Engineering Task1 and Engineering Task2

In [1]:
```
import pandas as pd
import numpy as np
import random
# Load the dataset
data = pd.read_csv('DSet.csv')
```

In [2]:
```
#Feature Engineering Task 1

def Engineering_Task1(data):
    # Iterate through each column
    for col in data.columns:
        # Check if the column contains missing values
        if data[col].isnull().sum() > 0:
            # If the column is categorical, impute with the most frequent value
            if data[col].dtype == 'object':
                data[col].fillna(data[col].value_counts().index[0], inplace=True)

        # If the column is numerical, impute with the mean value
            else:
                data[col].fillna(data[col].mean(), inplace=True)

# Save the imputed dataset
    data.to_csv('imputed_data.csv', index=False)
```

In [3]:
```
#Feature Engineering Task 2

def Engineering_Task2(data):
    # Iterate through each column
    for col in data.columns:
        # Calculate the mean and standard deviation of the column
        if data[col].dtype != 'object' and col!='id':
            mean = data[col].mean()
            std = data[col].std()
            # Normalize the column using the formula (X' = (X - μ) / σ)
            data[col] = (data[col] - mean) / std


    # Save the normalized dataset
    data.to_csv('normalized_data.csv', index=False)
    # print(data)
```

The function **Engineering_Task1** is a function that takes a dataset as input and performs missing value imputation on it. The algorithm iterates through each column of the dataset, checks if the column contains any missing values, and if so, imputes the missing values with the most frequent value for categorical columns or the mean value for numerical columns.
Here is the detailed explanation of the algorithm:

1.For each column in the dataset, check if it contains any missing values by calling the isnull() method on the column and summing the resulting boolean array.

2.If the column contains missing values, check if it is categorical or numerical by checking the dtype attribute of the column.

3.If the column is categorical, impute the missing values with the most frequent value by calling the value_counts() method on the column and selecting the index of the first value in the resulting Series.

4.If the column is numeric, impute the missing values with the mean value by calling the mean() method on the column.

The function **Engineering_Task2** is a function that takes a dataset as input and performs normalization on its numerical columns. The algorithm iterates through each numerical column of the dataset (excluding the 'id' column), calculates the mean and standard deviation of the column, and then applies the normalization formula ($X' = (X - \mu) / \sigma$) to each value in the column. Finally, the function saves the normalized dataset to a CSV file named "normalized_data.csv" and prints it to the console.

Here is the detailed explanation of the algorithm:

1.For each numerical column in the dataset (excluding the 'id' column), calculate its mean and standard deviation by calling the mean() and std() methods on the column, respectively.

2.Normalize the column by subtracting the mean from each value in the column and dividing the result by the standard deviation.

3.Replace the original values in the column with the normalized values.

Part A - Perceptron Learning Algorithm

In [4]:
```python
# Define the Perceptron class
class Perceptron:

    def __init__(self, learning_rate=0.01, n_iterations=1000):
        self.learning_rate = learning_rate
        self.n_iterations = n_iterations

    def fit(self, X, y):
        n_samples, n_features = X.shape
        self.weights = np.zeros(n_features)
        self.bias = 0
        for i in range(self.n_iterations):
            for j in range(n_samples):
                if y[j] * (np.dot(X[j], self.weights) + self.bias) <= 0:
                    self.weights += self.learning_rate * y[j] * X[j]
                    self.bias += self.learning_rate * y[j]


    def predict(self, X):
        linear_output = np.dot(X, self.weights) + self.bias
        return np.where(linear_output >= 0, 1, -1)
```

The Perceptron class is a Python class that implements the Perceptron algorithm, a binary classification algorithm that learns a linear decision boundary to separate two classes. The class has two methods: fit() and predict(). The fit() method trains the perceptron on a given training set, while the predict() method uses the trained model to predict the class labels of a given test set.

Here is the detailed explanation of the algorithm:

1.Initialize the Perceptron class with a learning rate (learning_rate) and number of iterations (n_iterations).

2.The fit() method takes as input the training set X and their corresponding labels y.

3.The dimensions of the training set are obtained by calling the shape attribute of X.

4.Initialize the weights (weights) to an array of zeros with dimensions equal to the number of features in the training set.

5.Initialize the bias (bias) to zero.

6.Train the model for n_iterations epochs by iterating over the samples in X and their corresponding labels in y.

7.If the predicted label of the sample X[j] is not equal to the true label y[j], update the weights and bias using the perceptron update rule:

weights = weights + learning_rate * y[j] * X[j]
bias = bias + learning_rate * y[j]

8.The predict() method takes as input a test set X.

9.Compute the linear output of the model by taking the dot product of X and the trained weights (weights) and adding the bias (bias).

10.Return a binary array of the predicted labels, where values greater than or equal to 0 are classified as one class and values less than 0 are classified as the other class.

Classifier (PM1)

```
#Perceptron Model - PM1

# Split the dataset into features and labels
Engineering_Task1(data)
X = data.drop(['diagnosis','id'], axis=1).values
y = data['diagnosis'].values

y[y=='M']=-1
y[y=='B']=1
# assume X and y are the feature matrix and target vector, respectively

# shuffle the indices of the data
indices = list(range(len(X)))
random.shuffle(indices)

# calculate the split point
split_idx = int(0.67 * len(X))


# split the data into training and testing sets
train_X = X[indices[:split_idx]]
train_y = y[indices[:split_idx]]
test_X = X[indices[split_idx:]]
test_y = y[indices[split_idx:]]


# Create an instance of the Perceptron class
perceptron = Perceptron()

# Train the Perceptron model
perceptron.fit(train_X,train_y)

# Predict the labels for the training data
y_pred = perceptron.predict(test_X)

# Calculate the accuracy of the model
accuracy = np.mean(y_pred == test_y)

# Print the accuracy of the model
print('PM1 Accuracy:', accuracy)
```

```
PM1 Accuracy: 0.898936170212766
```

This code implements a Perceptron model (PM1) for binary classification of a breast cancer dataset. The dataset is split into training and testing sets using a 67:33 ratio, with the training data used to fit the Perceptron model and the testing data used to evaluate the model's accuracy.

Here's a step-by-step breakdown of the algorithm:

1) Load the breast cancer dataset and split it into features (X) and labels (y).
2) Convert the labels to -1 (for malignant) and 1 (for benign).
3) Shuffle the indices of the data.
4) Calculate the split point (index) for splitting the data into training and testing sets.
5) Split the data into training and testing sets using the calculated split point.
6) Create an instance of the Perceptron class (presumably defined elsewhere in the code).
7) Train the Perceptron model using the training data (train_X, train_y).
8) Predict the labels for the testing data (test_X).
9) Calculate the accuracy of the model by comparing the predicted labels to the actual labels (test_y).
10) Print the accuracy of the model.

Classifier (PM2) - Changing the order of the training examples

```
#Perceptron Model - PM2

# shuffle the indices of the data
indices = list(range(len(X)))
random.shuffle(indices)

# calculate the split point
split_idx = int(0.67 * len(X))

# split the data into training and testing sets
train_X_2 = X[indices[:split_idx]]
train_y_2 = y[indices[:split_idx]]
test_X_2 = X[indices[split_idx:]]
test_y_2 = y[indices[split_idx:]]
# Train the Perceptron model
perceptron.fit(train_X_2,train_y_2)

# Predict the labels for the training data
y_pred = perceptron.predict(test_X_2)

# Calculate the accuracy of the model
accuracy = np.mean(y_pred == test_y_2)

# Print the accuracy of the model
print('PM2 Accuracy:', accuracy)
```

```
PM2 Accuracy: 0.8936170212765957
```

This code implements another Perceptron model (PM2) for binary classification of the same breast cancer dataset as in PM1. The key difference is that it uses a different random shuffle of the data to split it into training and testing sets.

Here's a step-by-step breakdown of the algorithm:

1) Shuffle the indices of the data.
2) Calculate the split point (index) for splitting the data into training and testing sets using a 67:33 ratio.
3) Split the data into training and testing sets using the calculated split point.
4) Train the Perceptron model using the training data (train_X_2, train_y_2).
5) Predict the labels for the testing data (test_X_2).
6) Calculate the accuracy of the model by comparing the predicted labels to the actual labels (test_y_2).
7) Print the accuracy of the model.

Learning Task 2:
PM3 - perceptron algorithm on the normalized data

```
#Perceptron Model - PM3

Engineering_Task2(data)
X = data.drop(['diagnosis','id'], axis=1).values
y = data['diagnosis'].values
# print(X)
y[y=='M']=-1
y[y=='B']=1
# assume X and y are the feature matrix and target vector, respectively
# print(data)
# shuffle the indices of the data
indices = list(range(len(X)))
random.shuffle(indices)

# calculate the split point
split_idx = int(0.67 * len(X))

indices = list(range(len(y)))
random.shuffle(indices)
# print('new data')
# print(data)


# split the data into training and testing sets
train_X = X[indices[:split_idx]]
train_y = y[indices[:split_idx]]
test_X = X[indices[split_idx:]]
test_y = y[indices[split_idx:]]


# Create an instance of the Perceptron class
perceptron = Perceptron()

# Train the Perceptron model
perceptron.fit(train_X,train_y)

# Predict the labels for the training data
y_pred = perceptron.predict(test_X)

# Calculate the accuracy of the model
accuracy = np.mean(y_pred == test_y)

# Print the accuracy of the model
print('PM3 Accuracy:', accuracy)
```

```
PM3 Accuracy: 0.9680851063829787
```

This code implements another Perceptron model (PM3) for binary classification of the same breast cancer dataset as in PM1 and PM2. The key difference is that it shuffles the indices of both the feature matrix and the target vector separately, and then splits the data into training and testing sets using the shuffled indices.

Here's a step-by-step breakdown of the algorithm:

1) Split the dataset into features (X) and labels (y).
2) Convert the labels from 'M' and 'B' to -1 and 1, respectively.
3) Shuffle the indices of the feature matrix (X).
4) Calculate the split point (index) for splitting the data into training and testing sets using a 67:33 ratio.
5) Shuffle the indices of the target vector (y).
6) Split the data into training and testing sets using the shuffled indices.
7) Create an instance of the Perceptron class.
8) Train the Perceptron model using the training data (train_X, train_y).
9) Predict the labels for the testing data (test_X).
10) Calculate the accuracy of the model by comparing the predicted labels to the actual labels (test_y).
11) Print the accuracy of the model.

Learning Task 3:
PM4 : Changing the order of features in the dataset randomly

```
#Perceptron Model - PM4

# changing order of features
X_permuted=X[:,np.random.permutation(X.shape[1])]

# now X_permuted_fixed has the same order of permutation for every row
train_X_permuted =X_permuted[indices[:split_idx]]
train_y_permuted =y[indices[:split_idx]]
test_X_permuted= X_permuted[indices[split_idx:]]
test_y_permuted= y[indices[split_idx:]]

# Train the Perceptron model
perceptron.fit(train_X_permuted,train_y_permuted)

# Predict the labels for the training data
y_pred = perceptron.predict(test_X_permuted)

# Calculate the accuracy of the model
accuracy = np.mean(y_pred == test_y_permuted)

# Print the accuracy of the model
print('PM4 Accuracy:', accuracy)
```

```
PM4 Accuracy: 0.9680851063829787
```

This code is implementing a Perceptron model, similar to the previous examples (PM1, PM2, and PM3), but with the additional step of permuting the order of the features in the dataset. The purpose of this is to test how the model performs when the features are not in their original order.

Here is a detailed algorithm for this code:

1) Load the dataset, drop the 'id' and 'diagnosis' columns, and assign the remaining columns to a variable 'X'. Assign the 'diagnosis' column to a variable 'y'.
2) Convert the 'M' and 'B' labels in 'y' to -1 and 1, respectively.
3) Shuffle the indices of 'X' and 'y' using the 'random.shuffle()' function.
4) Calculate the split index as 67% of the total length of 'X'.
5) Create a new variable 'X_permuted' by permuting the order of the columns in 'X' using the 'np.random.permutation()' function.

6) Using the shuffled indices, split 'X_permuted' and 'y' into training and testing sets, assigning them to variables 'train_X_permuted', 'train_y_permuted', 'test_X_permuted', and 'test_y_permuted'.
7) Create an instance of the Perceptron class.
8) Train the Perceptron model using the training set 'train_X_permuted' and 'train_y_permuted'.
9) Predict the labels for the testing set 'test_X_permuted'.
10) Calculate the accuracy of the model by comparing the predicted labels with the true labels in 'test_y_permuted'.
11) Print the accuracy of the model.