4 most common loss functions for Machine Learning Regression

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)

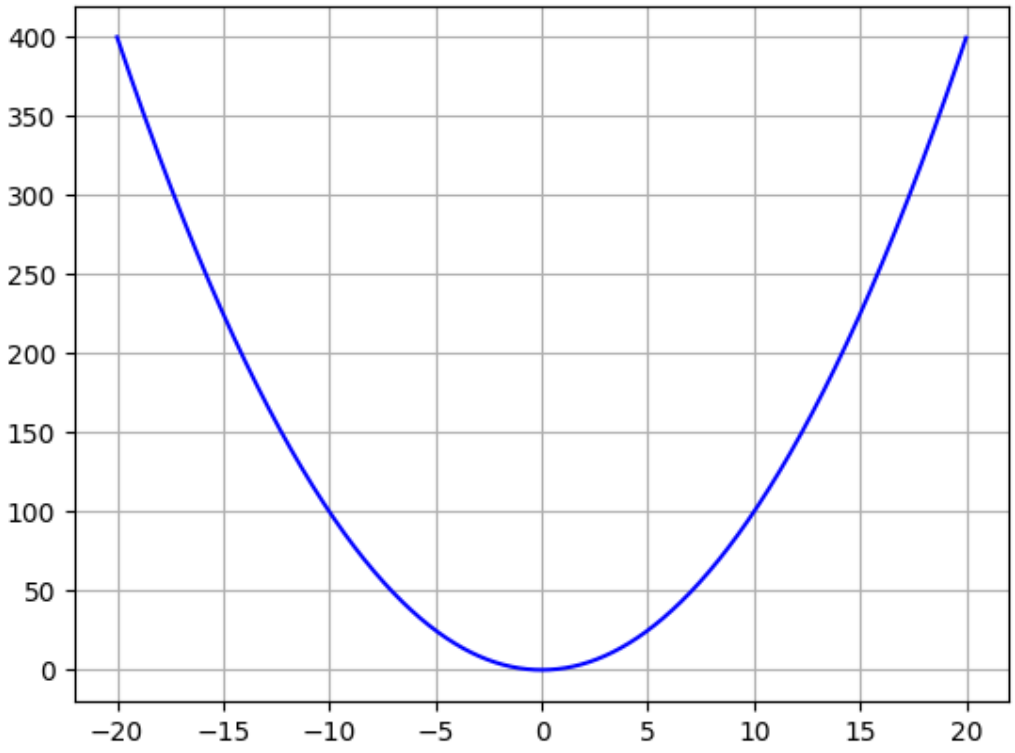
The Mean Squared Error (MSE) is perhaps the simplest and most common loss function, often taught in introductory Machine Learning courses. To calculate the MSE, you take the difference between your model's predictions and the ground truth, square it, and average it out across the whole dataset.

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:

```python
import numpy as np
import matplotlib.pyplot as plt

# MSE loss function
def mse_loss(y_pred, y_true):
    squared_error = (y_pred - y_true) ** 2
    sum_squared_error = np.sum(squared_error)
    loss = sum_squared_error / y_true.size
    return loss

# Plotting
x_vals = np.arange(-20, 20, 0.01)
y_vals = np.square(x_vals)

plt.plot(x_vals, y_vals, "blue")
plt.grid(True, which="major")
plt.show()
```



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 theses 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)

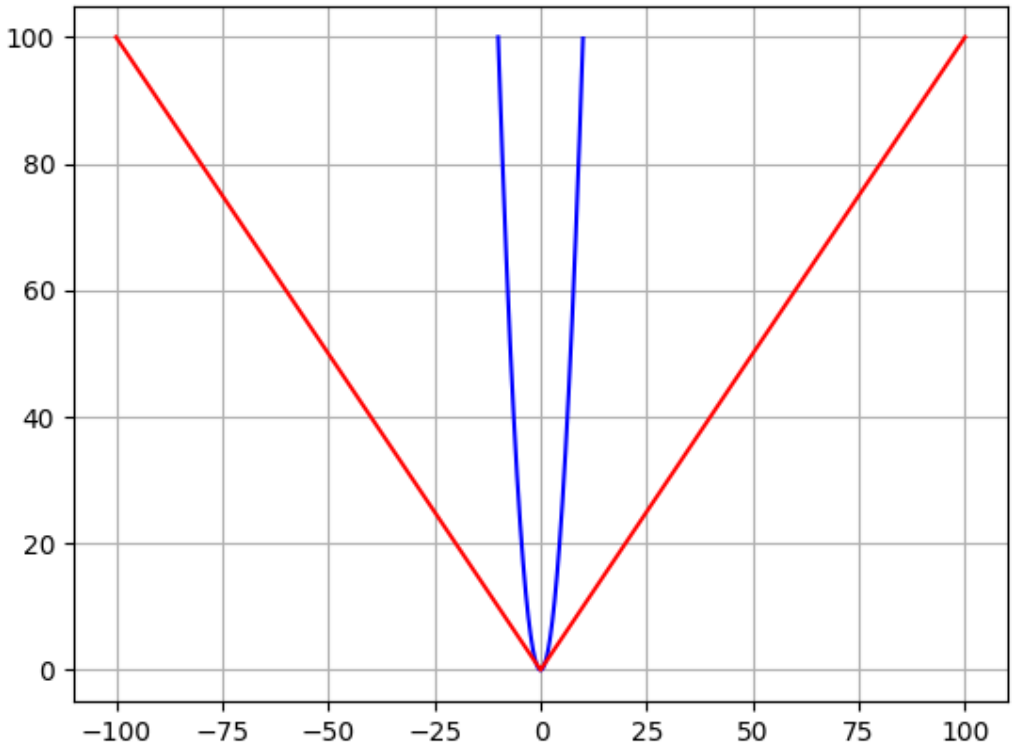
The Mean Absolute Error (MAE) is only slightly different in definition from the MSE, but interestingly provides almost exactly opposite properties! To calculate the MAE, you take the difference between your model's predictions and the ground truth, apply the absolute value to that difference, and then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

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

Once again our code is super easy in Python! We can write it in plain numpy and plot it using matplotlib. This time we'll plot it in red right on top of the MSE to see how they compare.

```python
import numpy as np
import matplotlib.pyplot as plt

# MAE loss function
def mae_loss(y_pred, y_true):
    abs_error = np.abs(y_pred - y_true)
    sum_abs_error = np.sum(abs_error)
    loss = sum_abs_error / y_true.size
    return loss

# Plotting
x_vals = np.arange(-100, 100, 0.01)
y_vals = np.abs(x_vals)

plt.plot(x_vals, y_vals, "red")
plt.grid(True, which="major")
plt.show()
```



MAE (red) and MSE (blue) loss functions

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.

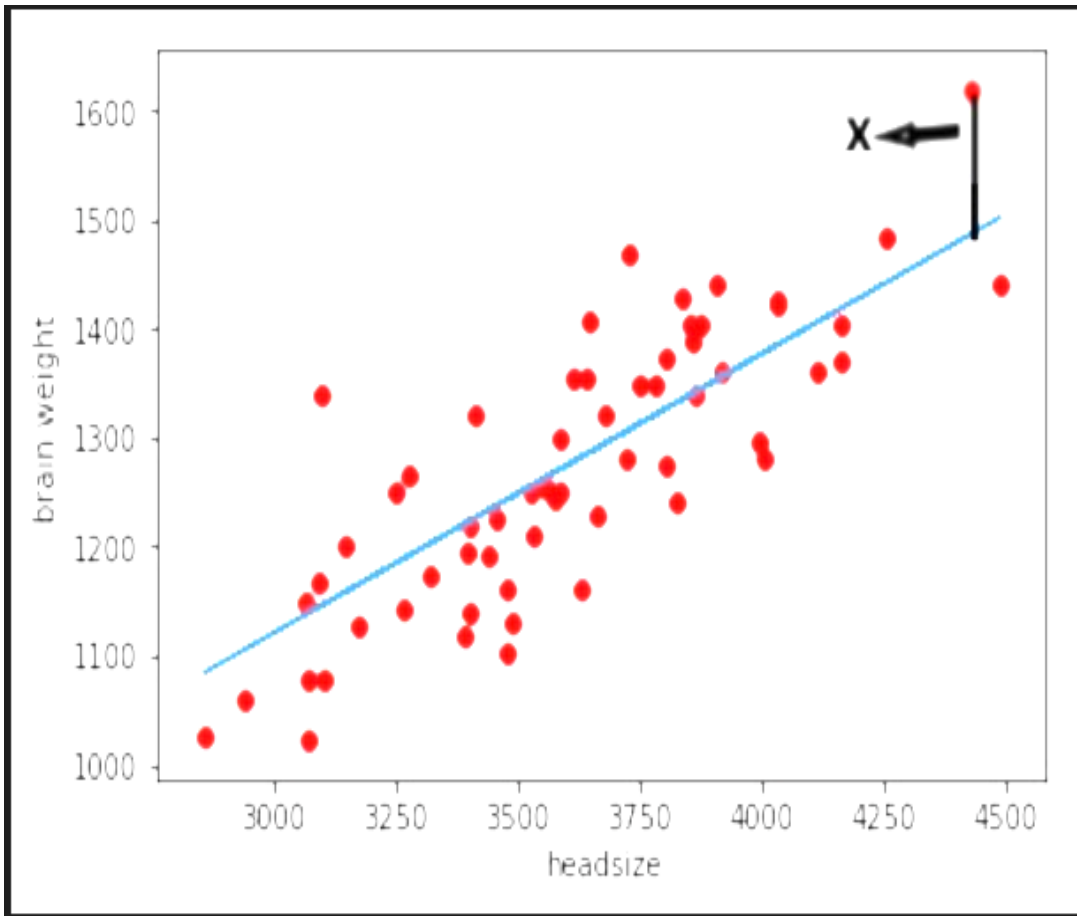
Disadvantage: If we do in fact care about the outlier predictions of our model, then the MAE won't be as effective. The large errors coming from the outliers end up being weighted the exact same as lower errors. This might results in our model being great most of the time, but making a few very poor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

(3) Root Mean Square Error

In a regression analysis, the difference between actual and projected values is gauged using a metric called root mean square error (RMSE).

Consider a scenario where we wish to anticipate a person's weight based on their height in order to succinctly describe RMSE. We utilise a regression model to estimate each person's weight from a dataset of ten individuals. We compare the anticipated weight to the actual weight after making predictions, then we compute the difference for each individual. The RMSE is the square root of the average of these differences' squares, which is what determines the RMSE.

```
import time
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

#reading the data
"""
here the directory of my code and the headbrain6.csv file
is same make sure both the files are stored in same folder or directory
"""
data=pd.read_csv('headbrain6.csv')
data.head()
x=data.iloc[:,2:3].values
y=data.iloc[:,3:4].values

#splitting the data into training and test
from sklearn.cross_validation import train_test_split
x_train,x_test,y_train,y_test=train_test_split(x,y,test_size=1/4,random_state=0)

#fitting simple linear regression to the training set
from sklearn.linear_model import LinearRegression
regressor=LinearRegression()
regressor.fit(x_train,y_train)

#predict the test result
y_pred=regressor.predict(x_test)

#to see the relationship between the training data values
plt.scatter(x_train,y_train,c='red')
plt.show()

#to see the relationship between the predicted
#brain weight values using scattered graph
plt.plot(x_test,y_pred)
plt.scatter(x_test,y_test,c='red')
plt.xlabel('headsize')
plt.ylabel('brain weight')

#errorin each value
for i in range(0,60):
print("Error in value number",i,(y_test[i]-y_pred[i]))
time.sleep(1)

#combined rmse value
rss=((y_test-y_pred)**2).sum()
mse=np.mean((y_test-y_pred)**2)
print("Final rmse value is =",np.sqrt(np.mean((y_test-y_pred)**2)))
```


1600
1500
1400
1300
1200
1100
1000
brain weight
3000
3250
3500
3750
4000
4250
4500
headsize
X

Advantage: RMSE is a commonly used statistic that assesses the precision of model predictions. It is simple to comprehend and interpret.

The RMSE penalises big deviations from the correct value more severely than small ones because it is sensitive to significant mistakes. This makes it a good statistic for models where costly or substantial impacts result from big inaccuracies.

RMSE may be used to assess the performance of the same model under various circumstances or to compare the performance of several models.

Disadvantage: The findings can be skewed by outliers since RMSE is sensitive to them. Even if the majority of the forecasts are right, a single significant inaccuracy might cause a high RMSE.

RMSE doesn't offer any details on the error's direction. For instance, RMSE won't give this information if the model continually overestimates or underestimates the real values.

RMSE bases its calculations on the supposition that the errors are regularly distributed, which may not always be the case. Other

measures, including mean absolute error (MAE), may be more appropriate in such circumstances.

(4) 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 both worlds from the two loss functions!

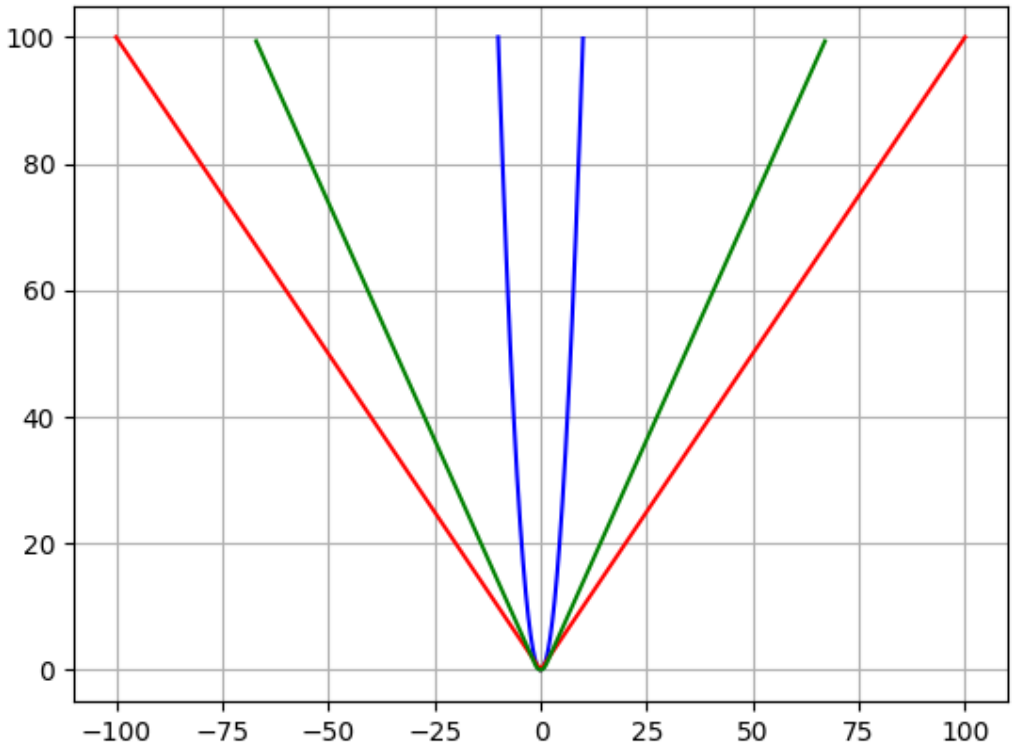
Using the MAE for larger loss values mitigates the weight that we put on outliers so that we still get a well-rounded model. At the same time we use the MSE for the smaller loss values to maintain a quadratic function near the centre.

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.

MSE

```python
import matplotlib.pyplot as plt
import numpy as np

# Huber loss function
def huber_loss(y_pred, y, delta=1.0):
    huber_mse = 0.5*(y-y_pred)**2
    huber_mae = delta * (np.abs(y - y_pred) - 0.5 * delta)
    return np.where(np.abs(y - y_pred) <= delta, huber_mse, huber_mae)

# Plotting
x_vals = np.arange(-65, 65, 0.01)

delta = 1.5
huber_mse = 0.5*np.square(x_vals)
huber_mae = delta * (np.abs(x_vals) - 0.5 * delta)
y_vals = np.where(np.abs(x_vals) <= delta, huber_mse, huber_mae)

plt.plot(x_vals, y_vals, "green")
plt.grid(True, which="major")
plt.show()
```



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!