

[Open in app](#)[Follow](#)

589K Followers



Understanding the 3 most common loss functions for Machine Learning Regression



George Seif · May 21, 2019 · 5 min read

☆ If you love to learn new things, check out my newsletter: **Mighty Knowledge**

A loss function in Machine Learning is a measure of how accurately your ML model is able to predict the expected outcome i.e the ground truth.

The loss function will take two items as input: the output value of our model and the ground truth expected value. The output of the loss function is called the *loss* which is a measure of how well our model did at predicting the outcome.

A high value for the loss means our model performed very poorly. A low value for the loss means our model performed very well.

Selection of the proper loss function is critical for training an accurate model. Certain loss functions will have certain properties and help your model learn in a specific way. Some may put more weight on outliers, others on the majority.

In this article we're going to take a look at the 3 most common loss functions for Machine Learning Regression. I'll explain how they work, their pros and cons, and how they can be most effectively applied when training regression models.

(1) Mean Squared Error (MSE)

The Mean Squared Error (MSE) is perhaps the simplest and most common loss function, often taught in introductory Machine Learning courses. To calculate the

[Open in app](#)

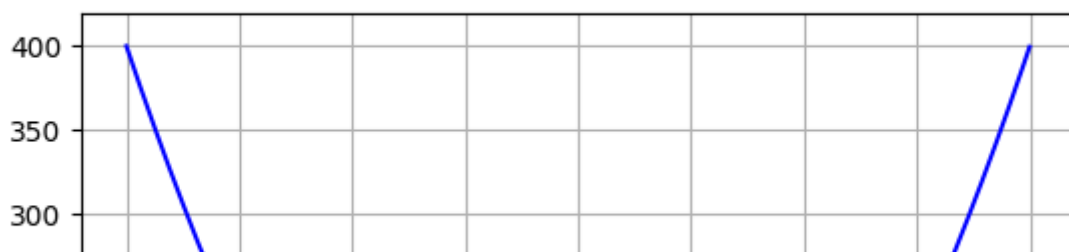
The MSE will never be negative, since we are always squaring the errors. The MSE is formally defined by the following equation:

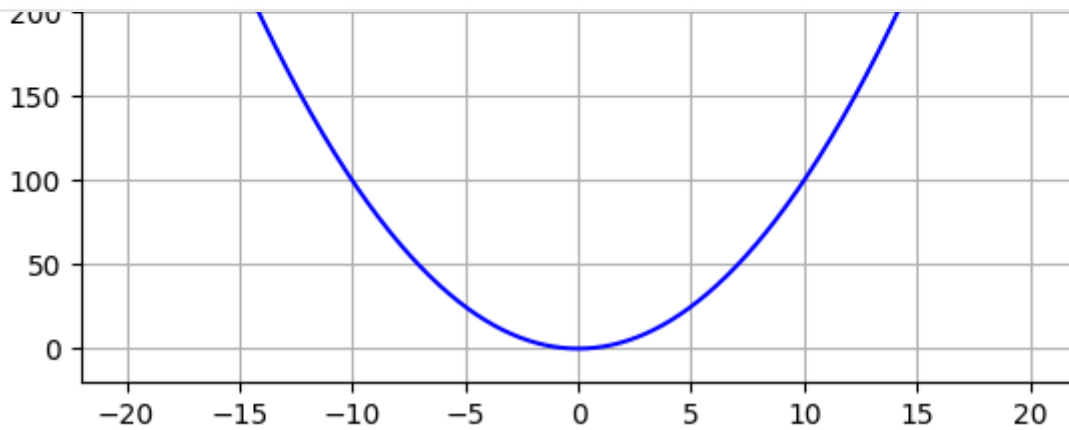
$$\text{MSE} = \frac{1}{N} \sum_{i=1}^N (y_i - \hat{y}_i)^2$$

Where N is the number of samples we are testing against. The code is simple enough, we can write it in plain numpy and plot it using matplotlib:

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  # MSE loss function
5  def mse_loss(y_pred, y_true):
6      squared_error = (y_pred - y_true) ** 2
7      sum_squared_error = np.sum(squared_error)
8      loss = sum_squared_error / y_true.size
9      return loss
10
11 # Plotting
12 x_vals = np.arange(-20, 20, 0.01)
13 y_vals = np.square(x_vals)
14
15 plt.plot(x_vals, y_vals, "blue")
16 plt.grid(True, which="major")
17 plt.show()
```

mse.py hosted with ❤ by GitHub

[view raw](#)

[Open in app](#)

MSE Loss Function

Advantage: The MSE is great for ensuring that our trained model has no outlier predictions with huge errors, since the MSE puts larger weight on these errors due to the squaring part of the function.

Disadvantage: If our model makes a single very bad prediction, the squaring part of the function magnifies the error. Yet in many practical cases we don't care much about these outliers and are aiming for more of a well-rounded model that performs good enough on the majority.

(2) Mean Absolute Error (MAE)

The Mean Absolute Error (MAE) is only slightly different in definition from the MSE, but interestingly provides almost exactly opposite properties! To calculate the MAE, you take the difference between your model's predictions and the ground truth, apply the absolute value to that difference, and then average it out across the whole dataset.

The MAE, like the MSE, will never be negative since in this case we are always taking the absolute value of the errors. The MAE is formally defined by the following equation:

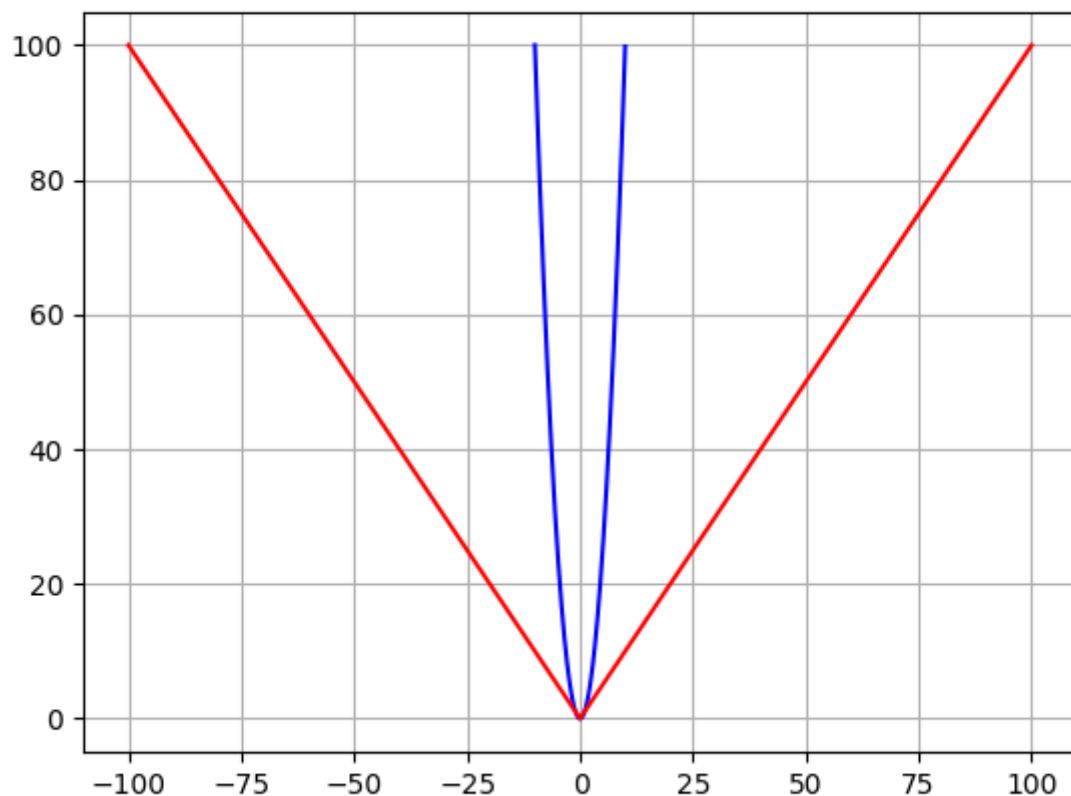
$$\text{MAE} = \frac{1}{n} \sum_{j=1}^n |y_j - \hat{y}_j|$$

[Open in app](#)

they compare.

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 # MAE loss function
5 def mae_loss(y_pred, y_true):
6     abs_error = np.abs(y_pred - y_true)
7     sum_abs_error = np.sum(abs_error)
8     loss = sum_abs_error / y_true.size
9     return loss
10
11 # Plotting
12 x_vals = np.arange(-100, 100, 0.01)
13 y_vals = np.abs(x_vals)
14
15 plt.plot(x_vals, y_vals, "red")
16 plt.grid(True, which="major")
17 plt.show()
```

mae.py hosted with ❤ by GitHub

[view raw](#)

[Open in app](#)


Advantage. The beauty of the MAE is that its advantage directly covers the MSE disadvantage. Since we are taking the absolute value, all of the errors will be weighted on the same linear scale. Thus, unlike the MSE, we won't be putting too much weight on our outliers and our loss function provides a generic and even measure of how well our model is performing.

Disadvantage: If we do in fact care about the outlier predictions of our model, then the MAE won't be as effective. The large errors coming from the outliers end up being weighted the exact same as lower errors. This might results in our model being great most of the time, but making a few very poor predictions every so-often.

(3) Huber Loss

Now we know that the MSE is great for learning outliers while the MAE is great for ignoring them. But what about something in the middle?

Consider an example where we have a dataset of 100 values we would like our model to be trained to predict. Out of all that data, 25% of the expected values are 5 while the other 75% are 10.

An MSE loss wouldn't quite do the trick, since we don't really have "outliers"; 25% is by no means a small fraction. On the other hand we don't necessarily want to weight that 25% too low with an MAE. Those values of 5 aren't close to the median (10 — since 75% of the points have a value of 10), but they're also not really outliers.

Our solution?

The *Huber Loss Function*.

The Huber Loss offers the best of both worlds by balancing the MSE and MAE together. We can define it using the following piecewise function:

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

What this equation essentially says is: for loss values less than delta, use the MSE; for loss values greater than delta, use the MAE. This effectively combines the best of

[Open in app](#)

Using the MAE for larger loss values mitigates the weight that we put on outliers so that we still get a well-rounded model. At the same time we use the MSE for the smaller loss values to maintain a quadratic function near the centre.

This has the effect of magnifying the loss values as long as they are greater than 1. Once the loss for those data points dips below 1, the quadratic function down-weights them to focus the training on the higher-error data points.

Check out the code below for the Huber Loss Function. We also plot the Huber Loss beside the MSE and MAE to compare the difference.

```

1  import matplotlib.pyplot as plt
2  import numpy as np
3
4  # Huber loss function
5  def huber_loss(y_pred, y, delta=1.0):
6      huber_mse = 0.5*(y-y_pred)**2
7      huber_mae = delta * (np.abs(y - y_pred) - 0.5 * delta)
8      return np.where(np.abs(y - y_pred) <= delta, huber_mse, huber_mae)
9
10 # Plotting
11 x_vals = np.arange(-65, 65, 0.01)
12
13 delta = 1.5
14 huber_mse = 0.5*np.square(x_vals)
15 huber_mae = delta * (np.abs(x_vals) - 0.5 * delta)
16 y_vals = np.where(np.abs(x_vals) <= delta, huber_mse, huber_mae)
17
18 plt.plot(x_vals, y_vals, "green")
19 plt.grid(True, which="major")
20 plt.show()

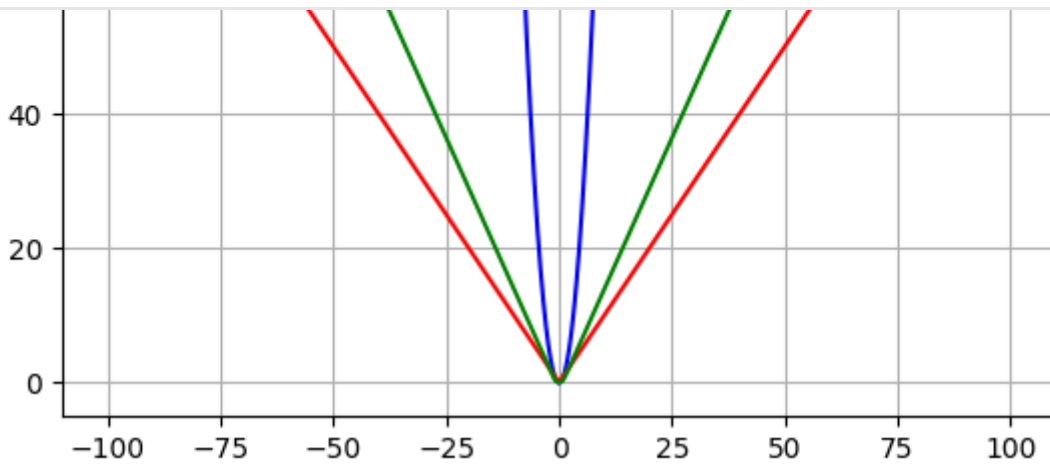
```

huber.py hosted with ❤ by GitHub

[view raw](#)

MSE



[Open in app](#)

MAE (red), MSE (blue), and Huber (green) loss functions

Notice how we're able to get the Huber loss right in-between the MSE and MAE.

Best of both worlds!

You'll want to use the Huber loss any time you feel that you need a balance between giving outliers some weight, but not too much. For cases where outliers are very important to you, use the MSE! For cases where you don't care at all about the outliers, use the MAE!

Like to learn?

Follow me on [twitter](#) where I post all about the latest and greatest AI, Technology, and Science! Connect with me on [LinkedIn](#) too!

Sign up for The Variable

By Towards Data Science

Every Thursday, the Variable delivers the very best of Towards Data Science: from hands-on tutorials and cutting-edge research to original features you don't want to miss. [Take a look.](#)

Get this newsletter

Emails will be sent to jywang.ieee@gmail.com.
[Not you?](#)

Open in app



Machine Learning

Data Science

Artificial Intelligence

Education

Technology

About Write Help Legal

Get the Medium app



Evaluation Metrics for Regression Algorithms (Along with their implementation in Python)



Venu Gopal Kadamba

Follow

Nov 27, 2020 · 7 min read

This article focuses on the evaluation metrics that are used to evaluate a Regression Algorithm along with their implementation in Python. At the end of this article you will get familiar with evaluation metrics for regression algorithms along with their implementation in python.



Evaluation of a model is the most important part to build an effective Machine Learning Model. Before diving into the topic, let us understand what is a regression algorithm.

What is a Regression Algorithm?

Regression algorithms fall under Supervised Machine Learning Algorithms.

Regression algorithms predicts a continuous value based on input features. For Example: House price prediction based on features of a house (number of bedrooms, house size, location, age of the house, year of renovation).

What are Evaluation Metrics?

Evaluation Metrics are used to measure the quality of a Machine Learning algorithm. There are many evaluation metrics present for different types of algorithms. We will be discussing about the evaluation metrics for Regression.

Evaluation Metrics for Machine Learning Regression Algorithms:

1. Mean Absolute Error
2. Mean Square Error
3. Root Mean Square Error
4. R² Score
5. Adjusted R² Score

Mean Absolute Error (MAE)

Mean Absolute Error is the average of the sum of absolute difference between the actual values and the predicted values. Mean Absolute Error is not sensitive to outliers. MAE should be used when you are solving a regression problem and don't want outliers to play a big role in the prediction. It can be useful if you know that the distribution of data is multimodal.

Let us look into the formula for Mean Absolute Error:

$$\text{MAE} = \frac{1}{N} \sum_{i=1}^N |y_i - \hat{y}_i|$$

Mean Absolute Error Formula

Let us breakdown the formula:

y_i = predicted value

\hat{y}_i = actual value

Here $(y_i - \hat{y}_i)$ is the error value and absolute value of the error is taken to remove any negative signs.

Let us look into the implementation part of Mean Absolute Error using python. Let X_{train} , y_{train} be the train data and X_{test} , y_{test} be the test data to evaluate our model.

A model with less MAE performs better than model with large MAE value.

```
# Importing all necessary libraries
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_absolute_error

# Initializing the model and fitting the model with train data
model = LinearRegression()
model.fit(X_train, y_train)

# Generating predictions over test data
predictions = model.predict(X_test)

# Evaluating the model using MAE Evaluation Metric
print(mean_absolute_error(y_test, predictions))
```

Mean Square Error (MSE)

Mean Square Error is the average of the sum of square of the difference between actual and predicted values.

MSE is most useful when the dataset contains outliers, or unexpected values (too high or too low values). So, it should be taken into consideration that if our model makes a single very bad prediction MSE magnifies the error.

MSE is least useful when a single bad prediction would ruin the entire model's predicting abilities, i.e. when the dataset contains a lot of noise.

The MSE has the units squared of whatever is plotted on vertical axis or y-axis. Since square of the error is taken in the function.

A large MSE value means that the data values are dispersed widely around the mean of the data and a small MSE value means that the data values are closely dispersed around the mean. i.e. A model with small MSE value has better performance.

Let us look into the formula for Mean Square Error:

$$MSE = \frac{1}{n} \sum \left(y - \hat{y} \right)^2$$

The square of the difference
between actual and
predicted

Mean Squared Error Formula

The squaring of the error ($y_i - \hat{y}_i$) is necessary to remove any negative signs and also gives more weight to large differences.

Let us look into the implementation part of Mean Square Error using python. Let X_{train} , y_{train} be the train data and X_{test} , y_{test} be the test data to evaluate our model.

```
# Importing all necessary libraries
import numpy as np
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error

# Defining our own MSE function
def own_mean_squared_error(actual, predictions):
    return ((predictions - actual) ** 2).mean()

# Initializing the model and fitting the model with train data
model = RandomForestRegressor(
    n_estimators = 100,
    criterion = 'mse'
)
model.fit(X_train, y_train)

# Generating predictions over test data
predictions = model.predict(X_test)

# Evaluating the model using MSE Evaluation Metric
print(mean_squared_error(y_test, predictions))
print(own_mean_squared_error(y_test, predictions))
```

Root Mean Square Error (RMSE)

Root Mean Square Error is same as Mean Square Error but root of the MSE is considered while evaluating the model. RMSE is more sensitive to the presence of false data .i.e. outliers. RMSE is most useful when large errors are present and they drastically effect the model performance. Since, RMSE assigns a higher weights to large errors.

RMSE is a frequently used evaluation metric for evaluating a model. Unlike MSE, Root Mean Square Error has the same unit of quantity plotted on vertical axis or y-axis. Since square root of the MSE value is taken in RMSE.

Let us look into the formula for Root Mean Square Error:

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

Root Mean Square Error Formula

There is no inbuilt function available to calculate Root Mean Square Error. Let us look into the implementation part of Root Mean Square Error by defining our own function. Let X_train, y_train be the train data and X_test, y_test be the test data to evaluate our model.

```
# Importing all necessary libraries
import numpy as np
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error

# Defining RMSE function
def root_mean_squared_error(actual, predictions):
    return np.sqrt(mean_squared_error(actual, predictions))

# Initializing the model and fitting the model with train data
model = RandomForestRegressor(
    n_estimators = 100,
    criterion = 'mse'
)
model.fit(X_train,y_train)
```

```
# Generating predictions over test data
predictions = model.predict(X_test)

# Evaluating the model using RMSE Evaluation Metric
print(root_mean_squared_error(y_test, predictions))
```

Note: For sklearn version $\geq 0.22.0$, sklearn.metrics has a mean_squared_error function with a squared kwarg (default value is True). Setting squared value to False will return RMSE value.

```
# For sklearn versions >= 0.22.0
print(mean_squared_error(y_test, predictions, squared = False))
```

R² Score

R² score also known as the coefficient of determination gives the measure of how good a model fits to a given dataset. It indicates how closer are the predicted values to the actual values.

Let us look into the formula for better understanding:

$$R^2 = 1 - \frac{SS_{RES}}{SS_{TOT}} = 1 - \frac{\sum_i (y_i - \hat{y}_i)^2}{\sum_i (y_i - \bar{y})^2}$$

R² Formula

Let's breakdown the formula and look into each term:

SS_{res} = Sum of Square of Residuals

SS_{tot} = Total Sum of Squares

The R² value ranges from $-\infty$ to 1. A model with negative R² value indicates that the best fit line is performing worse than the average fit line.

Let us look into the implementation for R² evaluation metric:

```
# Importing all necessary libraries
from sklearn.linear_model import LinearRegression
from sklearn.metrics import r2_score

# Initializing the model and fitting the model with train data
model = LinearRegression()
model.fit(X_train,y_train)

# Generating predictions over test data
predictions = model.predict(X_test)

# Evaluating the model using R2 Evaluation Metric
print(r2_score(y_test, predictions))
```

The major drawback of the R² metric is that, as the number of input features for the model increases the R² value also increases irrespective of the significance of the added feature with respect to output variable. i.e. even if the added feature has no correlation with the output variable, the R² value increases.

Adjusted R² Score

Adjusted R² is a modified form of R² that penalizes the addition of new independent variable or predictor and only increases if the new independent variable or predictor enhances the model performance.

Let us look into the formula for Adjusted R²:

$$Adjusted\ R^2 = 1 - (1 - R^2) * \frac{n - 1}{n - k - 1}$$

Adjusted R² Formula

Let us breakdown the formula and look into its each term:

R² : It is R² Score

n : Number of Samples in our Dataset

k : Number of Predictors

There is no inbuilt function to calculate only adjusted R². Let us look into the implementation part of Adjusted R²:

```

# Importing all necessary libraries
from sklearn.linear_model import LinearRegression
from sklearn.metrics import r2_score

# Defining the adjusted R2 function
def adjusted_r2_score(actual, predictions, num_pred, num_samples):
    n = num_samples
    k = num_pred
    r2 = r2_score(actual, predictions)
    adjusted_r2 = 1 - ((1-r2) * ((n-1)/(n-k-1)))
    return adjusted_r2

# Initializing the model and fitting the model with train data
model = LinearRegression()
model.fit(X_train, y_train)

# Generating predictions over test data
predictions = model.predict(X_test)

# Evaluating the model using Adjusted R2 Evaluation Metric
num_samples = X_test.shape[0]
num_predictors = X_test.shape[1]
adjusted_r2_score(y_test, predictions, num_predictors, num_samples)

```


Note: Adjusted R² will be always less than or equal to R² Score.

The above mentioned evaluation metrics are 5 most commonly used Evaluation Metrics for evaluating Regression Algorithms.

If you liked this article please follow me. If you noticed any mistakes in the formulas, code or in the content, please let me know.

You can find me at : [LinkedIn](#), [GitHub](#)

Venu Gopal Kadamba

Student  and Python Programmer. venugopalkadamba has 22 repositories available. Follow their code on GitHub.

github.com

Venu Gopal Kadamba

View Venu Gopal Kadamba's profile on LinkedIn, the world's largest professional community.

www.linkedin.com

Thank You!

Sign up for Analytics Vidhya News Bytes

By Analytics Vidhya

Latest news from Analytics Vidhya on our Hackathons and some of our best articles! [Take a look.](#)

Get this newsletter

Emails will be sent to jywang.ieee@gmail.com.

[Not you?](#)

Machine Learning

Deep Learning

Data Science

Artificial Intelligence

Evaluation

[About](#) [Write](#) [Help](#) [Legal](#)

Get the Medium app

