



INT354: MACHINE LEARNING-I: NOTES

Unit V

Regression Metrics: R2 Score, Mean Absolute Error, Mean Squared Error, Mean Squared Logarithmic Error, Mean Absolute Percentage Error, Explained Variance Score, D2 Score Visual Evaluation of Regression Models

Regression Metrics:

Introduction

- Machine learning models aim to understand patterns within data, enabling predictions, answers to questions, or a deeper understanding of concealed patterns. This iterative learning process involves the model acquiring patterns, testing against new data, adjusting parameters, and repeating until achieving satisfactory performance. The evaluation phase, essential for regression models, employs loss functions.
- Role of Loss Functions in Model Evaluation
- Loss functions compare the model's predicted values with actual values, gauging its efficacy in mapping the relationship between X (feature) and Y (target). The loss, indicating the disparity between predicted and actual values, guides model refinement. A higher loss denotes poorer performance, demanding adjustments for optimal training.
- Crucial Factors in Choosing Loss Functions
- Selecting an appropriate loss function hinges on various factors such as the algorithm, data outliers, and the need for differentiability. With many options available, each with distinct properties, there is no universal solution. This article provides a comprehensive list of regression loss functions, outlining their



advantages and drawbacks. Implementable across various libraries, the code examples use NumPy for enhanced transparency into the underlying mechanisms.

Let's delve into the world of regression loss functions without delay.

Loss function vs Cost function

The loss function is a function that calculates loss for one data point.

$$\text{Squared Error} = (y_i - \hat{y}_i)^2$$

A loss function

The function that calculates loss for the entire data being used is called the cost function.

$$\text{Mean Squared Error} = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$$

A cost function

List of Top 13 Evaluation Metrics

Here is a list of 13 evaluation metrics

- Mean Absolute Error (MAE)
- Mean Bias Error (MBE)
- Relative Absolute Error (RAE)
- Mean Absolute Percentage Error (MAPE)
- Mean Squared Error (MSE)



- Root Mean Squared Error (RMSE)
- Relative Squared Error (RSE)
- Normalized Root Mean Squared Error (NRMSE)
- Relative Root Mean Squared Error (RRMSE)
- Root Mean Squared Logarithmic Error (RMSLE)
- Hyber Loss
- Log Cosh Loss
- Quantile Loss

Mean Absolute Error (MAE)

Mean absolute error, or L1 loss, stands out as one of the simplest and easily comprehensible loss functions and evaluation metrics. It computes by averaging the absolute differences between predicted and actual values across the dataset.

Mathematically, it represents the arithmetic mean of absolute errors, focusing solely on their magnitude, irrespective of direction. A lower MAE indicates superior model accuracy.

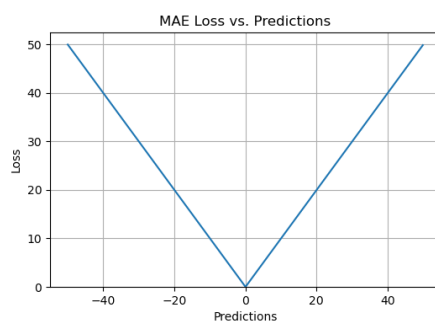
MAE formula is:

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

where

- y_i = actual value
- \hat{y}_i = predicted value
- n = sample size

Python Code:



Pros of the MAE Evaluation Metric:

- It is an easy to calculate evaluation metric.
- All the errors are weighted on the same scale since absolute values are taken.
- It is useful if the training data has outliers as MAE does not penalize high errors caused by outliers.
- It provides an even measure of how well the model is performing.

Cons of the MAE evaluation metric:

- Sometimes the large errors coming from the outliers end up being treated as the same as low errors.



- MAE follows a scale-dependent accuracy measure where it uses the same scale as the data being measured. Hence it cannot be used to compare series' using different measures.
- One of the main disadvantages of MAE is that it is not differentiable at zero. Many optimization algorithms tend to use differentiation to find the optimum value for parameters in the evaluation metric.
- It can be challenging to compute gradients in MAE.

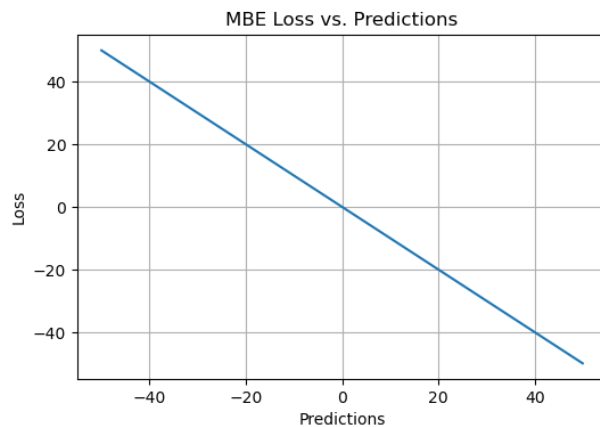
Mean Bias Error (MBE)

In “Mean Bias Error,” bias reflects the tendency of a measurement process to overestimate or underestimate a parameter. It has a single direction, positive or negative. Positive bias implies an overestimated error, while negative bias implies an underestimated error. Mean Bias Error (MBE) calculates the mean difference between predicted and actual values, quantifying overall bias without considering absolute values. Similar to MAE, MBE differs in not taking the absolute value. Caution is needed with MBE, as positive and negative errors can cancel each other out.

The formula for MBE:

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

```
def mean_bias_error(true, pred):  
    bias_error = true - pred  
    mbe_loss = np.mean(np.sum(diff) / true.size)  
    return mbe_loss
```



Pros of the MBE Evaluation Metric

- MBE is a good measure if you want to check the direction of the model (i.e. whether there is a positive or negative bias) and rectify the model bias.

Cons of the MBE evaluation Metric

- It is not a good measure in terms of magnitude as the errors tend to compensate each other.
- It is not highly reliable because sometimes high individual errors produce low MBE.
- As an evaluation metric, it can be consistently wrong in one direction. For example, if you're trying to predict traffic patterns and it always shows lower traffic than what is actually observed.

Relative Absolute Error (RAE)

Relative Absolute Error is calculated by dividing the total absolute error by the absolute difference between the mean and the actual value. The formula for RAE is:

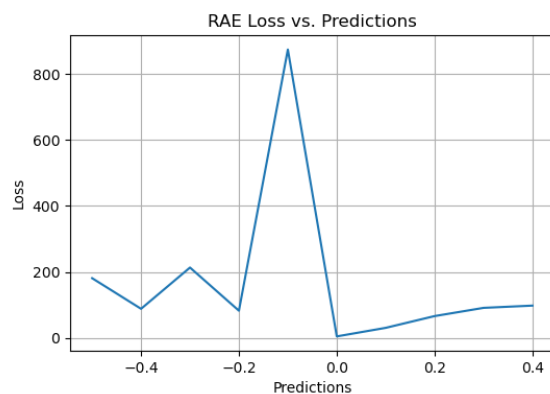
$$\text{RAE} = \frac{\sum_{i=1}^n |y_i - \hat{y}_i|}{\sum_{i=1}^n |y_i - \bar{y}|}$$

$$\text{where } \bar{y} = \frac{1}{n} \sum_{i=1}^n y_i$$

where \bar{y} is the mean of the n actual values.

RAE measures the performance of a predictive model and is expressed in terms of a ratio. The value of RAE can range from zero to one. A good model will have values close to zero, with zero being the best value. This error shows how the mean residual relates to the mean deviation of the target function from its mean.

```
def relative_absolute_error(true, pred):
    true_mean = np.mean(true)
    squared_error_num = np.sum(np.abs(true - pred))
    squared_error_den = np.sum(np.abs(true - true_mean))
    rae_loss = squared_error_num / squared_error_den
    return rae_loss
```



Pros of the RAE Evaluation Metric

- RAE can be used to compare models where errors are measured in different units.
- In some cases, RAE is reliable as it offers protection from outliers.

Cons of the RAE evaluation Metric

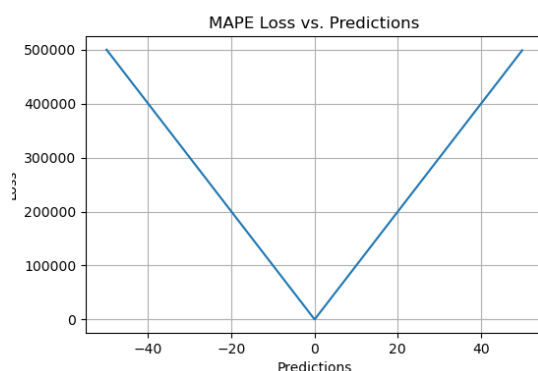
- One main drawback of RAE is that it can be undefined if the reference forecast is equal to the ground truth.

Mean Absolute Percentage Error (MAPE)

Calculate Mean Absolute Percentage Error (MAPE) by dividing the absolute difference between the actual and predicted values by the actual value. This absolute percentage is averaged across the dataset. MAPE, also known as Mean Absolute Percentage Deviation (MAPD), increases linearly with error. Lower MAPE values indicate better model performance.

$$\text{MAPE} = \frac{1}{n} \sum_{i=1}^n \frac{|y_i - \hat{y}_i|}{y_i} \cdot 100\%$$

```
def mean_absolute_percentage_error(true, pred):
    abs_error = (np.abs(true - pred)) / true
    sum_abs_error = np.sum(abs_error)
    mape_loss = (sum_abs_error / true.size) * 100
    return mape_loss
```





Pros of the MAPE Evaluation Metric

- MAPE is independent of the scale of the variables since its error estimates are in terms of percentage.
- All errors are normalized on a common scale and it is easy to understand.
- As MAPE uses absolute percentage errors, the problem of positive values and the negative values canceling each other out is avoided.

Cons of the MAPE evaluation Metric

- MAPE faces a critical problem when the denominator becomes zero, resulting in a “division by zero” challenge.
- MAPE exhibits bias by penalizing negative errors more than positive errors, potentially favoring methods with lower values.
- Due to the division operation, MAPE’s sensitivity to alterations in actual values leads to varying loss for the same error. For example, an actual value of 100 and a predicted value of 75 result in a 25% loss, while an actual value of 50 and a predicted value of 75 yield a higher 50% loss, despite the identical error of 25.

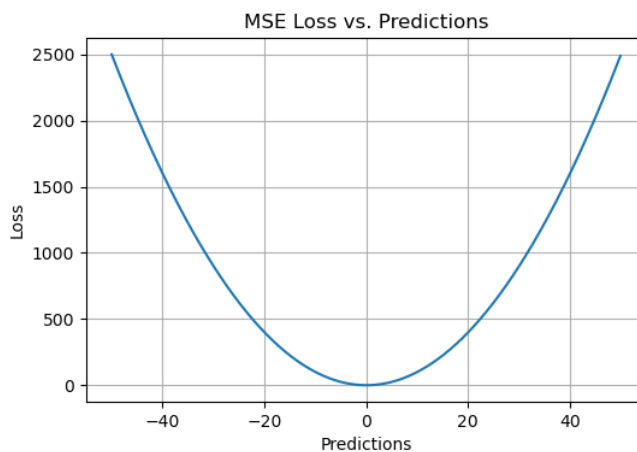
Mean Squared Error (MSE)

MSE is one of the most common regression loss functions. In Mean Squared Error also known as L2 loss, we calculate the error by squaring the difference between the predicted value and actual value and averaging it across the dataset. MSE is also known as Quadratic loss as the penalty is not proportional to the error but to the square of the error. Squaring the error gives higher weight to the outliers, which results in a smooth gradient for small errors. Optimization algorithms benefit from this penalization for large errors as it is helpful in finding the optimum values for parameters. MSE will never be

negative since the errors are squared. The value of the error ranges from zero to infinity. MSE increases exponentially with an increase in error. A good model will have an MSE value closer to zero.

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

```
def mean_squared_error(true, pred):  
    squared_error = np.square(true - pred)  
    sum_squared_error = np.sum(squared_error)  
    mse_loss = sum_squared_error / true.size  
    return mse_loss
```



Pros of the MSE Evaluation Metric

- MSE values are expressed in quadratic equations. Hence when we plot it, we get a gradient descent with only one global minima.
- For small errors, it converges to the minima efficiently. There are no local minima.
- MSE penalizes the model for having huge errors by squaring them.



- It is particularly helpful in weeding out outliers with large errors from the model by putting more weight on them.

Cons of the MSE evaluation Metric

- One of the advantages of MSE becomes a disadvantage when there is a bad prediction. The sensitivity to outliers magnifies the high errors by squaring them.
- MSE will have the same effect for a single large error as too many smaller errors. But mostly we will be looking for a model which performs well enough on an overall level.
- MSE is scale-dependent as its scale depends on the scale of the data. This makes it highly undesirable for comparing different measures.
- When a new outlier is introduced into the data, the model will try to take in the outlier. By doing so it will produce a different line of best fit which may cause the final results to be skewed.

Root Mean Squared Error (RMSE)

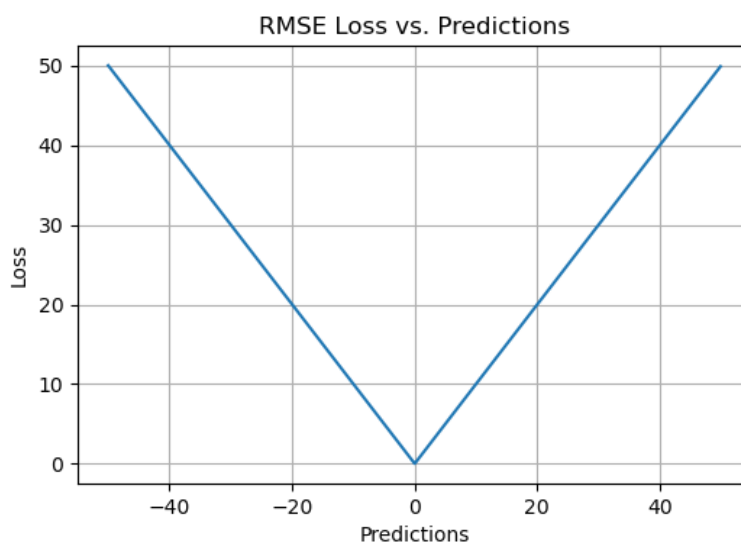
Root Mean Squared Error (RMSE) is a popular metric used in machine learning and statistics to measure the accuracy of a predictive model. It quantifies the differences between predicted values and actual values, squaring the errors, taking the mean, and then finding the square root. RMSE provides a clear understanding of the model's performance, with lower values indicating better predictive accuracy.

It is computed by taking the square root of MSE. RMSE is also called the Root Mean Square Deviation. It measures the average magnitude of the errors and is concerned with the deviations from the actual value. RMSE value with zero indicates that the model has

a perfect fit. The lower the RMSE, the better the model and its predictions. A higher RMSE indicates that there is a large deviation from the residual to the ground truth. RMSE can be used with different features as it helps in figuring out if the feature is improving the model's prediction or not.

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

```
def root_mean_squared_error(true, pred):  
    squared_error = np.square(true - pred)  
    sum_squared_error = np.sum(squared_error)  
    rmse_loss = np.sqrt(sum_squared_error / true.size)  
    return rmse_loss
```



Pros of the RMSE Evaluation Metric

- RMSE is easy to understand.
- It serves as a heuristic for training models.
- It is computationally simple and easily differentiable which many optimization algorithms desire.

- RMSE does not penalize the errors as much as MSE does due to the square root.

Cons of the RMSE Metric

- Like MSE, RMSE is dependent on the scale of the data. It increases in magnitude if the scale of the error increases.
- One major drawback of RMSE is its sensitivity to outliers and the outliers have to be removed for it to function properly.
- RMSE increases with an increase in the size of the test sample. This is an issue when we calculate the results on different test samples.

Relative Squared Error (RSE)

In order to calculate Relative Squared Error, you take the Mean Squared Error (MSE) and divide it by the square of the difference between the actual and the mean of the data.

In other words, we divide the MSE of our model by the MSE of a model which uses the mean as the predicted value.

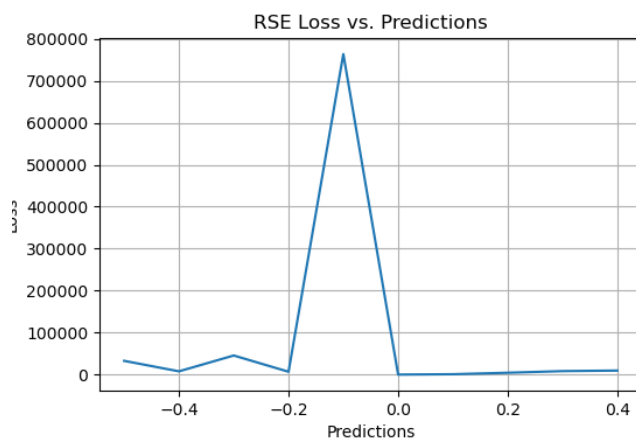
$$\text{RSE} = \frac{\sum_{i=1}^n (y_i - \hat{y}_i)^2}{\sum_{i=1}^n (y_i - \bar{y})^2}$$

$$\text{where } \bar{y} = \frac{1}{n} \sum_{i=1}^n y_i$$

```
def relative_squared_error(true, pred):
```

```
true_mean = np.mean(true)
squared_error_num = np.sum(np.square(true - pred))
squared_error_den = np.sum(np.square(true - true_mean))
rse_loss = squared_error_num / squared_error_den
return rse_loss
```

The output value of RSE is expressed in terms of ratio. It can range from zero to one. A good model should have a value close to zero while a model with a value greater than 1 is not reasonable.



Pros of the RSE Evaluation Metric

- RSE is not scale-dependent. Hence it can be used to compare between models where errors are measured in different units.
- RSE is not sensitive to the mean and the scale of predictions.

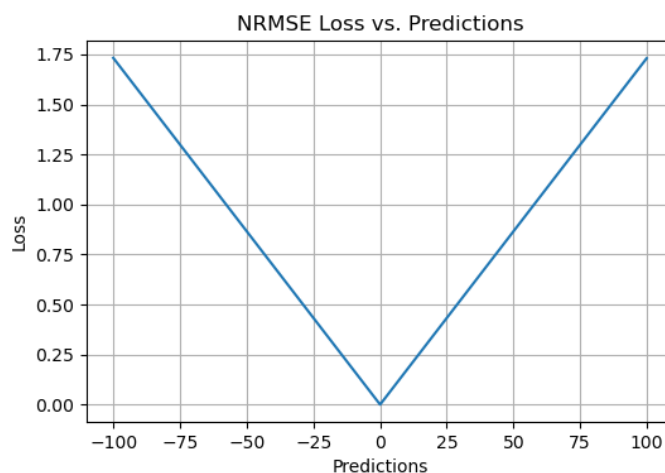
Normalized Root Mean Squared Error (NRMSE)

The Normalized RMSE is generally computed by dividing a scalar value. It can be in different ways like,

- RMSE / maximum value in the series
- RMSE / mean

- RMSE / difference between the maximum and the minimum values (if mean is zero)
- RMSE / standard deviation
- RMSE / interquartile range

```
# implementation of NRMSE with standard deviation
def normalized_root_mean_squared_error(true, pred):
    squared_error = np.square((true - pred))
    sum_squared_error = np.sum(squared_error)
    rmse = np.sqrt(sum_squared_error / true.size)
    nrmse_loss = rmse/np.std(pred)
    return nrmse_loss
```



Opting for the interquartile range can be the most suitable choice, especially when dealing with outliers. NRMSE proves effective for comparing models with different dependent variables or when modifications like log transformation or standardization occur. This metric addresses scale-dependency issues, facilitating comparisons across models of varying scales or datasets.

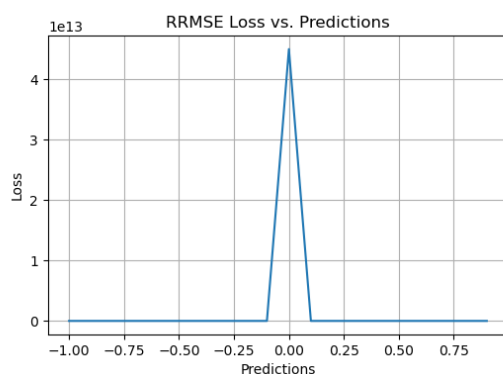
Relative Root Mean Squared Error (RRMSE)

Relative Root Mean Squared Error (RRMSE) is a variant of Root Mean Squared Error (RMSE), gauging predictive model accuracy relative to the target variable range. It normalizes RMSE by the target variable range and presents it as a percentage for easy cross-dataset or cross-variable comparison. RRMSE, a dimensionless form of RMSE, scales residuals against actual values, allowing comparison of different measurement techniques.

- Excellent when RRMSE < 10%
- Good when RRMSE is between 10% and 20%
- Fair when RRMSE is between 20% and 30%
- Poor when RRMSE > 30%

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

```
def relative_root_mean_squared_error(true, pred):
    num = np.sum(np.square(true - pred))
    den = np.sum(np.square(pred))
    squared_error = num/den
    rmse_loss = np.sqrt(squared_error)
    return rmse_loss
```



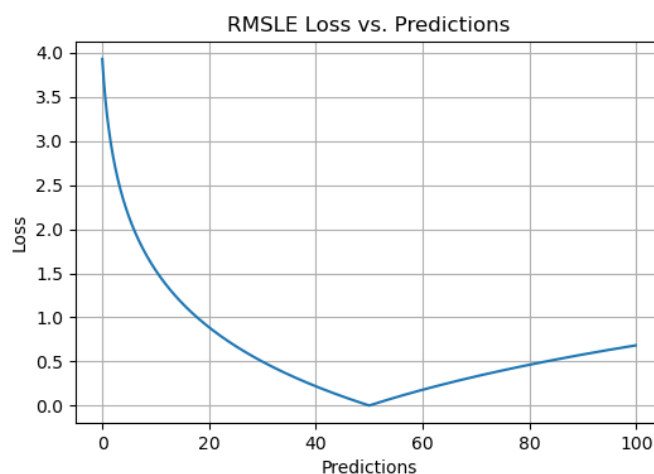
Root Mean Squared Logarithmic Error (RMSLE)

Root Mean Squared Logarithmic Error is calculated by applying log to the actual and the predicted values and then taking their differences. RMSLE is robust to outliers where the small and the large errors are treated evenly.

It penalizes the model more if the predicted value is less than the actual value while the model is less penalized if the predicted value is more than the actual value. It does not penalize high errors due to the log. Hence the model has a large penalty for underestimation than overestimation. This can be helpful in situations where we are not bothered by overestimation but underestimation is not acceptable.

$$\text{RMSLE} = \sqrt{(\log(y_i + 1) - \log(\hat{y}_i + 1))^2}$$

```
def root_mean_squared_log_error(true, pred):  
    square_error = np.square((np.log(true + 1) - np.log(pred + 1)))  
    mean_square_log_error = np.mean(square_error)  
    rmsle_loss = np.sqrt(mean_square_log_error)  
    return rmsle_loss
```





Pros of the RMSLE Evaluation Metric

- RMSLE is not scale-dependent and is useful across a range of scales.
- It is not affected by large outliers.
- It considers only the relative error between the actual value and the predicted value.

Cons of the RMSLE evaluation Metric

- It has a biased penalty where it penalizes underestimation more than overestimation.

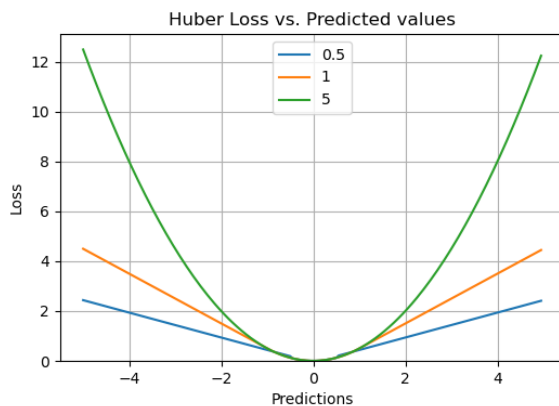
Huber Loss

What if you want a function that learns about the outliers as well as ignores them? Well, Huber loss is the one for you. Huber loss is a combination of both linear and quadratic scoring methods. It has a hyperparameter delta (δ) which can be tuned according to the data. The loss will be linear (L1 loss) for values above delta and quadratic (L2 loss) for values below it. It balances and combines good properties of both MAE (Mean Absolute Error) and MSE (Mean Squared Error).

In other words, for loss values less than delta, MSE will be used and for loss values greater than delta, MAE will be used. The choice of delta (δ) is extremely critical because it defines our choice of the outlier. Huber loss reduces the weight we put on outliers for larger loss values by using MAE while for smaller loss values it maintains a quadratic function using MSE.

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

```
def huber_loss(true, pred, delta):
    huber_mse = 0.5 * np.square(true - pred)
    huber_mae = delta * (np.abs(true - pred) - 0.5 * (np.square(delta)))
    return np.where(np.abs(true - pred) <= delta, huber_mse, huber_mae)
```



Pros of the Huber Loss Evaluation Metric

- It is differentiable at zero.
- Outliers are handled properly due to the linearity above delta.
- The hyperparameter, δ can be tuned to maximize model accuracy.

Cons of the Huber Loss evaluation Metric

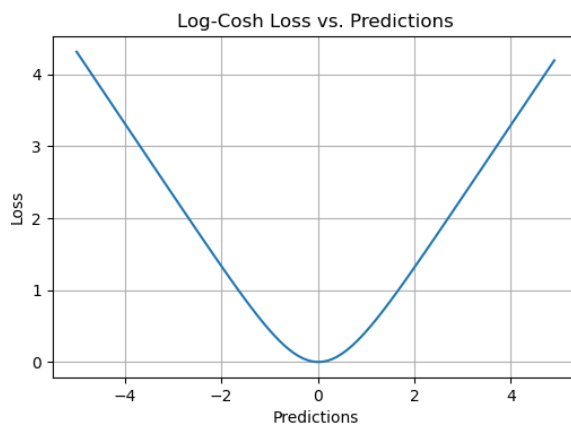
- The additional conditionals and comparisons make Huber loss computationally expensive for large datasets.
- In order to maximize model accuracy, δ needs to be optimized and it is an iterative process.
- It is differentiable only once.

Log Cosh Loss

Log cosh calculates the logarithm of the hyperbolic cosine of the error. This function is smoother than quadratic loss. It works like MSE but is not affected by large prediction errors. It is quite similar to Huber loss in the sense that it is a combination of both linear and quadratic scoring methods.

$$\text{Logcosh}(t) = \sum_{i=1}^n \log(\cosh(\hat{y}_i - y_i))$$

```
def log_cosh(true, pred):
    logcosh = np.log(np.cosh(pred - true))
    logcosh_loss = np.sum(logcosh)
    return logcosh_loss
```



Pros of the Log Cosh Loss Evaluation Metric

- It has the advantages of Huber loss while being twice differentiable everywhere. Some optimization algorithms like XGBoost favors double differentials over functions like Huber which can be differentiable only once.
- It requires fewer computations than Huber.

Cons of the Log Cosh Loss evaluation Metric

- It is less adaptive as it follows a fixed scale.
- Compared to Huber loss, the derivation is more complex and requires much in-depth study.

Quantile Loss

Quantile regression loss function is applied to predict quantiles. The quantile is the value that determines how many values in the group fall below or above a certain limit. It estimates the conditional median or *quantile* of the response(dependent) variables across values of the predictor(independent) variables. The loss function is an extension of MAE except for the 50th percentile, where it is MAE. It provides prediction intervals even for residuals with non-constant variance and it does not assume a particular parametric distribution for the response.

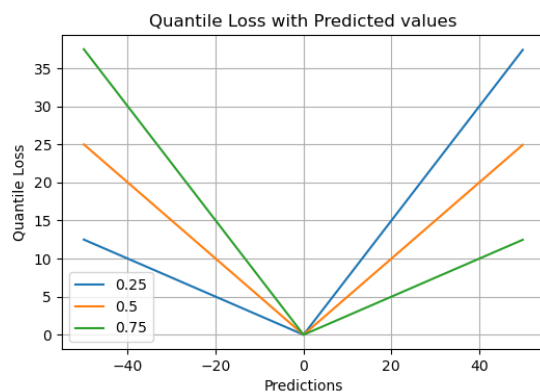
$$L_{Quantile} = \sum_{i=y_i < \hat{y}_i} (\gamma - 1)|y_i - \hat{y}_i| + \sum_{i=y_i \geq \hat{y}_i} (\gamma)|y_i - \hat{y}_i|$$

γ represents the required quantile. The quantiles values are selected based on how we want to weigh the positive and the negative errors.

In the loss function above, γ has a value between 0 and 1. When there is an underestimation, the first part of the formula will dominate and for overestimation, the second part will dominate. The chosen value of quantile(γ) gives different penalties for over-prediction and under prediction. When $\gamma = 0.5$, underestimation and overestimation are penalized by the same factor and the median is obtained. When the value of γ is larger, overestimation is penalized more than underestimation. For example, when $\gamma = 0.75$ the model will penalize overestimation and it will cost three times as much as

underestimation. Optimization algorithms based on gradient descent learn from the quantiles instead of the mean.

```
def quantile_loss(true, pred, gamma):
    val1 = gamma * np.abs(true - pred)
    val2 = (1-gamma) * np.abs(true - pred)
    q_loss = np.where(true >= pred, val1, val2)
    return q_loss
```



Pros of the Quantile Loss Evaluation Metric

- It is particularly useful when we are predicting an interval instead of point estimates.
- This function can also be used to calculate prediction intervals in neural nets and tree-based models.
- It is robust to outliers.

Cons of the Quantile Loss evaluation Metric

- Quantile loss is computationally intensive.
- If we use a squared loss to measure the efficiency or if we are to estimate the mean, then quantile loss will be worse.

R2 Score:

The most important thing we do after making any model is evaluating the model. We have different evaluation matrices for evaluating the model. However, the choice of evaluation matrix to use for evaluating the model depends upon the type of problem we are solving whether it's a regression, classification, or any other type of problem. In this article, we will explain R-Square for regression analysis problems.

What is R-Squared

R-squared is a statistical measure that represents the goodness of fit of a regression model. The value of R-square lies between 0 to 1. Where we get R-square equals 1 when the model perfectly fits the data and there is no difference between the predicted value and actual value. However, we get R-square equals 0 when the model does not predict any variability in the model and it does not learn any relationship between the dependent and independent variables.

How is R-Squared Calculated

R-squared also known as the *coefficient of determination* measures the variability in the dependent variable **Y** that is being explained by the independent variables **X_i** in the regression model.

We calculate R-Square in the following steps

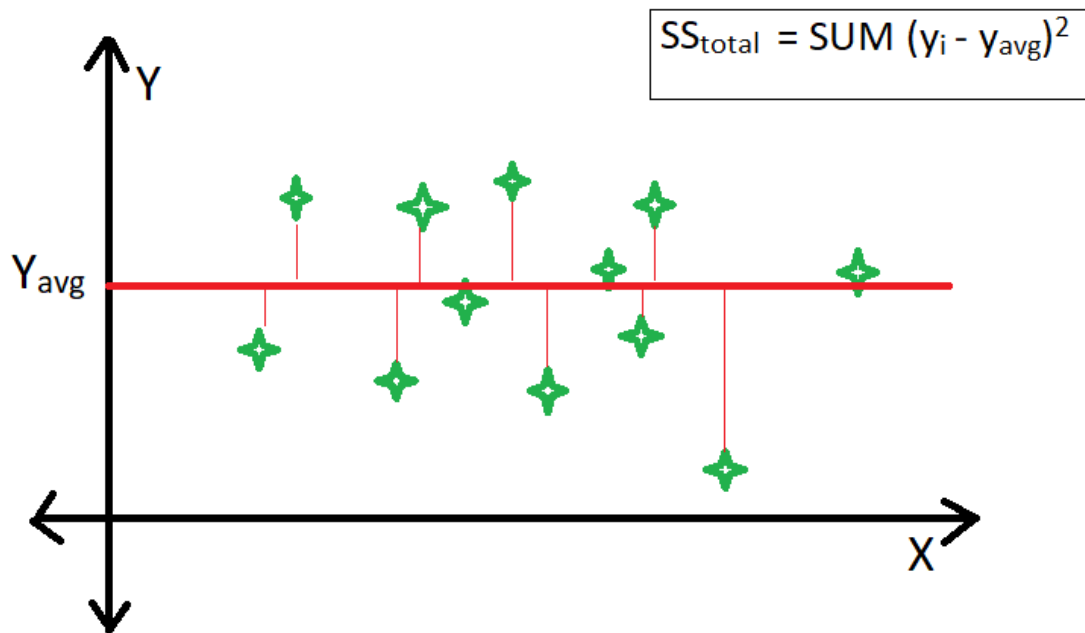
1. First, calculate the mean of the target/dependent variable **y** and we denote it by \bar{y}
2. Calculate the total sum of squares by subtracting each observation y_i from \bar{y} , then squaring it and summing these square differences across all the values. It is denoted by
$$SS_{tot} = \sum_{i=1}^n (y_i - \bar{y})^2$$
3. We estimate the model parameter using a suitable regression model such as Linear Regression or SVM Regressor
4. We calculate the Sum of squares due to regression which is denoted by SSR. This is calculated by subtracting each predicted value of **y** denoted by \hat{y}_i from y_i squaring these differences and then summing all the n terms.
$$SSR = \sum_{i=1}^n (\hat{y}_i - \bar{y})^2$$
5. We calculate the sum of squares (SS_{res}). It explains unaccounted variability in the dependent **y** after predicting these values from an independent variable in the model.
$$SS_{res} = \sum_{i=1}^n (y_i - \hat{y}_i)^2$$

$$R^2 = \frac{SSR}{SS_{tot}} \text{ or } R^2 = 1 - \frac{SS_{res}}{SS_{tot}}$$

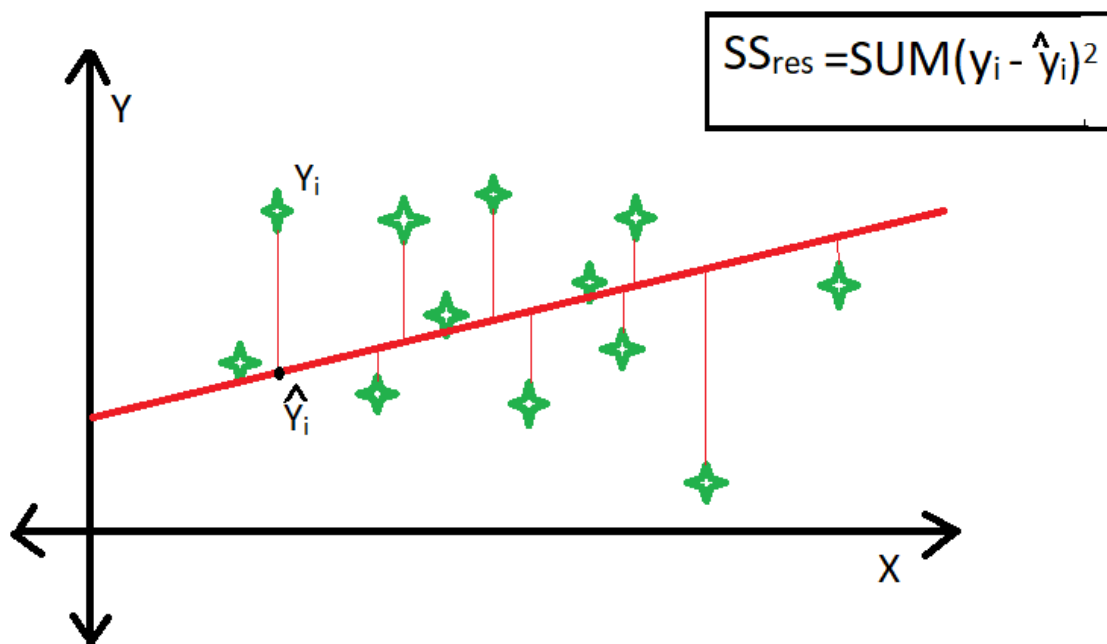
6. we can then use either

R-Squared Goodness Of Fit

R-square is a comparison of the residual sum of squares (SS_{res}) with the total sum of squares (SS_{tot}). The total sum of squares is calculated by summation of squares of perpendicular distance between data points and the average line.



The residual sum of squares is calculated by the summation of squares of perpendicular distance between data points and the best-fitted line.



R square is calculated by using the following formula :

$$R^2 = 1 - \frac{SS_{res}}{SS_{tot}}$$



Where SS_{res} is the residual sum of squares and SS_{tot} is the total sum of squares. The goodness of fit of regression models can be analyzed on the basis of the R-square method. The more the value of the r-square near 1, the better the model is.

Note: The value of R-square can also be negative when the model fitted is worse than the average fitted model.

Limitation of using the R-square method –

- The value of r-square always increases or remains the same as new variables are added to the model, without detecting the significance of this newly added variable (i.e. the value of r-square never decreases on the addition of new attributes to the model). As a result, non-significant attributes can also be added to the model with an increase in the r-square value.
- This is because SS_{tot} is always constant and the regression model tries to decrease the value of SS_{res} by finding some correlation with this new attribute hence the overall value of r-square increases, which can lead to a poor regression model.

R-Squared Vs Adjusted R-Squared

Adjusted R-Squared is an updated version of R-squared which takes account of the number of independent variables while calculating R-squared. The main problem with R-squared is that the R-Square value always increases with an increase in independent variables irrespective of the fact that where the independent variable is contributing to the model or not. This leads to the model having high variance if the model has a lot of independent variables.

Formula For Adjusted R-Squared

$$AdjustedR^2 = 1 - \frac{(1-R^2) \cdot (n-1)}{n-k-1}$$

- Here R-square is the value that we calculate using the method explained above
- n is the total number of observations in the data
- k is the number of independent variables (predictors) in the regression model

Mean Absolute Error:

What is the Mean Absolute Error?

Mean Absolute Error (MAE) is a measure of the average size of the mistakes in a collection of predictions, without taking their direction into account. It is measured as the average absolute difference between the predicted values and the actual values and is used to assess the effectiveness of a regression model.

The MAE loss function formula:

- $MAE = (1/n) \sum_{i=1}^n |y_i - \hat{y}_i|$

where:

- n is the number of observations in the dataset.



- y_i is the true value.
- \hat{y}_i is the predicted value.

The MAE is a linear score, meaning all individual differences contribute equally to the mean. It provides an estimate of the size of the inaccuracy, but not its direction (e.g., over or under-prediction).

The importance of MAE

The Mean Absolute Error (MAE) is a crucial performance statistic for regression models since it is an easy-to-understand, interpretable, and reliable tool for assessing the accuracy of predictions. Among the many reasons, these are the most significance:

- **Resiliency to outliers.** The MAE is not as impacted by extreme results as other metrics, such as Mean Squared Error (MSE), are. This makes it an appropriate measure for datasets that include outliers or extreme values.
- **Linear score.** All individual differences are given equal weight in the average. This makes it simple to compare the performance of several models or variations of the same model.
- **Straightforward.** The MAE interpretation is a basic and obvious statistic that represents the average magnitude of the forecasts' mistakes. It is simple for non-technical stakeholders to comprehend.
- **Same units as the response variable.** The ML MAE is expressed in the same units as the response variable, making it simple to comprehend the size of the prediction error. This is important when trying to understand the performance of the model in the context of the issue you're attempting to solve.
- **Used in several disciplines.** MAE is used in finance, engineering, and meteorology. It is even regarded as a standard measure in some instances.
- **Offer information about the size of the error.** MAE provides information about the magnitude of the error produced by the model. It allows for model comparison and the selection of the best one, as well as the improvement of a model by determining the predicted mean absolute percentage error.

In conclusion, MAE is a commonly used and significant performance metric for regression models because it is intuitive, interpretable, resistant to

Mean Absolute Error in Python

The `mean absolute error()` method of the `sklearn.metrics` module in Python may be used to compute the MAE of a series of predictions.

The following demonstrates how to use the `mean absolute error()` function:

```
from sklearn.metrics import mean_absolute_error
```

```
import numpy as np
```

```
# Generate some sample data
```

```
y_true = np.array([1, 2, 3, 4, 5])
```

```
y_pred = np.array([1.5, 2.5, 2.8, 4.2, 4.9])
```

```
# Calculate the MAE
```

```
mae = mean_absolute_error(y_true, y_pred)
```

```
print("Mean Absolute Error:", mae)
```

This example provides some sample data and then uses the mean absolute error() function to determine the MAE of the forecasts. The first argument represents the actual values, while the second represents the expected values.

This code requires the Scikit-learn package to be installed in your Python environment.

The mean absolute error() method may also be used to determine MAE for multi-output issues. The first input is an array of actual values, while the second is of anticipated values.

Mean Squared Error:

Mean Square Error (MSE)

In the fields of regression analysis and machine learning, the Mean Square Error (MSE) is a crucial metric for evaluating the performance of predictive models. It measures the average squared difference between the predicted and the actual target values within a dataset. The primary objective of the MSE is to assess the quality of a model's predictions by measuring how closely they align with the ground truth.

Mathematical Formula

The MSE is calculated using the following formula

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^n (Y_i - \hat{Y}_i)^2$$

MSE = Mean Square error
n = Number of Data points

Y_i = Observed Values

\hat{Y} = Predicted Values

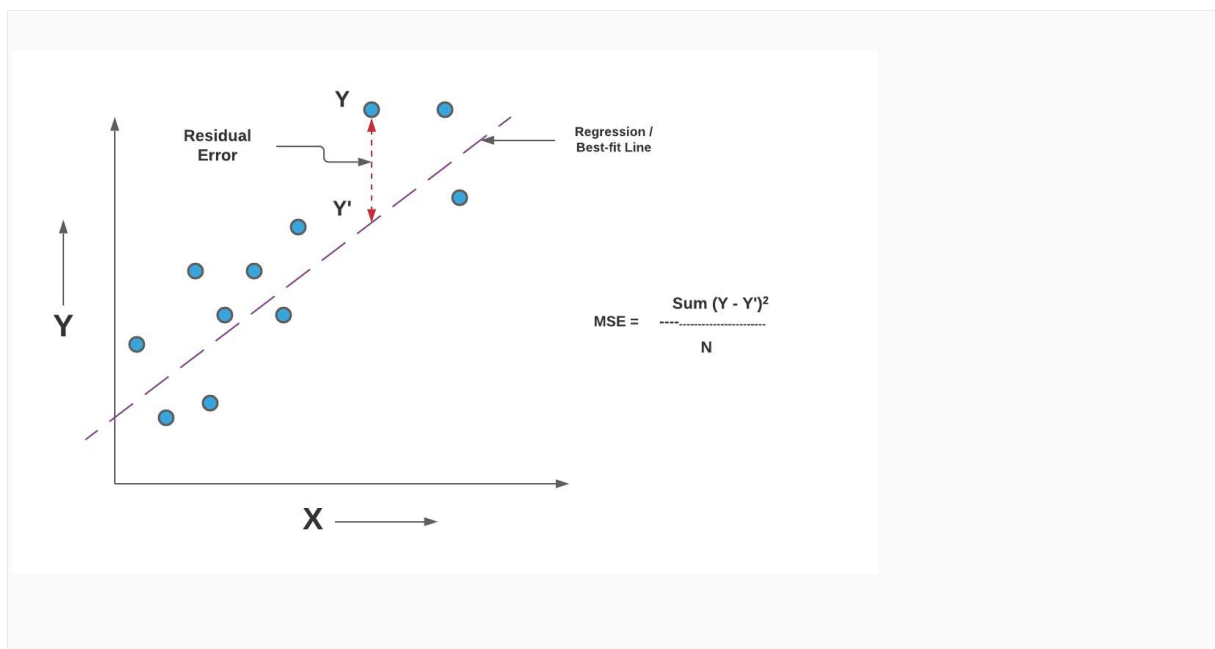
Interpretation

The MSE measures the average of the squared differences between predicted values and actual target values. By squaring the differences, the MSE places a higher weight on larger errors, making it sensitive to outliers. A lower MSE indicates that the model's predictions are closer to the true values, reflecting better overall performance.

Use Cases

MSE has widespread application in various scenarios:

- **Regression Models:** It is extensively used to evaluate the performance of regression models, including linear regression, polynomial regression, and more. The smaller the MSE, the better the model's predictive accuracy.
- **Model Selection:** In cases where multiple models are considered for a specific problem, the one with the lowest MSE is often preferred as it demonstrates better fitting to the data.
- **Feature Selection:** By comparing MSE values while including or excluding certain features, you can identify which features contribute significantly to prediction accuracy.



Mean Squared Error

Limitations



While MSE is a valuable metric, it has certain limitations:

- **Sensitivity to Outliers:** MSE is sensitive to outliers due to the squaring of errors. This may cause extreme values to have a significant impact on the model.
- **Scale Dependence:** The magnitude of the MSE depends on the scale of the target variable. This can make it challenging to compare MSE values across different datasets.

Here is a simple Python code snippet to calculate and display the Mean Square Error using the scikit-learn library:

```
import numpy as np

from sklearn.metrics import mean_squared_error

# Simulated ground truth and predicted values

ground_truth = np.array([2.5, 3.7, 4.2, 5.0, 6.1])

predicted_values = np.array([2.2, 3.5, 4.0, 4.8, 5.8])

# Calculate Mean Square Error

mse = mean_squared_error(ground_truth, predicted_values)

print(f'Mean Square Error: {mse}')
```

Output: Mean Square Error: 0.06

In this code, the `mean_squared_error()` function from scikit-learn has been used to calculate the MSE between the ground truth and predicted values.

Mean Squared Logarithmic Error:

What Is Mean Squared Logarithmic Error (MSLE)?

Mean squared logarithmic error (MSLE) is a less commonly used loss function. It's considered to be an improvement over using percentage based errors for training because its

numerical properties are better, but it essentially serves the same purpose: Trying to create a balance between data points with orders of magnitude difference during model training.

HOW DO YOU CALCULATE MSLE?

In Python, you most probably are going to

use `sklearn.metrics.mean_squared_logarithmic_error`, which works exactly like the MSE counterpart. It's easy to calculate it from scratch using code:

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

MSLE equation. | Image:

Mor Kapronczay

You have to add one to the actual and predicted target values, and take their differences by subtracting the latter from the former. Subsequently, we square those logarithmic differences 1-1, then take the mean.

```
import numpy as np

actual = np.array([1, 2, 3, 4, 5])
predicted = np.array([1.1, 1.9, 2.7, 4.5, 6])

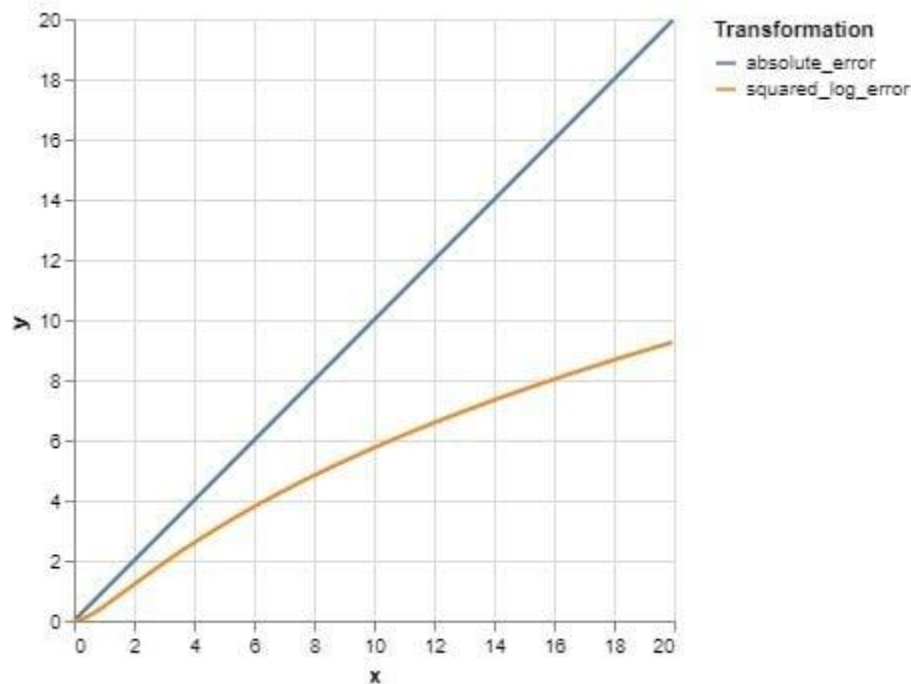
def msle(actual: np.ndarray, predicted: np.ndarray) -> float:
    log_differences = np.subtract(np.log(1 + actual), np.log(1 + predicted))
    squared_log_differences = np.square(log_differences)
    return np.mean(squared_log_differences)

msle(actual, predicted)
```

WHEN SHOULD YOU CONSIDER MSLE?

The logarithm essentially makes the error profile more flat, reducing the impact of the larger values. The plus one evades the logarithm, producing negative values for errors between zero

and one. It's important to note that this metric assumes non-negative target variables. As a result, the previously seen overwhelming effect of large values is reduced, producing a more equal emphasis on data points.



MSLE target variable

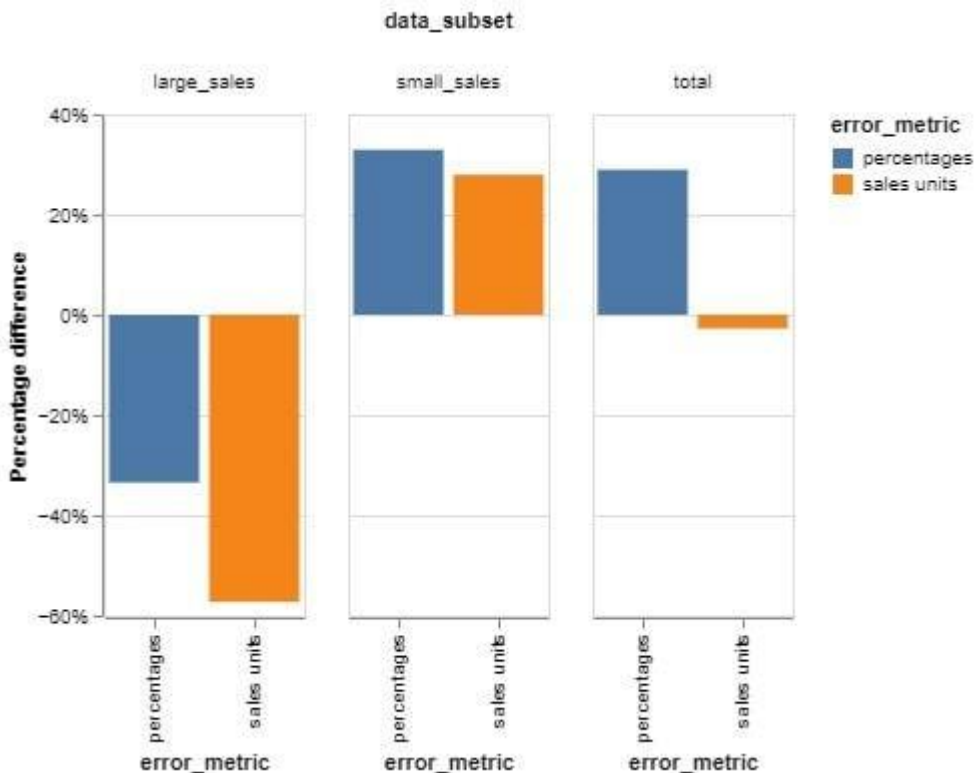
transformation line graph. | Image: Mor Kapronczay

In summary, you should use MSLE if:

- You want to level the playing field for target values that have different orders of magnitudes.
- You aim for a balanced model having roughly similar percentage errors.
- You can tolerate large differences in terms of units for large target values.

Mean Squared Error (MSE) vs. Mean Squared Logarithmic Error (MSLE)

Back to the store example. Let's run a quick simulation predicting the sales of individual products for the next period using sales data from the current (previous) period as a feature. We'll look at two models, one that's trained with mean squared error (MSE) and the other mean squared logarithmic error (MSLE). The bars show the percentage difference between the results of the two models in terms of their respective error bases.



A bar graph

comparison of the error metrics for MSE vs MSLE. | Image: Mor Kapronczay

Looking at the graph, the negative percentage differential shows that the MSE model is more effective in that particular segment, producing smaller errors, while the positive differential indicates that the MSLE model is performing better for that group.

For instance, the first bar group shows large sales places/products. The MSE objective model has a 30 percent better percentage error and is 56 percent better when errors are measured in sales units. On the other hand, when looking at small sales, the MSLE model is better in terms of units and percentages. The same phenomenon can be seen on the third bar group. Percentage-wise, MSLE performs better because of the better percentage errors it produces for small sales, which make up most of the data. Meanwhile, MSLE optimization results in large errors in sales units for large sales, effectively making MSE a slightly better performer in terms of units over the whole group.

So, what should you learn from all of this? In my view, these are the most important takeaways from this chart:

1. MSE trained models perform better on large sales occasions. These are generally fewer but might be more important. In contrast, MSLE performs better for the average, small sales stores.
2. In all data subsets, MSLE models provide an improvement if errors are measured in percentages. The same applies for MSE if the errors are in sales units.
3. Always consider the loss function you want to optimize for in your use case. Don't just go with the default one.

Mean Absolute Percentage Error:

What is Mean Absolute Percent Error (MAPE)

MAPE, or mean absolute percentage error, is a commonly used performance metric for regression defined as the mean of absolute relative errors:


$$MAPE = \frac{1}{N} \sum_{t=1}^N \left| \frac{E_t - A_t}{A_t} \right|$$

where N is the number of estimates (E_t) produced by the regression model and actuals (A_t) from ground truth data that are being compared when determining the performance of the regression model. MAPE is sometimes expressed as a percentage.

Why is MAPE Important?

MAPE's advantage is that it can be expressed as a percentage, making it understandable to a general audience when applied in any domain. By contrast, other metrics that are not expressed in relative terms or as percentages usually require domain expertise and context to understand the significance of their numerical values.

Because of its formulation relative to actual values, however, MAPE has two main disadvantages:

1. It is prone to division-by-zero errors when $A_t = 0$ for any t . For this reason, a small positive, constant term (for example or larger/smaller depending on the typical values of A_t) could be added to the denominator for numerical stability, or zero-valued actuals could be ignored when averaging.
2. It is asymmetric. For the same prediction errors, smaller actual values cause the relative error to become larger. Therefore, estimated and actual values are not interchangeable in the formulation. To address that limitation, an alternative formulation, called "symmetric MAPE." 

$$|A_t|_{\text{OBJ}}|E_t|_{\text{OBJ}}.$$

Explained Variance Score:

Explained Variance

The explained variance is used to measure the proportion of the variability of the predictions of a machine learning model. Simply put, it is the difference between the expected value and the predicted value. It is a very important concept to understand how much information we can lose by reconciling the dataset.

Hope you now understand the concept of explained variance in machine learning. Always remember to use it when working on regression-based problems to measure the difference between samples and predictions. In the section below, I'll walk you through its implementation using Python.

Explained Variance using Python

To calculate the explained variance of a machine learning model, I will first train a machine learning model using the **linear regression** algorithm and then calculate it using the Python programming language:

```
1 import numpy as np
2 import pandas as pd
3 from sklearn.linear_model import LinearRegression
4 from sklearn.utils import shuffle
5
6 data = pd.read_csv("https://raw.githubusercontent.com/amankharwal/Website-data/master/student-mat.csv")
7
8 data = data[["G1", "G2", "G3", "studytime", "failures", "absences"]]
9
10 predict = "G3"
11 x = np.array(data.drop([predict], 1))
12 y = np.array(data[predict])
13
14 from sklearn.model_selection import train_test_split
15 xtrain, xtest, ytrain, ytest = train_test_split(x, y, test_size=0.2)
16
17 linear_regression = LinearRegression()
18 linear_regression.fit(xtrain, ytrain)
19 predictions = linear_regression.predict(xtest)
20
21 # Calculation of Explained Variance
22 from sklearn.model_selection import cross_val_score
23 print(cross_val_score(linear_regression, x, y, cv=10, scoring="explained_variance").mean())
```



0.8274789442218667

A machine learning model must have at least 60 per cent of explained variance.

D2 Score Visual Evaluation of Regression Models:

Machine learning is an effective tool for predicting numerical values, and regression is one of its key applications. In the arena of regression analysis, accurate estimation is crucial for measuring the overall performance of predictive models. This is where the famous machine learning library Python Scikit-Learn comes in. Scikit-Learn gives a complete set of regression metrics to evaluate the quality of regression models.

In this article, we are able to explore the basics of regression metrics in scikit-learn, discuss the steps needed to use them effectively, provide some examples, and show the desired output for each metric.

Regression

[Regression](#) fashions are algorithms used to expect continuous numerical values primarily based on entering features. In scikit-learn, we will use numerous regression algorithms, such as Linear Regression, Decision Trees, Random Forests, and Support Vector Machines (SVM), amongst others.

Before learning about precise metrics, let's familiarize ourselves with a few essential concepts related to regression metrics:

1. True Values and Predicted Values:

In regression, we've got two units of values to compare: the actual target values (authentic values) and the values expected by our version (anticipated values). The performance of the model is assessed by means of measuring the similarity among these sets.

2. Evaluation Metrics:

Regression metrics are quantitative measures used to evaluate the nice of a regression model. Scikit-analyze provides several metrics, each with its own strengths and boundaries, to assess how well a model suits the statistics.

Types of Regression Metrics

Some common regression metrics in scikit-learn with examples

- Mean Absolute Error (MAE)
- Mean Squared Error (MSE)
- R-squared (R^2) Score
- Root Mean Squared Error (RMSE)

Mean Absolute Error (MAE)

In the fields of statistics and machine learning, the [Mean Absolute Error \(MAE\)](#) is a frequently employed metric. It's a measurement of the typical absolute discrepancies between a dataset's actual values and projected values.

Mathematical Formula

The formula to calculate MAE for a data with "n" data points is:

$$MAE = \frac{1}{n} \sum_{i=1}^n |x_i - y_i|$$

Where:

- x_i represents the actual or observed values for the i-th data point.
- y_i represents the predicted value for the i-th data point.

Python

```
from sklearn.metrics import mean_absolute_error

true_values = [2.5, 3.7, 1.8, 4.0, 5.2]
predicted_values = [2.1, 3.9, 1.7, 3.8, 5.0]

mae = mean_absolute_error(true_values, predicted_values)
print("Mean Absolute Error:", mae)
```

Output:

```
Mean Absolute Error: 0.22000000000000003
```

Mean Squared Error (MSE)

A popular metric in statistics and machine learning is the [Mean Squared Error](#) (MSE). It measures the square root of the average discrepancies between a dataset's actual values and projected values. MSE is frequently utilized in regression issues and is used to assess how well predictive models work.

Mathematical Formula

For a dataset containing 'n' data points, the MSE calculation formula is:

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

where:

- x_i represents the actual or observed value for the i-th data point.
- y_i represents the predicted value for the i-th data point.

Example:

Python

```
from sklearn.metrics import mean_squared_error

true_values = [2.5, 3.7, 1.8, 4.0, 5.2]
predicted_values = [2.1, 3.9, 1.7, 3.8, 5.0]

mse = mean_squared_error(true_values, predicted_values)
print("Mean Squared Error:", mse)
```

Output:

Mean Squared Error: 0.05799999999999996

R-squared (R²) Score

A statistical metric frequently used to assess the goodness of fit of a regression model is the **R-squared (R²)** score, also referred to as the coefficient of determination. It quantifies the percentage of the dependent variable's variation that the model's independent variables contribute to. R² is a useful statistic for evaluating the overall effectiveness and explanatory power of a regression model.

Mathematical Formula

The formula to calculate the R-squared score is as follows:

$$R^2 = 1 - \frac{SSR}{SST}$$

Where:

- R² is the R-Squared.
- SSR represents the sum of squared residuals between the predicted values and actual values.
- SST represents the total sum of squares, which measures the total variance in the dependent variable.

Python

```
from sklearn.metrics import r2_score

true_values = [2.5, 3.7, 1.8, 4.0, 5.2]
predicted_values = [2.1, 3.9, 1.7, 3.8, 5.0]

r2 = r2_score(true_values, predicted_values)
print("R-squared (R2) Score:", r2)
```

Output:

R-squared (R²) Score: 0.9588769143505389

Root Mean Squared Error (RMSE)

RMSE stands for [Root Mean Squared Error](#). It is a usually used metric in regression analysis and machine learning to measure the accuracy or goodness of fit of a predictive model, especially when the predictions are continuous numerical values.

The RMSE quantifies how well the predicted values from a model align with the actual observed values in the dataset. Here's how it works:

1. **Calculate the Squared Differences:** For each data point, subtract the predicted value from the actual (observed) value, square the result, and sum up these squared differences.
2. **Compute the Mean:** Divide the sum of squared differences by the number of data points to get the mean squared error (MSE).
3. **Take the Square Root:** To obtain the RMSE, simply take the square root of the MSE.

Mathematical Formula

The formula for RMSE for a data with 'n' data points is as follows:

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

Where:

- RMSE is the Root Mean Squared Error.
- x_i represents the actual or observed value for the i-th data point.
- y_i represents the predicted value for the i-th data point.

Python

```
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
import numpy as np

# Sample data
true_prices = np.array([250000, 300000, 200000, 400000, 350000])
predicted_prices = np.array([240000, 310000, 210000, 380000, 340000])

# Calculate RMSE
rmse = np.sqrt(mean_squared_error(true_prices, predicted_prices))

print("Root Mean Squared Error (RMSE):", rmse)
```

Output:

```
Root Mean Squared Error (RMSE): 12649.110640673518
```

NOTE:

When using regression metrics in scikit-learn, we generally aim to obtain a single numerical value for each metric.

Using Regression Metrics on California House Prices Dataset

Here are the steps for applying regression metrics to our model, and for a better understanding, we've illustrated them using the example of predicting house prices.

Import Libraries and Load the Dataset

Python

```
#importing Libraries
import pandas as pd
import numpy as np
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
```

We import necessary libraries and load the dataset from our own source or from scikit-learn library.

Loading the Dataset

Python3

```
# Load the California Housing Prices dataset
data = fetch_california_housing()
df = pd.DataFrame(data.data, columns=data.feature_names)
df['target'] = data.target
```

The code loads the dataset for California Housing Prices using the scikit-learn `fetch_california_housing` function, builds a DataFrame (df) containing the dataset's characteristics and the target variable, and then adds the target variable to the DataFrame.

Data Splitting and Train-Test Split

Python

```
# Split the data into features (X) and target variable (y)
X = df.drop(columns=['target'])
y = df['target']

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

The code divides the dataset into features (X) and the target variable (y) by removing the 'target' column from the DataFrame and allocating it to X while assigning the 'target' column to y. With a fixed random seed (`random_state=42`) for repeatability, it then further divides the data into training and testing sets, utilizing 80% of the data for training (X_train and y_train) and 20% for testing (X_test and y_test).

Create and Train the Regression Model

Create and Train the Regression Model

Python

```
# Create and train the Linear Regression model
model = LinearRegression()
model.fit(X_train, y_train)
```

This code builds a linear regression model (model) and trains it using training data (X_train and y_train) to discover a linear relationship between the characteristics and the target variable.

Make Predictions

Python

```
# Make predictions on the test set
y_pred = model.predict(X_test)
```

The code estimates the values of the target variable based on the discovered relationships between features and the target variable, using the trained Linear Regression model (model) to make predictions (y_pred) on the test set (X_test).

Calculate Evaluation Metrics

```
# Calculate evaluation metrics
mae = mean_absolute_error(y_test, y_pred)
mse = mean_squared_error(y_test, y_pred)
r_squared = r2_score(y_test, y_pred)
rmse = np.sqrt(mse)

# Print the evaluation metrics
print("Mean Absolute Error (MAE):", mae)
print("Mean Squared Error (MSE):", mse)
print("R-squared (R²):", r_squared)
print("Root Mean Squared Error (RMSE):", rmse)
```

Output:

```
Mean Absolute Error (MAE): 0.5332001304956553
Mean Squared Error (MSE): 0.5558915986952444
R-squared (R²): 0.5757877060324508
Root Mean Squared Error (RMSE): 0.7455813830127764
```




The code computes four regression assessment metrics, including Mean Absolute Error (MAE), Mean Squared Error (MSE), R-squared (R²), and Root Mean Squared Error (RMSE), based on the predicted values (y_{pred}) and the actual values from the test set (y_{test}). The model's success in foretelling the values of the target variable is then evaluated by printing these metrics, which shed light on the model's precision and goodness of fit.

Understanding the output:

1. Mean Absolute Error (MAE): 0.5332

- An MAE of 0.5332 means that, on average, the model's predictions are approximately \$0.5332 away from the true house prices.

2. Mean Squared Error (MSE): 0.5559

- An MSE of 0.5559 means that, on average, the squared prediction errors are approximately 0.5559.

3. R-squared (R²): 0.5758

- An R² of 0.5758 indicates that the model can explain approximately 57.58% of the variance in house prices.

4. Root Mean Squared Error (RMSE): 0.7456

- An RMSE of 0.7456 indicates that, on average, the model's predictions have an error of approximately \$0.7456 in the same units as the house prices.

Conclusion

In conclusion, understanding regression metrics in scikit-learn is important for all people running with predictive models. These metrics allow us to evaluate the quality of our regression models, helping us make wise decisions about overall performance evaluation. In this article, we have seen the logic behind regression metrics, the steps required to evaluate a regression model, and provided examples. Whether we're predicting house prices, stock market trends, or any other continuous numerical values, the process remains same.