3. **In such cases, other metrics like Mean Squared Error (MSE) or Huber loss might be more appropriate.**

## Note

1. The closer MAE is to 0, the more accurate the model is.
2. Also known as L1 Loss Function

## Implement MAE Error

```
In [3]: def mae(y_pred, y_true):
        total_error = 0
        for y_p, y_t in zip(y_pred, y_true):
            total_error += abs(y_p - y_t)
        print('The total error is : ', total_error)
        mae = total_error / len(y_pred)
        print('MSE is : ', mae)
        return mae
```

```
In [4]: mae(y_pred, y_true)
```

The total error is : 1.9  
MSE is : 0.31666666666666665

```
Out[4]: 0.31666666666666665
```

---

## Lets Check for above mentioned example

```
In [5]: y_predicted_house = [250000, 150000, 200000, 300000]
        y_true_house = [240000, 160000, 210000, 290000]
```

```
In [6]: mae(y_predicted_house, y_true_house)
```

The total error is : 40000  
MSE is : 10000.0

```
Out[6]: 10000.0
```

---

## Implementing same thing in np

```
In [7]: np.abs(y_pred - y_true)

Out[7]: array([0.7, 0.3, 0. , 0. , 0.5, 0.4])

np.mean(np.abs(y_pred - y_true))

In [8]: def np_mae(y_pred, y_true):
        return np.mean(np.abs(y_pred - y_true))

In [9]: np_mae(y_pred,y_true)

Out[9]: 0.31666666666666665
```

## 2. Mean Squared Error (MSE)

**Mean Squared Error (MSE)** is a commonly used metric to measure the average of the squares of the errors. It calculates the square of the difference between the predicted values and the actual values, and then averages those squared differences. MSE is widely used in regression analysis to assess the accuracy of a model.

### Formula

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

### Use Case

MSE is frequently used in regression problems, especially when large errors are undesirable and need to be penalized more heavily. It is particularly useful in models like linear regression, polynomial regression, and other machine learning algorithms where minimizing error is crucial.

### Advantages

- 1. **Penalizes Large Errors:** MSE gives more weight to larger errors since it squares the error terms. This can be useful if large errors are particularly undesirable.
- 2. **Smooth Gradient:** The squared nature of MSE provides a smooth gradient, which is beneficial for optimization algorithms like gradient descent.
- 3. **Widely Used:** MSE is a standard metric in regression analysis, making it easier to compare results across different studies and models.

### Disadvantages

- 1. **Sensitive to Outliers:** Due to squaring the error terms, MSE is highly sensitive to outliers. Large errors have a disproportionately large impact on the MSE.
- 2. **Not Interpretable:** The units of MSE are the square of the original units of the target variable, making it less interpretable compared to metrics like MAE.
- 3. **May Overemphasize Large Errors:** In some contexts, the heavy penalization of large errors may not be desirable and can lead to overfitting.

### Example

Consider a scenario where a machine learning model predicts house prices. You compare the predicted prices with the actual prices for four houses:

```
y_predicted = [250000, 150000, 200000, 300000]
y_true = [240000, 160000, 210000, 290000]
To calculate the MSE:
```

- 1. Calculate the squared errors for each prediction:
  - $(250000 - 240000)^2 = 100000000$
  - $(150000 - 160000)^2 = 100000000$
  - $(200000 - 210000)^2 = 100000000$
  - $(300000 - 290000)^2 = 100000000$
- 2. Sum these squared errors:

- $100000000 + 100000000 + 100000000 + 100000000 = 400000000$
3. Divide by the number of observations (4):

- $MSE = 400000000 / 4 = 100000000$

So, the MSE is 100000000.

1. **MSE is a valuable metric for evaluating regression models due to its smooth gradient and the penalization of large errors.**
2. **However, its sensitivity to outliers and lack of direct interpretability are important considerations when choosing an evaluation metric.**
3. **For contexts where large errors are particularly problematic or optimization requires smooth gradients, MSE is often the preferred choice.**

## Note -

1. Lower the value the better and 0 means the model is perfect.
2. Also Known as L2 Loss Function.

## Implement MSE Error

```
In [10]: def mse(y_predicted, y_true):
          total_error = 0
          for yp, yt in zip(y_predicted, y_true):
              total_error += (yp - yt) ** 2
          print("Total squared error is:", total_error)
          mse = total_error / len(y_predicted)
          print("Mean squared error is:", mse)
          return mse
```

```
In [11]: mse(y_pred, y_true)
```

Total squared error is: 0.99  
Mean squared error is: 0.165

```
Out[11]: 0.165
```

---

## Lets Check for above mentioned example

```
In [12]: y_predicted_house = [250000, 150000, 200000, 300000]
          y_true_house = [240000, 160000, 210000, 290000]
```

```
In [13]: mse(y_predicted_house, y_true_house)
```

Total squared error is: 400000000  
Mean squared error is: 100000000.0

```
Out[13]: 100000000.0
```

---

## Implementing same using np

```
In [14]: square_diff = (y_pred - y_true) ** 2
```

```
In [15]: def np_mse(y_pred, y_true):
          mse_value = np.mean(square_diff)
          return mse_value
```

```
In [16]: np_mse(y_pred, y_true)
```

```
Out[16]: 0.165
```

---

## 3. Huber Loss

**Huber Loss** is a loss function used in robust regression, which is less sensitive to outliers than the Mean Squared Error (MSE) but behaves similarly to Mean Absolute Error (MAE) when the error is large. It is quadratic for small errors and linear for large errors, providing a balance between MSE and MAE.

## Formula

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

## Use Case

Huber Loss is particularly useful in regression problems where you want to be less sensitive to outliers than MSE but still penalize large errors more than MAE. It is often used in applications like robust regression, where data may contain outliers, and you want a loss function that is robust to these outliers.

## Advantages

1. **Robust to Outliers:** Unlike MSE, Huber Loss is less sensitive to outliers because it transitions from a quadratic to a linear loss.
2. **Combines Best of Both Worlds:** Provides the benefits of both MSE (smooth gradients for small errors) and MAE (robustness to outliers).
3. **Differentiable Everywhere:** Smooth gradient transitions help in gradient-based optimization.

## Disadvantages

1. **Requires Tuning:** The parameter ( `\delta` ) needs to be tuned, which can be an additional hyperparameter to consider during model training.
2. **Complexity:** More complex to implement and understand compared to MSE and MAE.
3. **Less Common:** Not as widely used or standardized as MSE or MAE, making comparisons across models or studies potentially more challenging.


## Example

Suppose you have a machine learning model that predicts house prices. You compare the predicted prices with the actual prices for four houses:

```
y_predicted = [250000, 150000, 200000, 300000]
y_true = [240000, 160000, 210000, 290000]
delta = 10000
```

To calculate the Huber Loss:

1. Calculate the errors for each prediction:
  - 250000 - 240000 = 10000
  - 150000 - 160000 = -10000
  - 200000 - 210000 = -10000
  - 300000 - 290000 = 10000
2. Apply the Huber loss formula to each error:

No description has been provided for this image

3. Sum these losses: - 50000000 + 50000000 + 50000000 + 50000000 = 200000000

4. Divide by the number of observations (4):
  - Huber Loss = 200000000 / 4 = 50000000

So, the Huber Loss is 50000000.

1. **Huber Loss is a versatile loss function that offers robustness to outliers while maintaining smooth gradients for optimization.**
2. **It is particularly useful in regression problems with outlier data, although it requires tuning the ( `\delta` ) parameter and is more complex than simpler loss functions like MSE and MAE.**

## Note -

1. Huber Loss Smooth Mean Absolute Error

## Implement Huber Loss

```
In [17]: def huber_loss(y_pred, y_true, delta = 1.0):
          error = y_pred - y_true

          condition = np.abs(error) <= delta
          squared_loss = 0.5 * error**2
          linear_loss = delta * (np.abs(error) - 0.5 * delta)
```

```
huber_loss = np.where(condition, squared_loss, linear_loss)
mean_huber_loss = np.mean(huber_loss)
return mean_huber_loss
```

```
In [18]: huber_loss_value = huber_loss(y_pred, y_true)
print("Huber Loss:", huber_loss_value)
```

Huber Loss: 0.0825

## Lets Check for above mentioned example

```
In [19]: def huber_loss(y_pred, y_true, delta = 10000):
    error = y_predicted_house - y_true_house

    condition = np.abs(error) <= delta
    squared_loss = 0.5 * error**2
    linear_loss = delta * (np.abs(error) - 0.5 * delta)

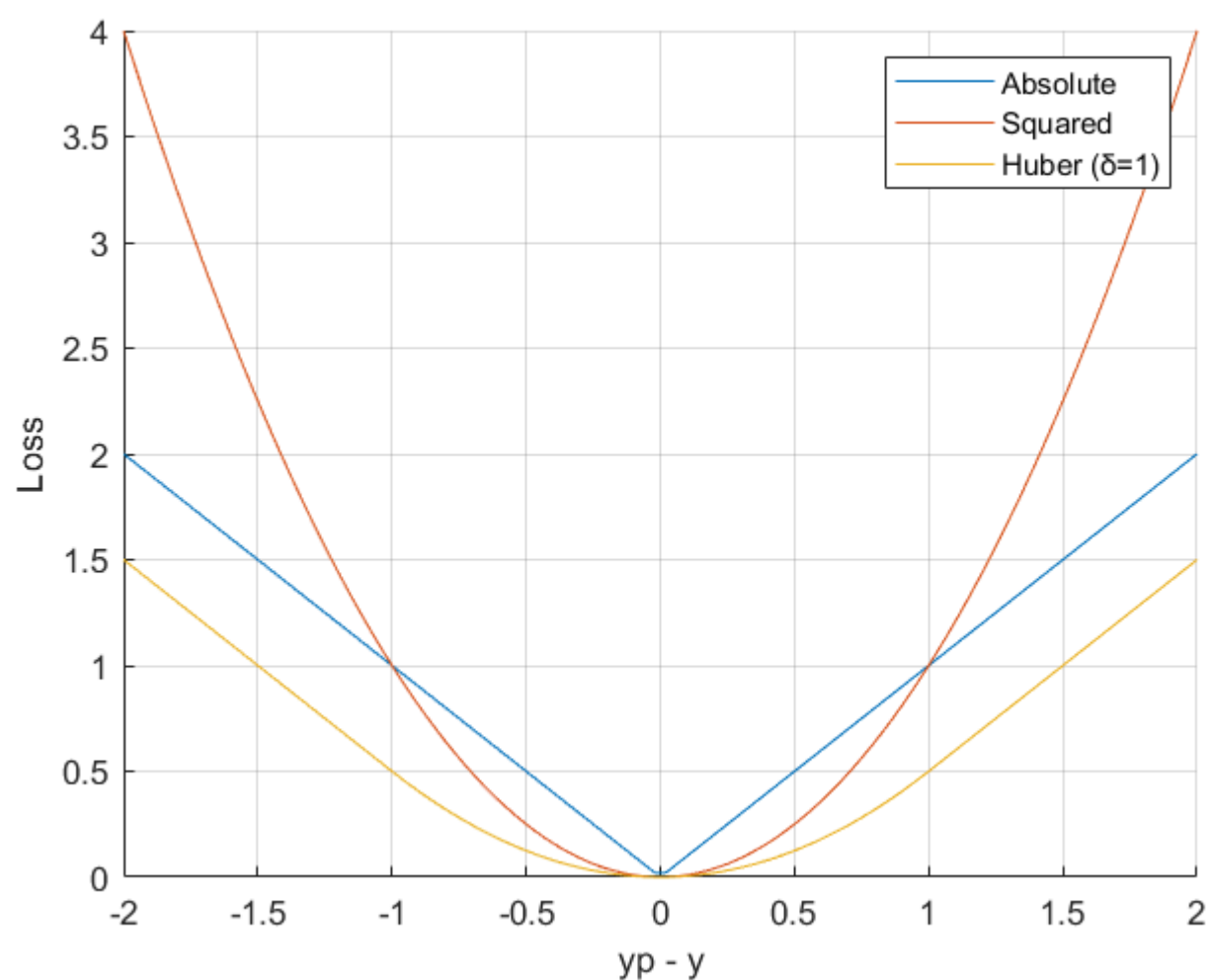
    huber_loss = np.where(condition, squared_loss, linear_loss)
    mean_huber_loss = np.mean(huber_loss)
    return mean_huber_loss
```

```
In [20]: y_predicted_house = np.array([250000, 150000, 200000, 300000])
y_true_house = np.array([240000, 160000, 210000, 290000])
```

```
In [21]: huber_loss_value = huber_loss(y_predicted_house, y_true_house)
print("Huber Loss:", huber_loss_value)
```

Huber Loss: 50000000.0

## Graphical Representation of Loss Functions



## Loss Function for Classification Machine Learning Problem

### 1. Binary Cross Entropy

**Binary Cross Entropy (BCE)**, also known as log loss, is a loss function commonly used for binary classification problems. It measures the performance of a classification model whose output is a probability value between 0 and 1. BCE quantifies the difference between the actual label (0 or 1) and the predicted probability.

Formula:



$$\text{Loss} = -\frac{1}{\text{output size}} \sum_{i=1}^{\text{output size}} y_i \cdot \log \hat{y}_i + (1 - y_i) \cdot \log (1 - \hat{y}_i)$$

## Use Case

BCE is widely used in binary classification tasks, such as:

- Spam detection (spam or not spam)
- Medical diagnosis (disease present or not)
- Fraud detection (fraudulent or non-fraudulent transactions)

## Advantages

1. **Probabilistic Interpretation:** BCE uses predicted probabilities, providing a clear interpretation of the model's confidence in its predictions.
2. **Differentiable:** The BCE function is smooth and differentiable, making it suitable for gradient-based optimization algorithms.
3. **Handles Imbalanced Classes:** With appropriate adjustments (like weighting), BCE can handle imbalanced datasets effectively.

## Disadvantages

1. **Sensitive to Predictions:** BCE can be highly sensitive to incorrect predictions with high confidence, leading to very large loss values.
2. **Requires Probabilistic Output:** Models must output probabilities, which may require additional transformations (e.g., using a sigmoid activation function).
3. **Logarithm Calculation:** Involves logarithm calculations, which can be computationally expensive and may lead to numerical instability for very small predicted probabilities.

## Example

Consider a binary classification problem where a model predicts whether an email is spam (1) or not spam (0). For a set of four emails, suppose the actual labels and predicted probabilities are:

```
y_true = [1, 0, 1, 0]
y_pred = [0.9, 0.1, 0.8, 0.4]
```

To calculate the BCE:

1. Calculate the individual terms for each observation:
  - For (  $y_1 = 1$  ): (  $-[1 \log(0.9) + (1 - 1) \log(1 - 0.9)] = -\log(0.9)$  )
  - For (  $y_2 = 0$  ): (  $-[0 \log(0.1) + (1 - 0) \log(1 - 0.1)] = -\log(0.9)$  )
  - For (  $y_3 = 1$  ): (  $-[1 \log(0.8) + (1 - 1) \log(1 - 0.8)] = -\log(0.8)$  )
  - For (  $y_4 = 0$  ): (  $-[0 \log(0.4) + (1 - 0) \log(1 - 0.4)] = -\log(0.6)$  )
2. Sum these values:
  - $\text{BCE} = ( -\frac{1}{4} (\log(0.9) + \log(0.9) + \log(0.8) + \log(0.6)) )$
3. Average the sum:
  - $\text{BCE} = ( -\frac{1}{4} (0.105 + 0.105 + 0.223 + 0.511) \approx 0.236 )$

So the binary cross entropy is : 0.23617255159896325

1. **Binary Cross Entropy is a fundamental loss function for binary classification problems, offering a probabilistic interpretation and smooth gradients for optimization.**
2. **However, it is sensitive to high-confidence errors and requires models to output probabilities, necessitating careful handling of numerical stability.**

## Note -

1. Also known as logloss
2. Cross-Entropy < 0.20 - Fine(Very Good)
3. Cross-Entropy > 0.30 - Not great(But Acceptable)
4. Cross-Entropy > 1.00 - Terrible(Very Bad)
5. Cross-Entropy > 2.00 - Extremely bad performing model

## Implement BCE Error

```
In [35]: def binary_cross_entropy(y_true, y_pred):
          # Clip predictions to avoid log(0) errors (i.e., clips the predicted probabilities to avoid taking the log of 0,
```

```
y_pred = np.clip(y_pred, 1e-10, 1 - 1e-10)

bce = -np.mean(y_true * np.log(y_pred) + (1 - y_true) * np.log(1 - y_pred))
return bce

# Example usage
bce_value = binary_cross_entropy(y_true, y_pred)
print("Binary Cross Entropy:", bce_value)
```

Binary Cross Entropy: 7.291519441910061

---

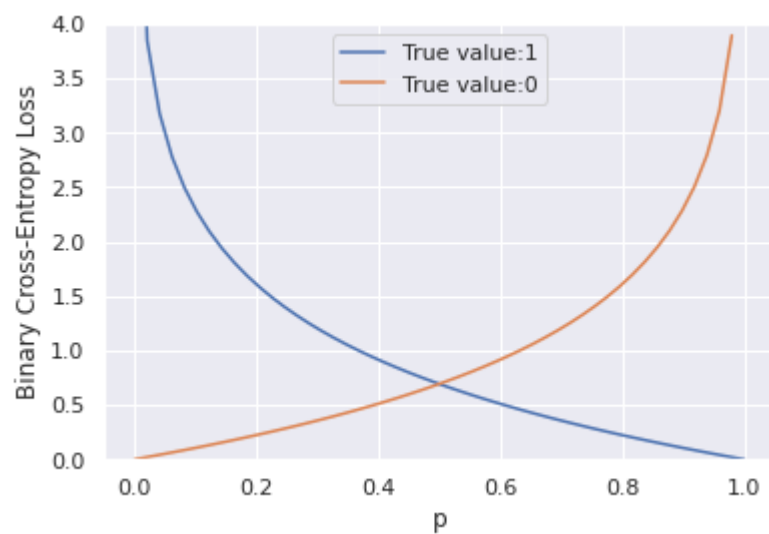
Lets chcek for above mentioned example

```
In [38]: y_true_example = np.array([1, 0, 1, 0])
y_pred_example = np.array([0.9, 0.1, 0.8, 0.4])
```

```
In [39]: bce_value = binary_cross_entropy(y_true_example, y_pred_example)
print("Binary Cross Entropy:", bce_value)
```

Binary Cross Entropy: 0.23617255159896325

## Graphical Representation of BCE



---

## 2. Categorical Cross Entropy

**Categorical Cross Entropy (CCE)**, also known as softmax loss, is a loss function commonly used for multi-class classification problems. It measures the performance of a classification model whose output is a probability distribution over multiple classes. CCE quantifies the difference between the true class labels and the predicted probabilities.

Formula:

$$\text{Loss} = - \sum_{i=1}^{\text{output size}} y_i \cdot \log \hat{y}_i$$

### Use Case

CCE is widely used in multi-class classification tasks, such as:

- Image classification (e.g., identifying objects in images)
- Text classification (e.g., sentiment analysis, topic categorization)
- Medical diagnosis (e.g., predicting disease categories)

### Advantages

1. **Probabilistic Interpretation:** CCE uses predicted probabilities, providing a clear interpretation of the model's confidence in its predictions.
2. **Differentiable:** The CCE function is smooth and differentiable, making it suitable for gradient-based optimization algorithms.
3. **Handles Multi-Class Problems:** Specifically designed for multi-class classification, making it ideal for tasks with multiple categories.

### Disadvantages



1. **Sensitive to Predictions:** CCE can be highly sensitive to incorrect predictions with high confidence, leading to very large loss values.
2. **Requires Probabilistic Output:** Models must output probabilities, which may require additional transformations (e.g., using a softmax activation function).
3. **Computationally Expensive:** Involves calculating logarithms, which can be computationally expensive and may lead to numerical instability for very small predicted probabilities.

## Example

Consider a scenario where a model predicts the type of fruit (apple, banana, or cherry). For a set of three fruits, the true labels and predicted probabilities might be:

```
y_true = [[1, 0, 0], [0, 1, 0], [0, 0, 1]]
y_pred = [[0.7, 0.2, 0.1], [0.1, 0.8, 0.1], [0.2, 0.2, 0.6]]
```

To calculate the CCE:

1. Calculate the cross-entropy for each observation:
  - For the first fruit:  $-(1 \log(0.7) + 0 \log(0.2) + 0 \log(0.1)) = -\log(0.7)$
  - For the second fruit:  $-(0 \log(0.1) + 1 \log(0.8) + 0 \log(0.1)) = -\log(0.8)$
  - For the third fruit:  $-(0 \log(0.2) + 0 \log(0.2) + 1 \log(0.6)) = -\log(0.6)$
2. Sum these values and divide by the number of observations (3):
  - $CCE = -\frac{1}{3} (\log(0.7) + \log(0.8) + \log(0.6))$

So the Categorical Cross Entropy value is : 0.3635480396729776

1. **Categorical Cross Entropy is a fundamental loss function for multi-class classification problems, offering a probabilistic interpretation and smooth gradients for optimization.**
2. **However, it is sensitive to high-confidence errors and requires models to output probabilities, necessitating careful handling of numerical stability.**

## Note -

1. Also know as softmax loss
2. Cross-Entropy = 0.00 - Perfect predictions.
3. Cross-Entropy < 0.02 - Great predictions.
4. Cross-Entropy < 0.05 - On the right track.
5. Cross-Entropy < 0.20 - Fine.
6. Cross-Entropy > 0.30 - Not great.
7. Cross-Entropy > 1.00 - Terrible.
8. Cross-Entropy > 2.00 - Extremely bad performing Model.

## Implement CCE Error

```
In [40]: def categorical_cross_entropy(y_true, y_pred):

    # Clip predictions to avoid log(0) errors
    y_pred = np.clip(y_pred, 1e-10, 1 - 1e-10)

    cce = -np.sum(y_true * np.log(y_pred)) / y_true.shape[0]

    return cce
cce_value = categorical_cross_entropy(y_true, y_pred)
print("Categorical Cross Entropy:", cce_value)
```

Categorical Cross Entropy: 1.9188209108717047

## Lets chcek for above mentioned example

```
In [43]: y_true = [[1, 0, 0], [0, 1, 0], [0, 0, 1]]
y_pred = [[0.7, 0.2, 0.1], [0.1, 0.8, 0.1], [0.2, 0.2, 0.6]]
```

```
In [44]: y_true_ex = np.array(y_true)
y_pred_ex = np.array(y_pred)
```

```
In [45]: cce_value = categorical_cross_entropy(y_true_ex, y_pred_ex)
print("Categorical Cross Entropy:", cce_value)
```

Categorical Cross Entropy: 0.3635480396729776

## Contact Information

*Please contact us for additional inquiries and collaboration opportunities.*

### Email

mdssohail1018@gmail.com

### Github

(tsohail12)

**Thank you for your time and consideration!!!**