

Model Evaluation Setup

Table of Contents

Introduction	2
Why Do We Evaluate Models?	3
Model Evaluation in Classification	4
Model Evaluation in Regression.....	7
Model Evaluation in Forecasting	9
Choosing the Right Regression/Forecasting Metric.....	9
Model Evaluation in Clustering.....	10
Choosing the Right Evaluation Metric.....	11
Evaluating the Stability of Clustering Results.....	11
Setup Model Evaluation Mechanism for a Model.....	11
Dataset Splitting	12
Set Up Model Evaluation in DataBricks (Code based)	15
Monitor Evaluation in Databricks - MLflow	18
Set Up Model Evaluation in Azure ML	20
General Model in Azure Machine Learning & in ADB - MLFlow.....	24
APPENDIX A – Evaluation Metrics	25
Forecast Evaluation Metrics	25
Classification Machine Learning Metrics.....	30
Clustering Machine Learning Metrics	33

Introduction

Model evaluation setup is a critical component in the development and deployment of machine learning models, playing a pivotal role in assessing their performance, generalization capabilities, and suitability for real-world applications. The process involves two key stages: inference and evaluation. Inference is a phase when trained model is queried and it infers the output for that query, and evaluation refers to measuring the outputs with defined metrics. In the inference phase, the trained model is deployed to a production environment, ensuring scalability, seamless data input handling, and robust monitoring and logging mechanisms. Concurrently, the evaluation phase involves a comprehensive analysis of the model's performance using carefully chosen metrics, such as accuracy, precision, recall, and various others, to gauge its effectiveness on both training and unseen data. Leveraging techniques like cross-validation, confusion matrices, and visualization tools, the evaluation setup provides insights into the model's behavior, identifies areas for improvement, and addresses considerations like bias, fairness, and interpretability. Iterative improvement based on evaluation results is paramount, creating a dynamic feedback loop that refines the model and enhances its reliability and relevance over time.

Evaluation surrounds with the following topics:

1. **Metrics:** These are evaluation functions based on the nature of your machine learning task (classification, regression, clustering, etc.). Common metrics include accuracy, precision, recall, F1 score, Mean Absolute Error (MAE), Mean Squared Error (MSE), etc.
2. **Train-Test Split:** Split the dataset into training and testing sets to evaluate the model's performance on unseen data. This is crucial for assessing how well the model generalizes.
3. **Cross-Validation:** Use cross-validation techniques, such as k-fold cross-validation, to robustly evaluate the model's performance across multiple subsets of the data.
4. **Confusion Matrix:** For classification tasks, create a confusion matrix to understand the distribution of true positives, true negatives, false positives, and false negatives.
5. **ROC and Precision-Recall Curves:** Receiver Operating Characteristic (ROC) curves and Precision-Recall curves visualize the trade-offs between true positive rate and false positive rate.
6. **Model Explainability:** Model explainability techniques allow to understand how the model makes decisions. This is important for interpreting and trusting the model's outputs.
7. **Bias and Fairness:** Bias and fairness of the model allows to check if the model is making decisions that could have social or ethical implications.
8. **Business Impact:** Evaluate the business impact of the model's predictions. This involves understanding how well the model aligns with the overall goals and objectives of the project.
9. **Iterative Improvement:** Use the insights gained from evaluation to iteratively improve the model. This may involve hyperparameter tuning, feature engineering, or collecting additional data.

Why Do We Evaluate Models?

Model evaluation is an integral part of the ML lifecycle. It enables data scientists to measure, interpret, and explain the performance of their models. It accelerates the model development timeframe by providing insights into how and why models are performing the way that they are performing. Especially as the complexity of ML models increases, being able to swiftly observe and understand the performance of ML models is essential in a successful ML development journey.

As, during the automation process, model runs are scheduled for many models. For such models, evaluation should be an ongoing process as measuring responses of the newly trained model is very necessary. If the responses are not proper or not as expected then we can evaluate and improve before deploying the model again to Production or for already deployed model, we can restore the older version from the backup.

Model evaluation metrics are split into two categories majorly based on the measurement of similarity and dissimilarity.

Metrics that can be interpreted as measuring similarity

1. Accuracy: Measures the overall correctness of predictions.
2. Precision: Measures the accuracy of positive predictions.
3. Recall (Sensitivity): Measures the ability of the model to capture all positive instances.
4. F1 Score: The harmonic mean of precision and recall, providing a balance between the two.
5. Jaccard Similarity Coefficient: Measures the similarity between two sets by comparing their intersection to the union.

Metrics that can be interpreted as measuring dissimilarity

1. Mean Squared Error (MSE): Measures the average squared difference between predicted and actual values in regression problems.
2. Log Loss (Cross-Entropy Loss): Commonly used for classification problems, it measures the difference between predicted probabilities and true outcomes.
3. Hamming Loss: Measures the fraction of labels that are incorrectly predicted.
4. Area Under the ROC Curve (AUC-ROC): Measures the ability of a model to distinguish between classes, and lower AUC can indicate dissimilarity in performance.

Model Evaluation in Classification

Classification models assign labels to input data points. Evaluating these models helps assess their ability to correctly classify instances. Here are some common evaluation metrics for classification.

1) Accuracy

Accuracy measures the ratio of correctly predicted instances to the total instances in the dataset.

- **Interpretation:** It provides a general measure of the model's correctness.

Use When

- Classes in the dataset are balanced.
- Equal importance is assigned to false positives and false negatives.

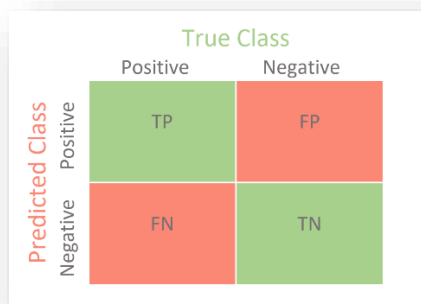
2) Confusion Matrix

A confusion matrix is a table that is often used to evaluate the performance of a classification model. It summarizes the predicted and actual classifications in a tabular form.

- **Interpretation:** The confusion matrix provides a detailed breakdown of the model's performance, showing the counts of true positives, true negatives, false positives, and false negatives.

Use When

- **Imbalanced Classes:** When the classes in the dataset are imbalanced, meaning some classes have significantly more instances than others.
- **Importance of Errors:** When the cost of false positives and false negatives is different, and you want to understand the impact of each type of error.



		True Class	
		Positive	Negative
Predicted Class	Positive	TP	FP
	Negative	FN	TN

3) Precision

Precision is the ratio of true positives to the sum of true positives and false positives.

- **Interpretation:** It measures the accuracy of positive predictions and the ability to avoid false positives.

Use When

- Minimizing false positives is critical.
- There is a need to focus on the reliability of positive predictions.

4) *Recall (Sensitivity or True Positive Rate)*

Recall is the ratio of true positives to the sum of true positives and false negatives.

- **Interpretation:** It measures the ability of the model to capture all the relevant instances.

Use When

- Identifying all positive instances is crucial.
- False negatives are costly.

5) *Specificity (True Negative Rate)*

Specificity is the ratio of true negatives to the sum of true negatives and false positives.

- **Interpretation:** It measures the model's ability to correctly identify negative instances.

Use When

- Reducing false positives is important.
- Emphasizing the accuracy of negative predictions is crucial.

6) *F1 Score*

The F1 Score is the harmonic mean of precision and recall.

- **Interpretation:** It balances precision and recall, providing a single metric for model evaluation.

Use When

- Achieving a balance between precision and recall is essential.
- There is an imbalance between positive and negative instances.

7) *Area Under the ROC Curve (AUC-ROC)*

AUC-ROC measures the area under the Receiver Operating Characteristic (ROC) curve.

- **Interpretation:** It assesses the model's ability to distinguish between positive and negative instances.

Use When

- Evaluating the overall performance of a binary classifier.
- Comparing models across different thresholds.

8) *Area Under the Precision-Recall Curve (AUC-PRC)*

AUC-PRC measures the area under the Precision-Recall curve.

- **Interpretation:** It evaluates the precision-recall trade-off across different thresholds.

Use When

- Focusing on the performance of a classifier at various precision-recall levels.
- Dealing with imbalanced datasets.

9) *Cohen's Kappa*

Cohen's Kappa measures the agreement between predicted and actual classifications, considering chance agreement.

- **Interpretation:** It provides a measure of classification accuracy while accounting for chance.

Use When

- There is a need to assess inter-rater reliability.
- Accounting for chance agreement is important.

10) *Matthews Correlation Coefficient (MCC)*

MCC is a balanced measure of classification performance, considering true positives, true negatives, false positives, and false negatives.

- **Interpretation:** It considers both positive and negative class predictions, providing a balanced metric.

Use When

- Evaluating binary classification tasks.
- Dealing with imbalanced datasets.

These metrics offer a comprehensive understanding of a classifier's performance, and the choice depends on the specific requirements and characteristics of the classification problem at hand.

These metrics collectively provide insights into the performance of clustering, regression, and classification models, helping practitioners make informed decisions about their suitability for specific tasks.

Model Evaluation in Regression

Regression models aim to establish relationships between dependent and independent variables. Evaluating these models is crucial to understand their predictive performance. Here are some common evaluation metrics for regression:

1) *Mean Absolute Error (MAE)*

The Mean Absolute Error is the average of the absolute differences between predicted and actual values.

- **Interpretation:** It represents the average magnitude of errors and is easy to understand.

Use When

All errors (overpredictions and underpredictions) are equally important.

2) *Mean Squared Error (MSE)*

The Mean Squared Error is the average of the squared differences between predicted and actual values.

- **Interpretation:** It penalizes larger errors more heavily than MAE.

Use When

Outliers in the data are important, and you want to minimize their impact.

3) *Root Mean Squared Error (RMSE)*

The Root Mean Squared Error is the square root of the average of the squared differences between predicted and actual values.

- **Interpretation:** It provides an error metric in the same units as the target variable.

Use When

Squared errors need to be converted back to the original scale.

4) *R-squared (R²)*

R-squared represents the proportion of the variance in the dependent variable that is predictable from the independent variables.

- **Interpretation:** It indicates the proportion of variance explained by the model.

Use When

You want a standardized metric that ranges from 0 to 1.

5) *Adjusted R-squared*

Adjusted R-squared is a modified version of R-squared that adjusts for the number of predictors in the model.

- **Interpretation:** It considers the complexity of the model by penalizing for overfitting.

Use When

Multiple predictors are present, and you want to avoid overfitting.

6) *Mean Percentage Error (MPE)*

Mean Percentage Error is the mean of the percentage differences between predicted and actual values.

- **Interpretation:** It represents the average percentage error in predictions.

Use When

Positive and negative errors have equal importance.

7) *Mean Absolute Percentage Error (MAPE)*

Mean Absolute Percentage Error is the mean of the absolute percentage differences between predicted and actual values.

- **Interpretation:** It is easy to interpret in percentage terms and penalizes large percentage errors more heavily.

Use When

Percentage errors are more meaningful, such as in financial forecasting.

8) *Coefficient of Determination (COD)*

The Coefficient of Determination is a measure of the proportion of variance in the dependent variable predictable from the independent variables.

- **Interpretation:** It is suitable for nonlinear regression models.

Use When

R-squared may not be applicable, and a metric for nonlinear models is needed.

9) *Mean Bias Deviation (MBD)*

Mean Bias Deviation is the mean of the differences between predicted and actual values.

- **Interpretation:** It measures the average bias in predictions.

Use When

Positive and negative biases have equal importance.

Model Evaluation in Forecasting

Timeseries forecasting considers timely (weekly, monthly, or yearly) data in as input, identifies relation of the data values with the dependent parameters like months, holidays, occasions, announcements and forecasts the output for pre-defined future time. Evaluation of the output data or model fitting/training in the forecast pipelines, involves measuring the errors with the actual outputs or the values. Different metrics which are used in calculating the errors are listed below:

1. Root Mean Squared Error (RMSE)
2. Normalized Root Mean Squared Error (N-RMSE)
3. Mean Absolute Error (MAE)
4. R-Squared Error
5. Mean Absolute Percentage Error (MAPE)

Please refer to the section [

Monitor Evaluation in Databricks - MLflow

MLflow is a lightweight set of APIs and user interfaces that can be used with any ML framework throughout the Machine Learning workflow. It includes four components: MLflow Tracking, MLflow Projects, MLflow Models and MLflow Model Registry.

The MLflow Model Registry component is a centralized model store, set of APIs, and UI, to collaboratively manage the full lifecycle of MLflow Models. It provides model lineage (which MLflow Experiment and Run produced the model), model versioning, stage transitions, annotations, and deployment management.

State of Model Evaluation in MLflow

MLflow model evaluation can happen in two ways. One using default evaluate metrics provided by `mlflow.evaluate` and another is with use of custom metric evaluation functions.

Consider following simple MLflow evaluation using MLflow Databricks. Model used in the code is XGBoost Model. Please refer to the link for more details of the model.

1. Import necessary libraries.

```
import xgboost
import shap
import mlflow
from sklearn.model_selection import train_test_split
from mlflow.models import infer_signature
```

2. Split the dataset in training and testing dataset.

```
# load UCI Adult Data Set; segment it into training and test sets
X, y = shap.datasets.adult()
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33,
random_state=42)

# train XGBoost model
model = xgboost.XGBClassifier().fit(X_train, y_train)

# infer model signature
predictions = model.predict(X_train)
signature = infer_signature(X_train, predictions)
```

3. Construct evaluation dataset from test set.

```
eval_data = X_test
eval_data["target"] = y_test
```

4. Now use MLflow and call evaluate method provided by MLflow.

```
with mlflow.start_run() as run:
    model_info = mlflow.sklearn.log_model(model, "model", signature=signature)
    result = mlflow.evaluate(
        model_info.model_uri,
        eval_data,
        targets="target",
        model_type="classifier",
        evaluators=["default"],
    )
```

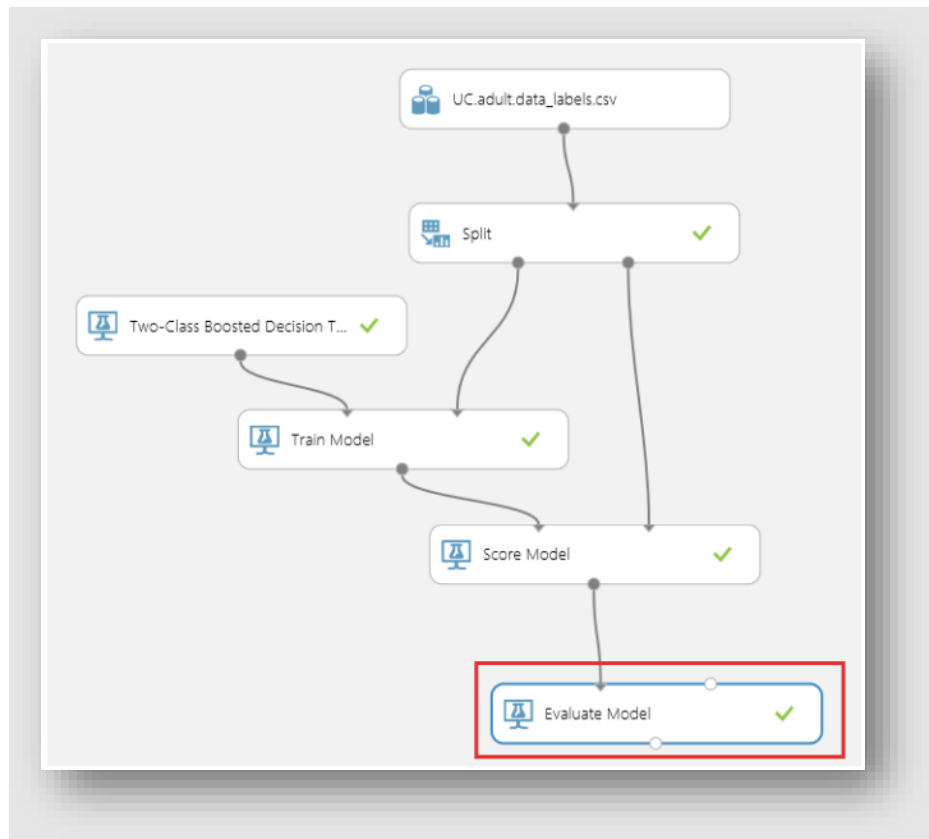
5. It logs accuracy, f1 score, false negatives, false positives, true negatives and true positives along with recall.

Please refer to the [link](#) for more details and custom function implementation with tracking. Further things can be found for MLflow in Documentation.

Set Up Model Evaluation in Azure ML

1. Access Azure ML Studio
 - a. Open Azure ML Studio in your web browser.
2. Navigate to Machine Learning Module
 - a. On the left navigation pane of Azure ML Studio, locate and click on the "Machine Learning" module.
3. Select Evaluate Submodule
 - a. Within the Machine Learning module, find and select the "Evaluate" submodule.

4. Add Evaluate Model Module to Visual Designer
 - a. Drag the "Evaluate Model" module from the options onto the visual designer surface.
5. Position the Module
 - a. Place the "Evaluate Model" module below the "Score Model" module on the visual designer surface.
6. Connect Modules
 - a. Connect the output of the "Split" module to the "Score Model" module. This is typically done by dragging a line or arrow from the output of the "Split" module to the input of the "Score Model" module.
 - b. Connect the output of the "Score Model" module to the left side of the "Evaluate Model" module using the dragging method to connect them.
7. Review the Configuration
 - a. Click on the "Evaluate Model" module to configure and review its settings. This might involve specifying which metrics to evaluate, selecting the model you want to evaluate, etc.
8. Save and Execute
 - a. Save your work in Azure ML Studio.
9. Run the Experiment
 - a. Execute the experiment in Azure ML Studio to run the evaluation process.
10. Review Results
 - a. Once the evaluation is complete, review the results to assess the accuracy and performance of your model.



Code based Explanation

1. Import necessary libraries.

```
from azureml.core import Workspace, Experiment
from azureml.train.automl.run import AutoMLRun
from azureml.core.model import Model
from azureml.core import Dataset
from azureml.widgets import RunDetails
from azureml.train.automl.runtime import AutoMLRuntime
from azureml.train.automl import constants
from azureml.core.authentication import InteractiveLoginAuthentication
from azureml.core.compute import AmlCompute
```

2. Connect to your Azure Machine Learning workspace.

```
# Load your workspace from the config file
ws = Workspace.from_config()
```

3. Load the registered model.

```
# Load the model by providing the model name and version
```

```
model_name = "your_model_name"
model_version = "your_model_version"
model = Model(ws, name=model_name, version=model_version)
```

4. Load the test dataset.

```
# Load your test dataset
test_dataset = Dataset.get_by_name(ws, name="your_test_dataset_name")
```

5. Create a script for model evaluation.

a. Save this script, for example, as evaluate_model.py.

```
import pandas as pd
from azureml.core import Model
from azureml.core.run import Run
from sklearn.metrics import accuracy_score

# Load the model
model = Model(ws, "your_model_name", version="your_model_version")

# Load the test dataset
test_data = test_dataset.to_pandas_dataframe()

# Your evaluation code here
# For example, if your model is a binary classifier
predictions = model.predict(test_data.drop(columns=["label_column"]))
accuracy = accuracy_score(test_data["label_column"], predictions)

# Log the accuracy metric
run = Run.get_context()
run.log("accuracy", accuracy)
```

6. Create an Experiment.

```
# Create an experiment
experiment_name = 'model-evaluation'
experiment = Experiment(workspace=ws, name=experiment_name)
```

7. Run the evaluation script.

```
# Run the evaluation script
run = experiment.start_logging()
run.upload_file("evaluate_model.py", "evaluate_model.py")
```

```
run.script_run("evaluate_model.py", arguments=[]) # Add any additional
arguments if needed
run.complete()
```

8. View the results.

```
# View the run details
RunDetails(run).show()
```

9. Retrieve and log evaluation metrics.

```
# Retrieve and log the evaluation metrics
metrics = run.get_metrics()
print("Accuracy:", metrics["accuracy"])
```

General Model in Azure Machine Learning & in ADB - MLFlow

1. Create an Azure Machine Learning Workspace
 - a. Create an Azure Machine Learning workspace using the Azure portal or Azure ML SDK.
2. Prepare and Upload Your Dataset
 - a. Prepare your dataset for training and testing.
 - b. Upload the dataset to your Azure ML workspace.
3. Define and Train a Classification Model
 - a. Define and train your classification model using Python and Azure ML SDK. This can involve selecting a machine learning algorithm, splitting your dataset into training and testing sets, and training the model.
4. Evaluate Model Performance + Track the model
 - a. After training the model, evaluate its performance using metrics such as accuracy. You can use test data for evaluation.
 - b. Calculate accuracy using the formula:
$$\text{Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Total Number of Predictions}}$$
5. Register the Model
 - a. After testing the model's performance, register the model in the Azure ML workspace.
6. Set Up Scoring Script
 - a. Create a scoring script that defines how to use the trained model for making predictions.
 - b. This script typically includes loading the model, making predictions, and returning the results.
7. Deploying the Model as a Web Service
 - a. Deploy the model as a web service using Azure Container Instances (ACI) or Azure Kubernetes Service (AKS). This makes the model accessible through an HTTP endpoint.
8. Set Up an Azure ML Pipeline
 - a. For a more automated and reproducible approach, consider setting up an Azure ML pipeline. A pipeline can include steps for data preparation, model training, model evaluation, and deployment.
9. Monitor and Manage the Model
 - a. Set up monitoring for your deployed model using Azure Application Insights to track usage, performance, and other metrics.
 - b. Use Azure ML capabilities to manage and version your models.

Same steps are applied to Databricks MLflow. Refer to the link for detailed information of model registry, model version update, model training and batch inference.

APPENDIX A – Evaluation Metrics] for the details and mathematical explanations for the mentioned metrics. Out of the mentioned metrics RMSE, MAPE, MAE and R-Squared Error are the preferred metrics. MAE and RMSE are commonly used and provide a good balance between simplicity and capturing different aspects of an error. MAPE is useful when percentage-based errors are important but be cautious of its sensitivity to zero values and R-Squared error is used for understanding the variability captured by the model, especially in regression-type forecasting tasks.

RMSE, MAE and MAPE majorly check differences between the predicted and observed values. For these metrics, **LOWER** values indicate that the model performance is good. Whereas for R-Squared Error, it measures proportion of the variability in the dependent variable (i.e., Time if no other variable is added). Higher R-Squared Error (~ 1) states a very large proportion of change in the data.

Choosing the Right Regression/Forecasting Metric

MAE, MSE, RMSE: These are common metrics and suitable for most regression tasks. Choose based on the desired level of sensitivity to outliers.

R-squared and Adjusted R-squared: These are useful for understanding the overall fit of the model and assessing the impact of adding predictors.

MAPE and MPE: Use when percentage errors are more meaningful, such as in financial forecasting.

COD: Useful for nonlinear regression models where R-squared may not be applicable.

Model Evaluation in Clustering

Clustering involves grouping data points based on their similarities. Model evaluation in clustering is crucial to assess the effectiveness of the clustering algorithm and the quality of the formed clusters. The evaluation metrics are essential for comparing the clustering results with the true underlying structure of the data. Here are some commonly used metrics for evaluating clustering models:

1) *Silhouette Score*

Measures of how well-separated clusters are. It ranges from -1 to 1, where a higher value indicates better-defined clusters.

- **Interpretation:** Closer to 1 is better, indicating well-separated clusters.

Use When

Assessing the separation and distinctiveness of clusters is crucial.

2) *Davies-Bouldin Index*

Quantifies the compactness and separation between clusters. Lower values indicate better clustering.

- **Interpretation:** Lower values are preferable, suggesting more compact and well-separated clusters.

Use When

Emphasizing both compactness and separation in cluster formation is important.

3) *Calinski-Harabasz Index*

Computes the ratio of between-cluster variance to within-cluster variance. Higher values imply better-defined clusters.

- **Interpretation:** Higher values are better, indicating well-separated clusters.

Use When

Evaluating the balance between within-cluster homogeneity and between-cluster separation is essential.

4) *Rand Index (RI)*

Measures the similarity between true and predicted clustering assignments by considering the pairs of data points and whether they are grouped together or not.

- **Interpretation:** A higher Rand Index indicates a better agreement between true and predicted clusters, with 1 indicating perfect agreement.

Use When

Assessing the overall agreement between true and predicted clusters is a primary concern.

5) *Adjusted Mutual Information (AMI)*

Measures the mutual information between true and predicted clusters while adjusting for chance. It considers the information shared between the cluster assignments.

- **Interpretation:** A higher AMI suggests a better alignment between true and predicted clusters, with 1 indicating perfect agreement. It accounts for chance agreement, making it suitable for varying cluster sizes and random chance.

Use When

Accounting for chance agreement and varying cluster sizes is necessary.

Choosing the Right Evaluation Metric

The nature and objectives of a clustering problem will dictate the most appropriate assessment measure to employ. If the goal of clustering is to group similar data points together, the Calinski-Harabasz index or the silhouette score can be beneficial. If the clustering results need to be compared to ground truth clustering, however, the Rand index or AMI would be more appropriate. So, it is important to consider the objectives and constraints of the clustering issue while selecting the evaluation metric.

Evaluating the Stability of Clustering Results

Clustering has certain challenges since the parameters of the algorithm and the initial conditions may affect the results. It is essential to execute the clustering technique repeatedly using multiple random initializations or settings in order to judge the sustainability of the clustering findings. One can evaluate the stability of the clustering results using metrics such as the Jaccard index or the variance of information.

Setup Model Evaluation Mechanism for a Model

Model evaluation in the ML model can be directly integrated into the model code itself. It will also consider the splitting ratio of training-testing data along with error metrics of the output. Larger training data is preferred as it provides more insights to the model for getting the patterns and it also provides a healthy amount of data after splitting for testing also.

If model has different parameters, then a search grid can also be applied. All possible combinations of different model parameters are created as grid, and model is trained for each group of parameters. In this method, model splitting happens in three, training, validation and testing data. Grid search is over training + validation data, where model is trained for the group of parameter values and evaluated for validation data. All the grid parameters groups and the output metrics values are noted, and then the best possible group of parameters are taken for final training.

In the final training, training and validations sets are merged in training sets, and testing set is used for evaluation of the model.

Dataset Splitting

Any data set is majorly split into training, validation, and testing phase. Training and Validation phase are majorly used during the training of the model, where effects of different parameters of model fitting is evaluated and its performance is validated on the validation set. Based on the outcomes a model with specific parameter value is chosen to be a final model and further proceeded for the final training.

During the final training, training + validation sets are combined to be a full training set and the model evaluation happens on the testing set. For splitting and evaluation multiple methods are used as given below:

1. Train-Test Split
2. K-Fold Cross Validation
3. Stratified K-Fold Cross Validation
4. Time Series Split
5. Leave-One-Out Cross-Validation

1) *Train-Test Split*

Method Description:

In the Train-Test Split method, the dataset is divided into two subsets: a training set used to train the model, and a testing set used to evaluate its performance. Typically, a certain percentage of the data is allocated for training, and the remaining portion is used for testing.

Sample Example:

```
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_iris

# Load the Iris dataset
iris = load_iris()
X, y = iris.data, iris.target

# Perform a train-test split with 80% training data and 20% testing data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)
```

When to Use: Suitable for relatively large datasets.

Advantages: Simple, computationally efficient.

Considerations: May not provide robust estimates, especially with small datasets.

2) *K-Fold Cross-Validation*

Method Description

In K-Fold Cross-Validation, the dataset is divided into 'K' folds. The model is trained and evaluated 'K' times, each time using a different fold as the test set and the remaining data as the training set. The final performance metric is often an average of the metrics obtained in each iteration.

Sample Example

```
from sklearn.model_selection import cross_val_score, KFold
from sklearn.ensemble import RandomForestClassifier
```

```
# Load the Iris dataset
iris = load_iris()
X, y = iris.data, iris.target

# Create a RandomForestClassifier
clf = RandomForestClassifier()

# Perform 5-fold cross-validation
kf = KFold(n_splits=5, shuffle=True, random_state=42)
scores = cross_val_score(clf, X, y, cv=kf)

# The 'scores' variable contains the performance metrics for each fold
```

When to Use: Generally, a good choice for a wide range of datasets.

Advantages: Reduces variability, provides a better estimate of model performance.

Consideration: Computationally more expensive than a simple train-test split.

3) *Leave-One-Out Cross-Validation (LOOCV)*

Method Description

In Leave-One-Out Cross-Validation, each data point is used as a test set exactly once, while the rest of the data is used for training. This is a special case of K-Fold Cross-Validation where 'K' is equal to the number of samples in the dataset.

Sample Code

```
from sklearn.model_selection import cross_val_score, LeaveOneOut
from sklearn.linear_model import LogisticRegression

# Load the Iris dataset
iris = load_iris()
X, y = iris.data, iris.target

# Create a Logistic Regression model
model = LogisticRegression()

# Perform Leave-One-Out Cross-Validation
loo = LeaveOneOut()
scores = cross_val_score(model, X, y, cv=loo)

# The 'scores' variable contains the performance metrics for each iteration
```

When to Use: Suitable for small to moderately sized datasets.

Advantages: Provides a nearly unbiased estimate but can be computationally expensive.

Considerations: Can be sensitive to outliers, may lead to overfitting.

4) *Stratified K-Fold Cross-Validation*

Method Description

Stratified K-Fold Cross-Validation ensures that each fold maintains the same class distribution as the entire dataset. This is particularly useful when dealing with imbalanced datasets.

```
from sklearn.model_selection import StratifiedKFold

# Assuming X and y are your features and labels
skf = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

for train_index, test_index in skf.split(X, y):
    X_train, X_test = X[train_index], X[test_index]
    y_train, y_test = y[train_index], y[test_index]
    # Your model training and evaluation here
```

When to Use: Important for imbalanced datasets, where certain classes have fewer samples.

Advantages: Ensures class distribution balance across folds.

Considerations: Computationally more expensive than simple K-Fold.

5) *Time Series Split*

Method Description

Time Series Split is suitable for time-ordered datasets. It ensures that training data comes before testing data, maintaining the temporal order of observations.

Sample Code

```
from sklearn.model_selection import TimeSeriesSplit

# Assuming X and y are your features and labels
tscv = TimeSeriesSplit(n_splits=5)

for train_index, test_index in tscv.split(X):
    X_train, X_test = X[train_index], X[test_index]
    y_train, y_test = y[train_index], y[test_index]
    # Your model training and evaluation here
```

When to Use: Appropriate for time-ordered datasets.

Advantages: Maintains temporal order, suitable for time series forecasting.

Considerations: May not be suitable for all types of datasets.

6) *GroupKFold*

Method Description

GroupKFold is used when samples are grouped, and each group is kept intact during training and testing. This is useful when you want to ensure that all samples from a specific group are either in the training or testing set.

```
from sklearn.model_selection import GroupKFold

# Assuming X, y, and groups are your features, labels, and group identifiers
gkf = GroupKFold(n_splits=5)

for train_index, test_index in gkf.split(X, y, groups=groups):
    X_train, X_test = X[train_index], X[test_index]
    y_train, y_test = y[train_index], y[test_index]
    # Your model training and evaluation here
```

When to Use: Useful when samples are grouped (e.g., different subjects, experiments).

Advantages: Ensures that groups are not split between training and testing sets.

Considerations: Requires information about groupings, may not be suitable for all datasets.

Set Up Model Evaluation in DataBricks (Code based)

1. Define a function for evaluation metrics calculation.
 - a. A metric evaluation function can be either defined as coded function.
 - b. Or a library with pre-defined measures can be used.

Using library functions are best, If the model input output can be restructured in the form of the input output parameters of the library-based functions. Different libraries like Scikit, Tensorflow, and Keras provide model evaluation functions supported in python. These functions are optimized to provide output in an efficient way, as compared to the code function defined manually in the model code notebooks.

If custom code/ metric is required, it should be defined in the form of a function like given below:

```
def calculate_accuracy(y_true, y_pred):
    """
    Calculate accuracy given the true labels (y_true) and predicted labels
    (y_pred).
```

```

Parameters:
- y_true: Array of true labels
- y_pred: Array of predicted labels

Returns:
- Accuracy score
"""
correct_predictions = sum(1 for true, pred in zip(y_true, y_pred) if true == pred)
total_samples = len(y_true)
accuracy = correct_predictions / total_samples
return accuracy

```

2. Call the function once output is generated.
 - a. If the library functions are considered for model evaluations then the original input and output must be restructured in the format that the function parameters required.

```

from sklearn.metrics import accuracy_score
accuracy = accuracy_score(y_true, y_pred)

```

3. Log the parameters in a file/table.
 - a. If the grid search is used with all the parameters values, then those values should be logged into a file or a table along with the evaluation metrics calculated.
 - b. Consider following Sample Code for the details.
4. Put alerts/ values monitoring model forecasting.
 - a. Let say, If we have a model automation up and running, we must need to monitor it.
 - b. The logged values of evaluation metrics should be noted, and outliers (very unexpected) values should be called out.
 - c. Please refer to the following code where unexpected values are called out by raising exception. However, different callout mechanisms like sending mail, stopping the also can be configured to stop model from proceeding ahead.

Sample Code

```

import pandas as pd
from sklearn.model_selection import TimeSeriesSplit
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import ParameterGrid
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split

# Sample data

```

```

# Assume 'X' is your feature matrix and 'y' is your target variable for time
series forecasting
# Replace this with your actual data
X, y = ...

# Define grid search parameters
param_grid = {
    'train_size': [0.6, 0.7, 0.8], # Training set size
    'test_size': [0.2], # Test set size
    'model_param1': [value1, value2], # Replace with actual model parameters
    'model_param2': [value3, value4] # Replace with actual model parameters
}

# Create DataFrame to store results
columns = ['Train Size', 'Test Size', 'Model Param 1', 'Model Param 2',
'MSE']
results_df = pd.DataFrame(columns=columns)

# Perform grid search
for params in ParameterGrid(param_grid):
    # Split data based on grid search parameters
    train_size = params['train_size']
    test_size = params['test_size']
    X_train, X_test, y_train, y_test = train_test_split(X, y,
train_size=train_size, test_size=test_size, shuffle=False)

    # Create and train your forecasting model with the specified parameters
    model = YourForecastingModel(param1=params['model_param1'],
param2=params['model_param2'])
    model.fit(X_train, y_train)

    # Make predictions
    y_pred = model.predict(X_test)

    try:
        # Evaluate the model using mean squared error (MSE)
        mse = mean_squared_error(y_test, y_pred)
        if not (0 <= mse < float('inf')):
            raise ValueError("Unexpected value for MSE")
    except Exception as e:
        # Handle unexpected values of MSE
        print(f"Error calculating MSE: {e}")
        mse = None # Set to a default value or handle as needed

    # Append results to DataFrame

```



```

    result_row = [train_size, test_size, params['model_param1'],
params['model_param2'], mse]
    results_df = results_df.append(pd.Series(result_row, index=columns),
ignore_index=True)

# Display the results DataFrame
print(results_df)

```

Please refer to the Databricks documentation on forecasting [\[Link\]](#).

Monitor Evaluation in Databricks - MLflow

MLflow is a lightweight set of APIs and user interfaces that can be used with any ML framework throughout the Machine Learning workflow. It includes four components: MLflow Tracking, MLflow Projects, MLflow Models and MLflow Model Registry.

The MLflow Model Registry component is a centralized model store, set of APIs, and UI, to collaboratively manage the full lifecycle of MLflow Models. It provides model lineage (which MLflow Experiment and Run produced the model), model versioning, stage transitions, annotations, and deployment management.

State of Model Evaluation in MLflow

MLflow model evaluation can happen in two ways. One using default evaluate metrics provided by mlflow.evaluate and another is with use of custom metric evaluation functions.

Consider following simple MLflow evaluation using MLflow Databricks. Model used in the code is XGBoost Model. Please refer to the [link](#) for more details of the model.

6. Import necessary libraries.

```

import xgboost
import shap
import mlflow
from sklearn.model_selection import train_test_split
from mlflow.models import infer_signature

```

7. Split the dataset in training and testing dataset.

```

# load UCI Adult Data Set; segment it into training and test sets
X, y = shap.datasets.adult()
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33,
random_state=42)

# train XGBoost model

```

```

model = xgboost.XGBClassifier().fit(X_train, y_train)

# infer model signature
predictions = model.predict(X_train)
signature = infer_signature(X_train, predictions)

```

8. Construct evaluation dataset from test set.

```

eval_data = X_test
eval_data["target"] = y_test

```

9. Now use MLflow and call evaluate method provided by MLflow.

```

with mlflow.start_run() as run:
    model_info = mlflow.sklearn.log_model(model, "model", signature=signature)
    result = mlflow.evaluate(
        model_info.model_uri,
        eval_data,
        targets="target",
        model_type="classifier",
        evaluators=["default"],
    )

```

10. It logs accuracy, f1 score, false negatives, false positives, true negatives and true positives along with recall.

Please refer to the [link](#) for more details and custom function implementation with tracking. Further things can be found for MLflow in [Documentation](#).

Set Up Model Evaluation in Azure ML

11. Access Azure ML Studio
 - a. Open Azure ML Studio in your web browser.
12. Navigate to Machine Learning Module
 - a. On the left navigation pane of Azure ML Studio, locate and click on the "Machine Learning" module.
13. Select Evaluate Submodule
 - a. Within the Machine Learning module, find and select the "Evaluate" submodule.
14. Add Evaluate Model Module to Visual Designer
 - a. Drag the "Evaluate Model" module from the options onto the visual designer surface.
15. Position the Module
 - a. Place the "Evaluate Model" module below the "Score Model" module on the visual designer surface.

16. Connect Modules

- a. Connect the output of the "Split" module to the "Score Model" module. This is typically done by dragging a line or arrow from the output of the "Split" module to the input of the "Score Model" module.
- b. Connect the output of the "Score Model" module to the left side of the "Evaluate Model" module using the dragging method to connect them.

17. Review the Configuration

- a. Click on the "Evaluate Model" module to configure and review its settings. This might involve specifying which metrics to evaluate, selecting the model you want to evaluate, etc.

18. Save and Execute

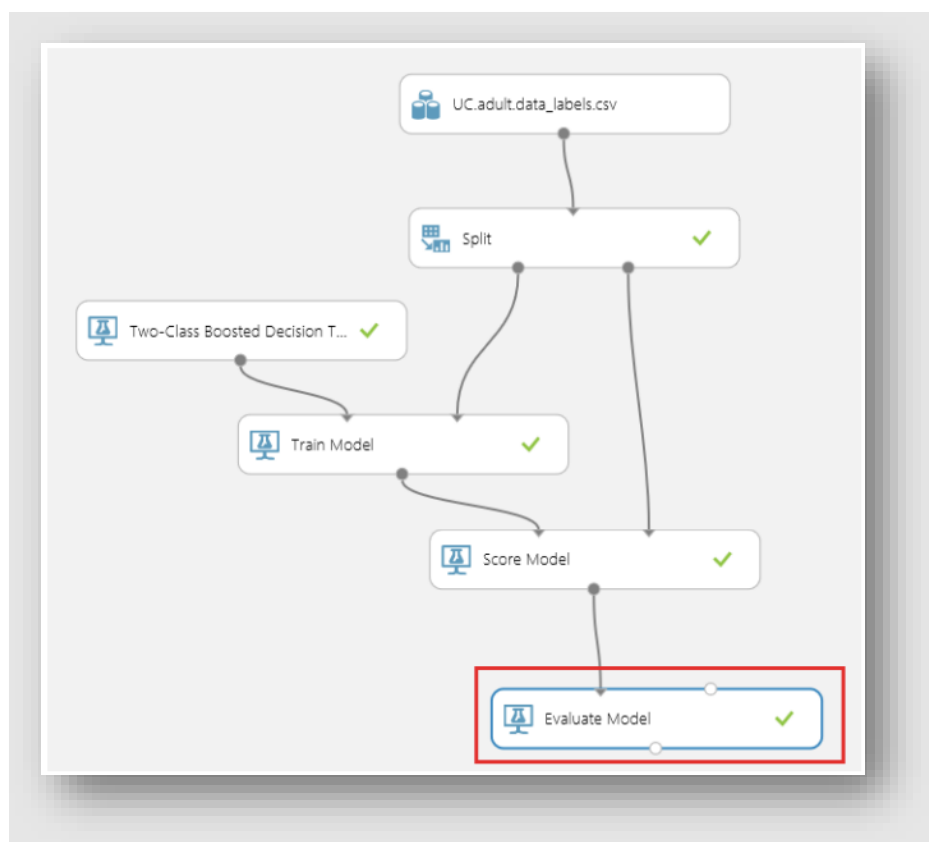
- a. Save your work in Azure ML Studio.

19. Run the Experiment

- a. Execute the experiment in Azure ML Studio to run the evaluation process.

20. Review Results

- a. Once the evaluation is complete, review the results to assess the accuracy and performance of your model.



Code based Explanation

10. Import necessary libraries.

```
from azureml.core import Workspace, Experiment
from azureml.train.automl.run import AutoMLRun
from azureml.core.model import Model
from azureml.core import Dataset
from azureml.widgets import RunDetails
from azureml.train.automl.runtime import AutoMLRuntime
from azureml.train.automl import constants
from azureml.core.authentication import InteractiveLoginAuthentication
from azureml.core.compute import AmlCompute
```

11. Connect to your Azure Machine Learning workspace.

```
# Load your workspace from the config file
ws = Workspace.from_config()
```

12. Load the registered model.

```
# Load the model by providing the model name and version
model_name = "your_model_name"
model_version = "your_model_version"
model = Model(ws, name=model_name, version=model_version)
```

13. Load the test dataset.

```
# Load your test dataset
test_dataset = Dataset.get_by_name(ws, name="your_test_dataset_name")
```

14. Create a script for model evaluation.

a. Save this script, for example, as evaluate_model.py.

```
import pandas as pd
from azureml.core import Model
from azureml.core.run import Run
from sklearn.metrics import accuracy_score

# Load the model
model = Model(ws, "your_model_name", version="your_model_version")

# Load the test dataset
test_data = test_dataset.to_pandas_dataframe()
```

```
# Your evaluation code here
# For example, if your model is a binary classifier
predictions = model.predict(test_data.drop(columns=["label_column"]))
accuracy = accuracy_score(test_data["label_column"], predictions)

# Log the accuracy metric
run = Run.get_context()
run.log("accuracy", accuracy)
```

15. Create an Experiment.

```
# Create an experiment
experiment_name = 'model-evaluation'
experiment = Experiment(workspace=ws, name=experiment_name)
```

16. Run the evaluation script.

```
# Run the evaluation script
run = experiment.start_logging()
run.upload_file("evaluate_model.py", "evaluate_model.py")

run.script_run("evaluate_model.py", arguments=[]) # Add any additional
arguments if needed
run.complete()
```

17. View the results.

```
# View the run details
RunDetails(run).show()
```

18. Retrieve and log evaluation metrics.

```
# Retrieve and log the evaluation metrics
metrics = run.get_metrics()
print("Accuracy:", metrics["accuracy"])
```

General Model in Azure Machine Learning & in ADB - MLFlow

10. Create an Azure Machine Learning Workspace
 - a. Create an Azure Machine Learning workspace using the Azure portal or Azure ML SDK.
11. Prepare and Upload Your Dataset
 - a. Prepare your dataset for training and testing.
 - b. Upload the dataset to your Azure ML workspace.
12. Define and Train a Classification Model
 - a. Define and train your classification model using Python and Azure ML SDK. This can involve selecting a machine learning algorithm, splitting your dataset into training and testing sets, and training the model.
13. Evaluate Model Performance + Track the model
 - a. After training the model, evaluate its performance using metrics such as accuracy. You can use test data for evaluation.
 - b. Calculate accuracy using the formula:
$$\text{Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Total Number of Predictions}}$$
14. Register the Model
 - a. After testing the model's performance, register the model in the Azure ML workspace.
15. Set Up Scoring Script
 - a. Create a scoring script that defines how to use the trained model for making predictions.
 - b. This script typically includes loading the model, making predictions, and returning the results.
16. Deploying the Model as a Web Service
 - a. Deploy the model as a web service using Azure Container Instances (ACI) or Azure Kubernetes Service (AKS). This makes the model accessible through an HTTP endpoint.
17. Set Up an Azure ML Pipeline
 - a. For a more automated and reproducible approach, consider setting up an Azure ML pipeline. A pipeline can include steps for data preparation, model training, model evaluation, and deployment.
18. Monitor and Manage the Model
 - a. Set up monitoring for your deployed model using Azure Application Insights to track usage, performance, and other metrics.
 - b. Use Azure ML capabilities to manage and version your models.

Same steps are applied to Databricks MLflow. Refer to the [link](#) for detailed information of model registry, model version update, model training and batch inference.

APPENDIX A – Evaluation Metrics

Forecast Evaluation Metrics

1) Mean Absolute Error (MAE)

Definition: The Mean Absolute Error measures the average absolute differences between actual and predicted values.

Formula: $MAE = \frac{1}{n} \sum_{i=1}^n |Y_i - \hat{Y}_i|$

Code Relevance:

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

Mean Absolute Scaled Error (MASE)

Definition: MASE compares the mean absolute error of the forecasting model to the mean absolute error of a naïve forecast.

Formula: $MASE = \frac{MAE}{\frac{1}{n-m} \sum_{i=m+1}^n |Y_i - Y_{i-m}|}$ m is the seasonality or frequency of the data.

Code Relevance:

```
from statsmodels.tsa.stattools import acf
from statsmodels.tools.eval_measures import meanabs
from statsmodels.tsa.statespace.tools import diff

# Assuming y_true and y_pred are NumPy arrays or lists
def mean_absolute_scaled_error(y_true, y_pred, seasonal_period=1):
    d = diff(y_true, seasonal_period)
    q = acf(y_true, fft=False, nlags=1)
    scaled_error = meanabs(y_true - y_pred) / meanabs(d)
    return scaled_error / (1 - sum(q))
```

2) Mean Absolute Percentage Error (MAPE)

Definition: MAPE expresses the prediction error as a percentage of the actual values.

Formula: $MAPE = \frac{1}{n} \sum_{i=1}^n \left| \frac{Y_i - \hat{Y}_i}{Y_i} \right| \times 100$

Code Relevance:

```
from sklearn.metrics import mean_absolute_error
import numpy as np

# Assuming y_true and y_pred are NumPy arrays or lists
def mean_absolute_percentage_error(y_true, y_pred):
    y_true, y_pred = np.array(y_true), np.array(y_pred)
    return np.mean(np.abs((y_true - y_pred) / y_true)) * 100
```

3) Median Absolute Percentage Error (MdAPE)

Definition: Similar to MAPE, MdAPE uses the median absolute percentage error, making it less sensitive to outliers.

Formula:
$$MdAPE = \text{Median} \left(\left| \frac{Y_i - \hat{Y}_i}{Y_i} \right| \right) \times 100$$

Code Relevance:

```
import numpy as np

# Assuming y_true and y_pred are NumPy arrays or lists
def median_absolute_percentage_error(y_true, y_pred):
    y_true, y_pred = np.array(y_true), np.array(y_pred)
    mask = y_true != 0 # Avoid division by zero
    percentage_errors = np.abs((y_true[mask] - y_pred[mask]) / y_true[mask]) *
100
    return np.median(percentage_errors)
```

4) Symmetric Mean Absolute Percentage Error (SMAPE)

Definition: SMAPE is similar to MAPE but symmetric, which means it does not disproportionately penalize underestimates or overestimates.

Formula:
$$SMAPE = \frac{1}{n} \sum_{i=1}^n \frac{2 \times |Y_i - \hat{Y}_i|}{|Y_i| + |\hat{Y}_i|} \times 100$$

Code Relevance:

```
import numpy as np

# Assuming y_true and y_pred are NumPy arrays or lists
def symmetric_mean_absolute_percentage_error(y_true, y_pred):
    y_true, y_pred = np.array(y_true), np.array(y_pred)
    numerator = np.abs(y_pred - y_true)
    denominator = (np.abs(y_true) + np.abs(y_pred)) / 2
    percentage_errors = np.divide(numerator, denominator,
out=np.zeros_like(numerator), where=denominator != 0) * 100
```



```
return np.mean(percentage_errors)
```

5) Mean Squared Error (MSE)

Definition: The Mean Squared Error measures the average of the squared differences between actual and predicted values.

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

Code Relevance:

```
from sklearn.metrics import mean_squared_error

# Assuming y_true and y_pred are NumPy arrays or lists
mse = mean_squared_error(y_true, y_pred)
```

6) Root Mean Squared Error (RMSE)

Definition: RMSE is a measure of the average magnitude of the errors between predicted and observed values. It gives more weight to larger errors.

Formula:
$$RMSE = \sqrt{MSE}$$

Code Relevance:

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

# Assuming y_true and y_pred are NumPy arrays or lists
mse = mean_squared_error(y_true, y_pred)
rmse = np.sqrt(mse)
```

7) Normalized Root Mean Squared Error (N-RMSE)

Definition: NRMSE is a variation of the Root Mean Squared Error (RMSE) that is normalized by the range of the data. It provides a measure of the relative accuracy of the model compared to the scale of the data.

Formula:
$$NRMSE = \frac{RMSE}{\max(Y) - \min(Y)}$$

Code Relevance:

```
import numpy as np

# Assuming y_true and y_pred are NumPy arrays or lists
```

```
def normalized_root_mean_squared_error(y_true, y_pred):
    rmse = np.sqrt(np.mean((np.array(y_true) - np.array(y_pred))**2))
    data_range = np.max(y_true) - np.min(y_true)
    nrmse = rmse / data_range
    return nrmse
```

8) Coefficient of Determination (R-squared)

Definition: R-squared represents the proportion of the variance in the dependent variable that is predictable from the independent variable(s).

Formula:
$$R^2 = 1 - \frac{\sum_{i=1}^n (Y_i - \hat{Y}_i)^2}{\sum_{i=1}^n (Y_i - \bar{Y})^2}$$
 \bar{Y} is the mean of the actual values.

Code Relevance:

```
from sklearn.metrics import r2_score

# Assuming y_true and y_pred are NumPy arrays or lists
r_squared = r2_score(y_true, y_pred)
```

9) Theil's U Statistic

Definition: Theil's U Statistic measures forecast accuracy relative to the naïve forecast (e.g., predicting the next value to be the same as the last observed value).

Formula:
$$U = \frac{\sqrt{\frac{1}{n} \sum_{i=1}^n (\hat{Y}_i - Y_i)^2}}{\sqrt{\frac{1}{n} \sum_{i=1}^n (Y_i)^2}}$$

Code Relevance:

```
import numpy as np

# Assuming y_true and y_pred are NumPy arrays or lists
def theils_u_statistic(y_true, y_pred):
    numerator = np.mean(np.abs(y_true - y_pred))
    denominator = np.mean(y_true)
    theils_u = numerator / denominator
    return theils_u
```

10) Geometric Mean Relative Absolute Error (GMRAE)

Definition: GMRAE is the geometric mean of the relative absolute errors. It is another metric that helps in assessing the accuracy of the forecasting model.

$$GMRAE = \left(\prod_{i=1}^n \frac{|Y_i - \hat{Y}_i|}{Y_i} \right)^{\frac{1}{n}}$$

Formula: _____

Code Relevance:

```
import numpy as np

# Assuming y_true and y_pred are NumPy arrays or lists
def geometric_mean_relative_absolute_error(y_true, y_pred):
    relative_absolute_errors = np.abs(y_true - y_pred) / np.abs(y_true)
    geometric_mean_rae = np.exp(np.mean(np.log(relative_absolute_errors + 1)))
    gmrae = (geometric_mean_rae - 1) * 100 # Convert to percentage
    return gmrae
```

11) Mean Bias Deviation (MBD)

Definition: MBD measures the average percentage difference between the actual and predicted values.

$$MBD = \frac{1}{n} \sum_{i=1}^n \frac{Y_i - \hat{Y}_i}{Y_i} \times 100$$

Formula: _____

Code Relevance:

```
import numpy as np

# Assuming y_true and y_pred are NumPy arrays or lists
def mean_bias_deviation(y_true, y_pred):
    bias_deviation = y_pred - y_true
    mbd = np.mean(bias_deviation)
    return mbd
```

12) Q-Statistic

Definition: Q-Statistic assesses the accuracy of a forecast by comparing the forecasted and observed values against a specified quantile.

$$Q = \frac{\text{Prob}(Y_i \leq \hat{Y}_i) - \frac{1}{2}}{\text{Prob}(Y_i \geq \hat{Y}_i)}$$

Formula: _____ Y_i is the actual value, and \hat{Y}_i is the forecasted value.

Code Relevance:

```
from scipy.stats import chi2_contingency

# Assuming data is a 2D array or DataFrame where each row represents a
# participant and each column represents a condition
data = [
    [1, 0, 1],
    [0, 1, 1],
    [1, 1, 0],
    [1, 0, 1],
]

# Perform Cochran's Q test
statistic, p_value, _, _ = chi2_contingency(data)
```

13) Mean Forecast Error (MFE)

Definition: MFE measures the average forecast error without considering the direction (overestimation or underestimation).

Formula:
$$MFE = \frac{1}{n} \sum_{i=1}^n (Y_i - \hat{Y}_i)$$

Code Relevance:

```
import numpy as np

# Assuming y_true and y_pred are NumPy arrays or lists
def mean_forecast_error(y_true, y_pred):
    forecast_error = y_pred - y_true
    mfe = np.mean(forecast_error)
    return mfe
```

Classification Machine Learning Metrics

1) Accuracy

Definition: Accuracy measures the proportion of correctly classified instances out of the total instances.

Formula:
$$\text{Accuracy} = \frac{\text{True Positives} + \text{True Negatives}}{\text{Total Population}}$$

Code Relevance:

```
from sklearn.metrics import accuracy_score
# Calculate accuracy
accuracy = accuracy_score(y_true, y_pred)
```

2) Precision

Definition: Precision measures the accuracy of positive predictions. It is the ratio of correctly predicted positive observations to the total predicted positives.

Formula:
$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}$$

Code Relevance:

```
from sklearn.metrics import precision_score
# Calculate precision
precision = precision_score(y_true, y_pred)
```

3) Recall (Sensitivity or True Positive Rate)

Definition: Recall measures the ability of the model to capture all the relevant instances. It is the ratio of correctly predicted positive observations to all actual positives.

Formula:
$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}$$

Code Relevance:

```
from sklearn.metrics import recall_score
# Calculate recall
recall = recall_score(y_true, y_pred)
```

4) F1 Score

Definition: The F1 Score is the harmonic mean of precision and recall. It provides a balance between precision and recall.

Formula:
$$F1 = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

Code Relevance:

```
from sklearn.metrics import f1_score
# Calculate F1 Score
f1 = f1_score(y_true, y_pred)
```

5) Specificity (True Negative Rate)

Definition: Specificity measures the ability of the model to correctly identify negative instances.

Formula:
$$\text{Specificity} = \frac{\text{True Negatives}}{\text{True Negatives} + \text{False Positives}}$$

Code Relevance:

```
from sklearn.metrics import confusion_matrix
# Calculate confusion matrix
conf_matrix = confusion_matrix(y_true, y_pred)
```

6) False Positive Rate (FPR)

Definition: FPR measures the proportion of actual negative instances that are incorrectly predicted as positive.

Formula:
$$\text{FPR} = \frac{\text{False Positives}}{\text{False Positives} + \text{True Negatives}}$$

Code Relevance:

```
from sklearn.metrics import confusion_matrix
# Calculate confusion matrix
conf_matrix = confusion_matrix(y_true, y_pred)
```

7) Matthews Correlation Coefficient (MCC)

Definition: MCC takes into account true and false positives and negatives and is particularly useful for imbalanced datasets.

Formula:
$$MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}}$$

- TP (True Positives): The number of instances that are correctly predicted as positive.
- TN (True Negatives): The number of instances that are correctly predicted as negative.
- FP (False Positives): The number of instances that are incorrectly predicted as positive.
- FN (False Negatives): The number of instances that are incorrectly predicted as negative.

Code Relevance:

```
from sklearn.metrics import matthews_corrcoef
# Calculate Matthews Correlation Coefficient
mcc = matthews_corrcoef(y_true, y_pred)
```

Clustering Machine Learning Metrics

1) Silhouette Score

Definition: Silhouette Score measures how similar an object is to its own cluster compared to other clusters. It ranges from -1 to 1.

Formula:
$$\frac{1}{n} \sum_{i=1}^n \frac{b_i - a_i}{\max\{a_i, b_i\}}$$

Code Relevance:

```
from sklearn.metrics import silhouette_score

# Calculate Silhouette Score
silhouette_avg = silhouette_score(X, labels)
```

2) Adjusted Rand Index (ARI)

Definition: ARI measures the similarity between true and predicted cluster assignments, adjusted for chance.

Formula:
$$ARI = \frac{\text{Index} - \text{Expected Index}}{\text{Max Index} - \text{Expected Index}}$$

- Index: Index is the raw Rand Index.
- Expected Index: Expected Index is the expected index under a null hypothesis of random cluster assignments.
- Max Index: Max Index is the maximum possible Rand Index.

Code Relevance:

```
from sklearn.metrics import adjusted_rand_score

# Calculate Adjusted Rand Index
ari = adjusted_rand_score(true_labels, predicted_labels)
```

3) Davies-Bouldin Index

Definition: Davies-Bouldin Index measures the compactness and separation between clusters. A lower value indicates better clustering.

Formula:
$$DB = \frac{1}{k} \sum_{i=1}^k \max_{j \neq i} \left(\frac{\text{avg_distance}_i + \text{avg_distance}_j}{\text{distance_between_clusters}} \right)$$

- k is the number of clusters.
- avg_distance_i, avg_distance_j is the average distance from the points in cluster i to the centroid of cluster j.

- distance_between_clustersdistance_between_clusters is the distance between the centroids of clusters i and j.

Code Relevance:

```
from sklearn.metrics import davies_bouldin_score
# Calculate Davies-Bouldin Index
dbi = davies_bouldin_score(X, labels)
```

4) Homogeneity

Definition: Homogeneity measures the extent to which each cluster contains only data points that are members of a single class.

Formula:
$$H = 1 - \frac{H(C|K)}{H(C)}$$

- $H(C|K)$ is the conditional entropy of the true class labels given the cluster assignments.
- $H(C)$ is the entropy of the true class labels.

Code Relevance:

```
from sklearn.metrics import homogeneity_score
# Calculate Homogeneity Score
homogeneity = homogeneity_score(true_labels, predicted_labels)
```

5) Completeness

Definition: Completeness measures the extent to which all data points that are members of a given class are assigned to the same cluster.

Formula:
$$C = 1 - \frac{H(K|C)}{H(K)}$$

- $H(K|C)$ is the conditional entropy of the cluster assignments given the true class labels.
- $H(K)$ is the entropy of the cluster assignments.

Code Relevance:

```
from sklearn.metrics import completeness_score
# Calculate Completeness Score
completeness = completeness_score(true_labels, predicted_labels)
```

6) V-Measure

Definition: V-Measure is the harmonic mean of homogeneity and completeness. It provides a balanced measure that considers both aspects of clustering performance.

Formula:
$$V = \frac{2 \times (H \times C)}{(H + C)}$$

- H is homogeneity.
- C is completeness.

Code Relevance:

```
from sklearn.metrics import v_measure_score
# Calculate V-Measure Score
v_measure = v_measure_score(true_labels, predicted_labels)
```

These metrics provide various perspectives on the performance of a predictive model, considering different aspects of the errors and their magnitudes. The choice of which metric to use depends on the specific characteristics of the data and the goals of the modeling task.