Tugas Data Science Week 11

Nama : Mochammad Qussay Alhindi Achmadi
NIM : 1103213087

A. Perhitungan error pada regresi

1. SMAPE - Symmetric Mean Absolute Percentage Error

$$\text{SMAPE}(y, \hat{y}) = \frac{100\%}{N} \sum_{i=0}^{N-1} \frac{2 * |y_i - \hat{y}_i|}{|y| + |\hat{y}|}$$

which is an accuracy measure commonly used in forecasting and time series analysis.

Given the actual values y and the predicted values y_hat, the SMAPE is calculated as the average of the absolute percentage errors between the two, where each error is weighted by the sum of the absolute values of the actual and predicted values.

The resulting score ranges between 0 and 1, where a score of 0 indicates a perfect match between the actual and predicted values, and a score of 1 indicates no match at all. A smaller value of SMAPE is better, and it is often multiplied by 100% to obtain the percentage error. Best possible score is 0.0, smaller value is better. Range = [0, 1].

```
from numpy import array
from permetrics.regression import RegressionMetric

## For 1-D array
y_true = array([3, -0.5, 2, 7])
y_pred = array([2.5, 0.0, 2, 8])

evaluator = RegressionMetric(y_true, y_pred)
print(evaluator.symmetric_mean_absolute_percentage_error())

## For > 1-D array
y_true = array([[0.5, 1], [-1, 1], [7, -6]])
y_pred = array([[0, 2], [-1, 2], [8, -5]])

evaluator = RegressionMetric(y_true, y_pred)
print(evaluator.SMAPE(multi_output="raw_values"))
```

2. NSE - Nash-Sutcliffe Efficiency

$$\text{NSE}(y, \hat{y}) = 1 - \frac{\sum_{i=0}^{N-1} (y_i - \hat{y}_i)^2}{\sum_{i=0}^{N-1} (y_i - mean(y))^2}$$

is calculated as the ratio of the mean squared error between the observed and simulated streamflow to the variance of the observed streamflow. The NSE ranges between -inf and 1, with a value of 1 indicating perfect agreement between the observed and simulated streamflow

```
  from numpy import array
  from permetrics.regression import RegressionMetric

  ## For 1-D array
  y_true = array([3, -0.5, 2, 7])
  y_pred = array([2.5, 0.0, 2, 8])

evaluator = RegressionMetric(y_true, y_pred)
print(evaluator.nash_sutcliffe_efficiency())

  ## For > 1-D array
  y_true = array([[0.5, 1], [-1, 1], [7, -6]])
  y_pred = array([[0, 2], [-1, 2], [8, -5]])

evaluator = RegressionMetric(y_true, y_pred)
  print(evaluator.NSE(multi_output="raw_values"))
```

3. MASE - Mean Absolute Scaled Error

$$\mathrm{MASE}(y, \hat{y}) = \frac{\frac{1}{N}\sum_{i=0} N - 1\, |y_i - \hat{y}_i|}{\frac{1}{N-1}\sum_{i=1}^{N-1} |\, y_i - y_{i-1}|}$$

- Best possible score is 0.0, smaller value is better. Range = [0, +inf)
- m = 1 for non-seasonal data, m > 1 for seasonal data

```
  from numpy import array
  from permetrics.regression import RegressionMetric

  ## For 1-D array
  y_true = array([3, -0.5, 2, 7])
  y_pred = array([2.5, 0.0, 2, 8])

evaluator = RegressionMetric(y_true, y_pred)
print(evaluator.mean_absolute_scaled_error())

  ## For > 1-D array
  y_true = array([[0.5, 1], [-1, 1], [7, -6]])
  y_pred = array([[0, 2], [-1, 2], [8, -5]])

evaluator = RegressionMetric(y_true, y_pred)
  print(evaluator.MASE(multi_output="raw_values"))
```

4. MSLE - Mean Squared Logarithmic Error

$$\mathrm{MSLE}(y, \hat{y}) = \frac{1}{N}\sum_{i=0}^{N-1} (\log_e (1 + y_i) - \log_e (1 + \hat{y}_i))^2$$

Where $\log_e(x)$ means the natural logarithm of x. This metric is best to use when targets having exponential growth, such as population counts, average sales of a commodity over a span of years etc. Note that this metric penalizes an under-predicted estimate greater than an over-predicted estimate.

The Mean Squared Logarithmic Error (MSLE) is a statistical measure used to evaluate the accuracy of a forecasting model, particularly when the data has a wide range of values. It measures the average of the squared differences between the logarithms of the predicted and actual values.

The logarithmic transformation used in the MSLE reduces the impact of large differences between the actual and predicted values and provides a better measure of the relative errors between the two values. The MSLE is always a positive value, with a smaller MSLE indicating better forecast accuracy.

The MSLE is commonly used in applications such as demand forecasting, stock price prediction, and sales forecasting, where the data has a wide range of

values and the relative errors are more important than the absolute errors. + It is important to note that the MSLE is not suitable for data with negative values or zero values, as the logarithm function is not defined for these values. + Best possible score is 0.0, smaller value is better. Range = [0, +inf)

```
 from numpy import array
 from permetrics.regression import RegressionMetric

 ## For 1-D array
 y_true = array([3, -0.5, 2, 7])
 y_pred = array([2.5, 0.0, 2, 8])

evaluator = RegressionMetric(y_true, y_pred)
print(evaluator.mean_squared_log_error())

 ## For > 1-D array
 y_true = array([[0.5, 1], [-1, 1], [7, -6], [1, 2]])
 y_pred = array([[0, 2], [-1, 2], [8, -5], [1.1, 1.9]])

evaluator = RegressionMetric(y_true, y_pred)
 print(evaluator.MSLE(multi_output="raw_values"))
```

5. **SE - Squared Error**

$$\text{SE}(y, f_i) = \frac{1}{n}\sum_{i=1}^{n} (y_i - f_i)^2$$

- Best possible score is 0.0, smaller value is better. Range = [0, +inf)

- Note: Computes the squared error between two numbers, or for element between a pair of list, tuple or numpy arrays.

- The Squared Error (SE) is a metric used to evaluate the accuracy of a regression model by measuring the average of the squared differences between the

```
  from numpy import array
  from permetrics.regression import RegressionMetric

  ## For 1-D array
  y_true = array([3, -0.5, 2, 7])
  y_pred = array([2.5, 0.0, 2, 8])

evaluator = RegressionMetric(y_true, y_pred)
print(evaluator.single_squared_error())

  ## For > 1-D array
  y_true = array([[0.5, 1], [-1, 1], [7, -6]])
  y_pred = array([[0, 2], [-1, 2], [8, -5]])

evaluator = RegressionMetric(y_true, y_pred)
  print(evaluator.SE())
```

B. Perhitungan Performansi pada Klasifikasi

1. F-Beta Score (FBS)

The F-beta score is the weighted harmonic mean of precision and recall, reaching its optimal value at 1 and its worst value at 0.

The beta parameter represents the ratio of recall importance to precision importance. beta > 1 gives more weight to recall, while beta < 1 favors precision. For example, beta = 2 makes recall twice as important as precision, while beta = 0.5 does the opposite. Asymptotically, beta -> +inf considers only recall, and beta -> 0 only precision.

$$F_\beta = \frac{(1+\beta^2)\mathrm{tp}}{(1+\beta^2)\mathrm{tp} + \mathrm{fp} + \beta^2\mathrm{fn}}$$

```
  from numpy import array
  from permetrics.classification import ClassificationMetric

  ## For integer labels or categorical labels
  y_true = [0, 1, 0, 0, 1, 0]
  y_pred = [0, 1, 0, 0, 0, 1]

  # y_true = ["cat", "ant", "cat", "cat", "ant", "bird", "bird", "bird"]
  # y_pred = ["ant", "ant", "cat", "cat", "ant", "cat", "bird", "ant"]

cm = ClassificationMetric(y_true, y_pred)

print(cm.fbeta_score(average=None))
print(cm.fbeta_score(average="micro"))
print(cm.FBS(average="macro"))
print(cm.FBS(average="weighted"))
```

2. **G-Mean Score (GMS)**

G-mean is a performance metric in the field of machine learning and specifically in binary classification problems. It is a balanced version of the geometric mean, which is calculated as the square root of the product of true positive rate (TPR) and true negative rate (TNR) also known as sensitivity and specificity, respectively.

The G-mean is a commonly used metric to evaluate the performance of a classifier in imbalanced datasets where one class has a much higher number of samples than the other. It provides a balanced view of the model's performance as it penalizes low values of TPR and TNR in a single score. The G-mean score provides a balanced evaluation of a classifier's performance by considering both the positive and negative classes.

- The formula for the G-mean score is given by

$$G - mean = sqrt(TPR * TNR)$$
$$Gmean = \sqrt{TPR * TNR}$$

- where TPR (True Positive Rate) is defined as

$$TPR = \frac{TP}{TP + FN}$$

- and TNR (True Negative Rate) is defined as

$$TNR = \frac{TN}{TN + FP}$$

with TP (True Positives) as the number of instances that are correctly classified as positive, TN (True Negatives) as the number of instances that are correctly classified as negative, FP (False Positives) as the number of instances that are wrongly classified as positive, and FN (False Negatives) as the number of instances that are wrongly classified as negative.

- Best possible score is 1.0, higher value is better. Range = [0, 1]

```
from numpy import array
from permetrics.classification import ClassificationMetric

## For integer labels or categorical labels
y_true = [0, 1, 0, 0, 1, 0]
y_pred = [0, 1, 0, 0, 0, 1]

# y_true = ["cat", "ant", "cat", "cat", "ant", "bird", "bird", "bird"]
# y_pred = ["ant", "ant", "cat", "cat", "ant", "cat", "bird", "ant"]

cm = ClassificationMetric(y_true, y_pred)
```

```
print(cm.g_mean_score(average=None))
print(cm.GMS(average="micro"))
print(cm.GMS(average="macro"))
  print(cm.GMS(average="weighted"))
```

3. Negative Predictive Value (NPV)

$$NPV = \frac{tn}{tn + fn}$$

The negative predictive value is defined as the number of true negatives (people who test negative who don't have a condition) divided by the total number of people who test negative.

The negative predictive value is the ratio tn / (tn + fn) where tn is the number of true negatives and fn the number of false negatives.

In the multi-class and multi-label case, this is the average of the NPV score of each class with weighting depending on the average parameter.

- Best possible score is 1.0, higher value is better. Range = [0, 1]

```
from numpy import array
from permetrics.classification import ClassificationMetric

## For integer labels or categorical labels
y_true = [0, 1, 0, 0, 1, 0]
y_pred = [0, 1, 0, 0, 0, 1]

# y_true = ["cat", "ant", "cat", "cat", "ant", "bird", "bird", "bird"]
# y_pred = ["ant", "ant", "cat", "cat", "ant", "cat", "bird", "ant"]

cm = ClassificationMetric(y_true, y_pred)

print(cm.NPV(average=None))
print(cm.NPV(average="micro"))
print(cm.NPV(average="macro"))
print(cm.NPV(average="weighted"))
```

4. Specificity Score (SS)

$$TNR = \frac{tn}{tn + fp}$$

The specificity score is the ratio tn / (tn + fp) where tn is the number of false positives and fp the number of false positives. It measures how many observations out of all negative observations have we classified as negative. In fraud detection example, it tells us how many transactions, out of all non-fraudulent transactions, we marked as clean.

In the multi-class and multi-label case, this is the average of the SS score of each class with weighting depending on the average parameter.

- Best possible score is 1.0, higher value is better. Range = [0, 1]

```
from numpy import array
from permetrics.classification import ClassificationMetric

## For integer labels or categorical labels
y_true = [0, 1, 0, 0, 1, 0]
y_pred = [0, 1, 0, 0, 0, 1]

# y_true = ["cat", "ant", "cat", "cat", "ant", "bird", "bird", "bird"]
# y_pred = ["ant", "ant", "cat", "cat", "ant", "cat", "bird", "ant"]

cm = ClassificationMetric(y_true, y_pred)

print(cm.specificity_score(average=None))
print(cm.specificity_score(average="micro"))
print(cm.specificity_score(average="macro"))
print(cm.specificity_score(average="weighted"))
```

5. **Matthews Correlation Coefficient (MCC)**

$$MCC = \frac{tp * tn - fp * fn}{(tp + fp)(tp + fn)(tn + fp)(tn + fn)}$$

In the multi-class and multi-label case, this is the average of the MCC score of each class with weighting depending on the average parameter.

- Best possible score is 1.0, higher value is better. Range = [-1, +1]

```
from numpy import array
from permetrics.classification import ClassificationMetric

## For integer labels or categorical labels
y_true = [0, 1, 0, 0, 1, 0]
y_pred = [0, 1, 0, 0, 0, 1]

# y_true = ["cat", "ant", "cat", "cat", "ant", "bird", "bird", "bird"]
# y_pred = ["ant", "ant", "cat", "cat", "ant", "cat", "bird", "ant"]

cm = ClassificationMetric(y_true, y_pred)

print(cm.MCC(average=None))
print(cm.MCC(average="micro"))
print(cm.MCC(average="macro"))
print(cm.MCC(average="weighted"))
```

C. Perhitungan Performansi pada Clustering

1. Duda Hart Index (DHI)

The Duda index, also known as the D-index or Duda-Hart index, is a clustering evaluation metric that measures the compactness and separation of clusters. It was proposed by Richard O. Duda and Peter E. Hart in their book “Pattern Classification and Scene Analysis.”

The Duda index is defined as the ratio between the average pairwise distance within clusters and the average pairwise distance between clusters. A lower value of the Duda index indicates better clustering, indicating that the clusters are more compact and well-separated. Here’s the formula to calculate the Duda index:

```
Duda Index = (Average pairwise intra-cluster distance) / (Average pairwise inter-
cluster distance)
```

To calculate the Duda index, you need the following steps:

```
Compute the average pairwise distance within each cluster (intra-cluster
distance).
Compute the average pairwise distance between different clusters (inter-cluster
distance).
Divide the average intra-cluster distance by the average inter-cluster distance
to obtain the Duda index.
```

The Duda index is a useful metric for evaluating clustering results, particularly when the compactness and separation of clusters are important. However, it’s worth noting that the Duda index assumes Euclidean distance and may not work well with all types of data or distance metrics.

```python
import numpy as np
from permetrics import ClusteringMetric

## For integer labels or categorical labels
data = np.array([[1, 2], [3, 4], [5, 6], [7, 8], [9, 10]])
y_pred = np.array([0, 0, 1, 1, 1])

cm = ClusteringMetric(X=data, y_pred=y_pred)

print(cm.duda_hart_index())
print(cm.DHI())
```

2. Sum of Squared Error Index (SSEI)

Sum of Squared Error (SSE) is a commonly used metric to evaluate the quality of clustering in unsupervised learning problems. SSE measures the sum of squared distances between each data point and its corresponding centroid or cluster center. It quantifies the compactness of the clusters.

Here’s how you can calculate the SSE in a clustering problem:

```
1) Assign each data point to its nearest centroid or cluster center based on
some distance metric (e.g., Euclidean distance).
```

```
2) For each data point, calculate the squared Euclidean distance between the
data point and its assigned centroid.
3) Sum up the squared distances for all data points to obtain the SSE.
```

Higher SSE values indicate higher dispersion or greater variance within the clusters, while lower SSE values indicate more compact and well-separated clusters. Therefore, minimizing the SSE is often a goal in clustering algorithms.

```
import numpy as np
from permetrics import ClusteringMetric

## For integer labels or categorical labels
data = np.array([[1, 2], [3, 4], [5, 6], [7, 8], [9, 10]])
y_pred = np.array([0, 0, 1, 1, 1])

cm = ClusteringMetric(X=data, y_pred=y_pred)

print(cm.sum_squared_error_index())
print(cm.SSEI())
```

3. **Purity Score (PuS)**

Purity is a metric used to evaluate the quality of clustering results, particularly in situations where the ground truth labels of the data points are known. It measures the extent to which the clusters produced by a clustering algorithm match the true class labels of the data. Here’s how Purity is calculated:

```
1) For each cluster, find the majority class label among the data points in that
cluster.
2) Sum up the sizes of the clusters that belong to the majority class label.
3) Divide the sum by the total number of data points.
```

The resulting value is the Purity score, which ranges from 0 to 1. A Purity score of 1 indicates a perfect clustering, where each cluster contains only data points from a single class.

Purity is a simple and intuitive metric but has some limitations. It does not consider the actual structure or distribution of the data within the clusters and is sensitive to the number of clusters and class imbalance. Therefore, it may not be suitable for evaluating clustering algorithms in all scenarios.

```
import numpy as np
from permetrics import ClusteringMetric

## For integer labels or categorical labels
y_true = np.array([0, 0, 1, 1, 1, 2, 2, 1])
y_pred = np.array([0, 0, 1, 1, 2, 2, 2, 2])

cm = ClusteringMetric(y_true=y_true, y_pred=y_pred)

print(cm.purity_score())
print(cm.PuS())
```

4. **Tau Score (TS)**
   The Tau index, also known as the Tau coefficient, is a measure of agreement or similarity between two clustering solutions. It is commonly used to compare the similarity of two different clusterings or to evaluate the stability of a clustering algorithm.

   The Tau index is based on the concept of concordance, which measures the extent to which pairs of objects are assigned to the same clusters in two different clustering solutions. The index ranges from -1 to 1, where 1 indicates perfect agreement, 0 indicates random agreement, and -1 indicates perfect disagreement or inversion of the clustering solutions.

   The calculation of the Tau index involves constructing a contingency table that counts the number of pairs of objects that are concordant (i.e., assigned to the same cluster in both solutions) and discordant (i.e., assigned to different clusters in the two solutions).

   The formula for calculating the Tau index is as follows:

```
        Tau = (concordant_pairs - discordant_pairs) / (concordant_pairs +
discordant_pairs)
```

   A higher value of the Tau index indicates greater similarity or agreement between the two clusterings, while a lower value indicates less agreement. It's important to note that the interpretation of the Tau index depends on the specific clustering algorithm and the data being clustered.

```
import numpy as np
from permetrics import ClusteringMetric

## For integer labels or categorical labels
y_true = np.array([0, 0, 1, 1, 1, 2, 2, 1])
y_pred = np.array([0, 0, 1, 1, 2, 2, 2, 2])

cm = ClusteringMetric(y_true=y_true, y_pred=y_pred)

print(cm.tau_score())
print(cm.TS())
```

5. **Ball Hall Index**
   The Ball Hall Index is a clustering validity index that measures the compactness and separation of clusters in a clustering result. It provides a quantitative measure of how well-separated and tight the clusters are.

   The formula for calculating the Ball Hall Index is as follows:

```
BHI = Xichma(1 / (2 * n_i) * Xichma(d(x, c_i)) / n
```

Where:

n is the total number of data points n_i is the number of data points in cluster i d(x, c_i) is the Euclidean distance between a data point x and the centroid c_i of cluster i

The Ball Hall Index computes the average distance between each data point and its cluster centroid and then averages this across all clusters. The index is inversely proportional to the compactness and separation of the clusters. A smaller BHI value indicates better-defined and well-separated clusters.

A lower BHI value indicates better clustering, as it signifies that the data points are closer to their own cluster centroid than to the centroids of other clusters, indicating a clear separation between clusters.

```python
import numpy as np
from permetrics import ClusteringMetric

## For integer labels or categorical labels
data = np.array([[1, 2], [3, 4], [5, 6], [7, 8], [9, 10]])
y_pred = np.array([0, 0, 1, 1, 1])

cm = ClusteringMetric(X=data, y_pred=y_pred)

print(cm.ball_hall_index())
print(cm.BHI())
```