Standard Scaling Normalization

Standard scaling transforms the data by removing the mean and scaling to unit variance, resulting in a distribution with a mean of 0 and a standard deviation of 1.

Formula:

$$\text{New Value} = \frac{\text{Original Value} - \mu}{\sigma}$$

Where:

- μ is the mean of the feature
- σ is the standard deviation of the feature

Algorithm: Standard Scaling Normalization (5–6 steps)

- **Step 1**: Import required libraries (`numpy` and `StandardScaler` from `sklearn.preprocessing`).
- **Step 2**: Input the dataset as a 2D array.
- **Step 3**: Initialize the `StandardScaler` .
- **Step 4**: Fit the scaler on the data and transform it to standardize the values.
- **Step 5**: Store the standardized data.
- **Step 6**: Output the original data and the standardized (scaled) data.

Min-Max Scaling Normalization

Min-Max scaling transforms features by scaling each value to a fixed range, typically [0, 1], based on the minimum and maximum values of each feature.

Formula:

$$\text{New Value} = \frac{\text{Original Value} - \text{Minimum Value}}{\text{Maximum Value} - \text{Minimum Value}}$$

Algorithm: Min-Max Scaling Normalization (5–6 steps)

- **Step 1**: Import required libraries (`numpy` and `MinMaxScaler` from `sklearn.preprocessing`).
- **Step 2**: Input the dataset as a 2D array.
- **Step 3**: Initialize the `MinMaxScaler` .
- **Step 4**: Fit the scaler on the data and transform it to the scaled form.
- **Step 5**: Store the scaled data.
- **Step 6**: Output the original data and the Min-Max scaled data.

Decimal Scaling Normalization

Decimal scaling normalization moves the decimal point of values to bring them into a smaller range, typically between -1 and 1, by dividing each value by a power of 10 based on the maximum absolute value.

Formula:

$$\text{New Value} = \frac{\text{Original Value}}{10^j}$$

where
$j = \lceil \log_{10}(\max(|\text{value}|)) \rceil$
and $\lceil \rceil$ denotes the ceiling function (rounding up).

Algorithm: Decimal Scaling Normalization (5–6 steps)

- **Step 1**: Import required libraries (`numpy` and `math`).
- **Step 2**: Input the dataset as a 2D array.
- **Step 3**: Copy the original data to preserve it.
- **Step 4**: For each column, find the maximum absolute value.
- **Step 5**: Calculate j as the ceiling of $\log_{10}$ of the maximum absolute value, and divide each column value by 10^j.
- **Step 6**: Output the original data and the normalized data.

Agglomerative Clustering

Agglomerative Clustering is a bottom-up hierarchical clustering technique where each data point starts in its own cluster, and pairs of clusters are merged step-by-step based on their similarity until the desired number of clusters is achieved.

Algorithm: Agglomerative Clustering (5–6 steps)

- **Step 1**: Import required libraries (`numpy` , `AgglomerativeClustering` , `make_blobs` , and `matplotlib.pyplot`).
- **Step 2**: Generate synthetic data using `make_blobs()` with specified samples, features, and cluster centers.
- **Step 3**: Initialize the `AgglomerativeClustering` model by specifying the number of clusters.
- **Step 4**: Fit the model to the data and predict cluster labels for each data point.
- **Step 5**: Visualize the clusters using different colors in a scatter plot.
- **Step 6**: Print the assigned cluster labels for all data points.

Apriori Algorithm

The Apriori algorithm identifies frequent itemsets in a transaction dataset by exploring larger and larger sets of items, using the principle that any subset of a frequent itemset must also be frequent.

Algorithm: Apriori (5–6 steps)

- **Step 1**: Import necessary libraries and define the list of transactions.
- **Step 2**: Set minimum support and compute minimum support count.
- **Step 3**: List all unique items and initialize frequent 1-itemsets based on support.
- **Step 4**: Define functions to calculate support count and generate candidate itemsets.
- **Step 5**: Iteratively generate larger frequent itemsets until no more candidates meet the minimum support.
- **Step 6**: Print all frequent itemsets with their support counts and percentages.

Brute Force Algorithm

The Brute Force method for finding frequent itemsets generates all possible item combinations and checks their support against a minimum threshold, without using any pruning strategies.

Algorithm: Brute Force (5–6 steps)

- **Step 1**: Import necessary libraries and define the list of transactions.
- **Step 2**: Set minimum support and calculate the minimum support count.
- **Step 3**: Extract all unique items from the transactions.
- **Step 4**: For each possible itemset size, generate all candidate combinations using `itertools.combinations()`.
- **Step 5**: Calculate the support count for each candidate and identify frequent itemsets that meet the minimum support.
- **Step 6**: Print all frequent itemsets grouped by their size.

FP-Growth:

FP-Growth (Frequent Pattern Growth) is an efficient algorithm used to mine frequent itemsets in large datasets, typically used for market basket analysis. Unlike the Apriori algorithm, FP-Growth does not generate candidate itemsets and instead uses a compact tree structure (FP-tree) to represent the database and directly mines the frequent itemsets.

Algorithm: FP-Growth (5–6 steps)

Step 1: Import necessary libraries (pandas, TransactionEncoder from mlxtend.preprocessing, and fpgrowth from mlxtend.frequent_patterns).

Step 2: Input the transaction data (list of transactions, each being a list of items).

Step 3: Set the minimum support threshold and calculate the relative minimum support.

Step 4: Convert the transaction data into a binary matrix using the TransactionEncoder.

Step 5: Apply the FP-Growth algorithm using the `fpgrowth()` function, passing in the binary matrix and the minimum support.

Step 6: Output the frequent itemsets discovered by FP-Growth.

K-Means Clustering

K-Means is an unsupervised machine learning algorithm used to partition data into `k` clusters, where each cluster is represented by its centroid. The algorithm iteratively assigns data points to the nearest centroid and updates the centroids based on the mean of the points in each cluster.

Algorithm: K-Means Clustering (5–6 steps)

- **Step 1**: Import necessary libraries (`numpy`, `KMeans`, `make_blobs`, `matplotlib`).
- **Step 2**: Generate sample data using `make_blobs()` to create data with specified features and clusters.
- **Step 3**: Initialize the `KMeans` model with the desired number of clusters (`n_clusters`) and fit it to the data using the `fit()` method.
- **Step 4**: Retrieve the cluster labels (`labels_`) and cluster centers (`cluster_centers_`).
- **Step 5**: Visualize the clusters by plotting the data points and centroids using `matplotlib`.
- **Step 6**: Output the cluster centers.

Logistic Regression

Logistic Regression is a statistical model used for binary classification tasks, which predicts the probability of a certain class or event occurring, based on input features.

Algorithm: Logistic Regression (5–6 steps)

- **Step 1**: Import necessary libraries (`numpy`, `train_test_split`, `LogisticRegression`, `accuracy_score`, and `classification_report`).
- **Step 2**: Input the feature data (`X`) and labels (`y`).
- **Step 3**: Split the data into training and testing sets using `train_test_split()`.
- **Step 4**: Initialize the `LogisticRegression` model and train it using the training data (`fit()` method).
- **Step 5**: Predict the labels for the test set and evaluate the model's performance using accuracy and classification report.
- **Step 6**: Output the accuracy and the classification report.

ECLAT (Equivalence Class Clustering and Apriori-Tidsets)

ECLAT is an algorithm used for mining frequent itemsets. It utilizes a depth-first search approach and employs tidsets (transaction ids) to efficiently find frequent itemsets.

Algorithm: ECLAT (5–6 steps)

- **Step 1**: Import required libraries (`defaultdict` from `collections`, `combinations` from `itertools`).
- **Step 2**: Define a function `run_eclat()` that takes transactions (`txns`) and a minimum support count (`min_sup_cnt`).
- **Step 3**: Build a dictionary (`i_tids`) that stores the transaction ids for each item, and filter out items that appear less than the minimum support count.
- **Step 4**: Initialize the current frequent itemsets (`curr_k_sets`) with single items that meet the support threshold.
- **Step 5**: Iteratively combine frequent itemsets of length `k` to generate frequent itemsets of length `k+1` by intersecting the transaction ids (tidsets) of each itemset pair.
- **Step 6**: Return all frequent itemsets once no further combinations meet the minimum support count.

Naive Bayes Classification

Naive Bayes is a probabilistic classifier based on applying Bayes' theorem with strong (naive) independence assumptions between the features.

Algorithm: Naive Bayes (5–6 steps)

- **Step 1**: Import necessary libraries (`numpy`, `train_test_split`, `GaussianNB`, `accuracy_score`, `classification_report`).
- **Step 2**: Input the feature data (`X`) and labels (`y`).
- **Step 3**: Split the data into training and testing sets using `train_test_split()`.
- **Step 4**: Initialize the `GaussianNB` model and train it using the training data.
- **Step 5**: Predict the labels for the test set and evaluate the model's performance using accuracy and classification report.
- **Step 6**: Output the accuracy and the classification report.

Decision Tree Classification

A Decision Tree is a supervised machine learning algorithm used for classification tasks, which recursively splits the data based on feature values to form a tree structure.

Algorithm: Decision Tree (5–6 steps)

- **Step 1**: Import necessary libraries (`numpy`, `train_test_split`, `DecisionTreeClassifier`, `export_graphviz`, `accuracy_score`, and `graphviz`).
- **Step 2**: Input the feature data (`X`) and target labels (`y`), along with feature names and class names.
- **Step 3**: Split the data into training and testing sets using `train_test_split()`.
- **Step 4**: Initialize the `DecisionTreeClassifier`, train it with the training data (`fit()` method).
- **Step 5**: Predict the labels for the test set, evaluate the accuracy of the model using `accuracy_score()`.
- **Step 6**: Export and visualize the decision tree using `export_graphviz()` and `graphviz.Source()`. Save the tree visualization as a `.pdf` file.

1. Standard GROUP BY Operation

The `GROUP BY` clause groups the rows that have the same values into summary rows, like total sales for each region-product combination.

Query:

```sql
SELECT
    Region,
    Product,
    SUM(SalesAmount) AS TotalSales
FROM
    SalesData
GROUP BY
    Region,
    Product
ORDER BY
    Region, Product;
```

2. ROLLUP Operation

`ROLLUP` is an extension to the `GROUP BY` clause that provides subtotals and grand totals for hierarchical groupings. It adds summary rows for each combination of grouped columns and the grand total.

Query:

```sql
SELECT
    Region,
    Product,
    SUM(SalesAmount) AS TotalSales
FROM
    SalesData
GROUP BY
    Region,
    Product
WITH ROLLUP;
```

3. CUBE Operation (Emulated using UNION ALL)

`CUBE` generates a result set that includes all possible combinations of the specified columns (cross-tabulation). Since MySQL lacks native support for `CUBE`, this operation is emulated using `UNION ALL`.

Query:

```sql
SELECT Region, Product, SUM(SalesAmount) AS TotalSales FROM SalesData GROUP BY Region, Product
UNION ALL
SELECT Region, NULL, SUM(SalesAmount) AS TotalSales FROM SalesData GROUP BY Region
UNION ALL
SELECT NULL, Product, SUM(SalesAmount) AS TotalSales FROM SalesData GROUP BY Product
UNION ALL
SELECT NULL, NULL, SUM(SalesAmount) AS TotalSales FROM SalesData
ORDER BY Region IS NULL, Region, Product IS NULL, Product;
```

4. GROUPING SETS Operation

`GROUPING SETS` allows you to specify multiple groupings in a single query. It can be used to calculate totals for multiple combinations of columns.

Query:

```sql
SELECT
    Region,
    Product,
    Year,
    SUM(SalesAmount) AS TotalSales
FROM
    SalesData
GROUP BY
    GROUPING SETS (
        (Region, Product),
        (Region, Year),
        (Product),
        () -- Grand Total
    )
ORDER BY
    Region IS NULL, Region, Product IS NULL, Product, Year IS NULL, Year;
```

Fill Missing Values with Region-Specific Median:

This algorithm fills the missing numeric values in a DataFrame with the median of that column, grouped by a specific categorical column (in this case, `region`). It helps to impute missing values while considering the region-specific distribution.

Algorithm: Fill Missing Values with Region-Specific Median (5–6 steps)

Step 1: Import the pandas library to handle the DataFrame.

Step 2: Create the initial DataFrame with numeric and categorical columns, including some missing values (NaN).

Step 3: Define a function `fill_m()` that accepts the DataFrame and the name of the column used for grouping (in this case, `region`).

Step 4: Inside the function, identify the numeric columns in the DataFrame and calculate the median values for each of those columns, grouped by the `region`.

Step 5: Iterate over each numeric column and region, and replace missing values (NaN) in the numeric columns with the corresponding region-specific median.

Step 6: Return the modified DataFrame with missing values filled.