```
"""
Algorithm for outlier detection
"""
# Import necessary libraries
from sklearn.ensemble import IsolationForest
from sklearn.impute import SimpleImputer
import pandas as pd

def remove_outliers(data, contamination=0.05, random_state=None):
    """
    Remove outliers from a dataset using the Isolation Forest algorithm.

    Parameters:
    - data (pd.DataFrame): Input DataFrame containing the dataset.
    - contamination (float): The proportion of outliers in the dataset. Default is 0.05 (5%).
    - random_state (int or None): Random seed for reproducibility. Default is None.

    Returns:
    - pd.DataFrame: DataFrame with outliers removed (inliers).
    - pd.DataFrame: DataFrame with the removed outliers.
    """
    # Replace missing values with the mean
    imputer = SimpleImputer(strategy='mean')
    data_imputed = pd.DataFrame(imputer.fit_transform(data), columns=data.columns)

    # Separate target variable (y) and features (X)
    y_column = 'case'  # Adjust with the actual column name for your target variable
    X = data_imputed.drop(y_column, axis=1)
    y = data_imputed[y_column]

    # Create an Isolation Forest model
    iso_forest = IsolationForest(contamination=contamination, random_state=random_state)

    # Fit the model and predict outliers
    outlier_preds = iso_forest.fit_predict(X)

    # Identify outliers and inliers
    outliers = data[outlier_preds == -1]
    inliers = data[outlier_preds == 1]

    # Return both the DataFrame with outliers removed (inliers) and the removed outliers
    return inliers, outliers
```

```
"""
Function to split the data into training, cv and test sets
"""
from sklearn.model_selection import train_test_split

def split_data(data, test_size=0.2, cv_size=0.2, random_state=None):
    """
    Split the dataset into training, cross-validation, and test sets.
```

```
    Parameters:
    - data (pd.DataFrame): Input DataFrame containing the dataset.
    - test_size (float): Proportion of the dataset to include in the test split.
    - cv_size (float): Proportion of the dataset to include in the cross-validation split.
    - random_state (int or None): Random seed for reproducibility.

    Returns:
    - tuple: (train_data, cv_data, test_data)
    """
    # Split into training and temporary set
    train_data, temp_data = train_test_split(data, test_size=(test_size + cv_size), random_state=random_state)

    # Split the temporary set into CV and test sets
    cv_data, test_data = train_test_split(temp_data, test_size=cv_size/(test_size + cv_size),
random_state=random_state)

    return train_data, cv_data, test_data
```

```
"""
Algorithm for feature scaling
"""
# Import necessary libraries
import pandas as pd

def normalize_data(data):
    """
    Perform Z-score normalization on a pandas DataFrame.

    Parameters:
    - data (pd.DataFrame): Input DataFrame containing the dataset.

    Returns:
    - pd.DataFrame: DataFrame with features Z-score normalized.
    """

    # Ensure the input is a DataFrame
    if not isinstance(data, pd.DataFrame):
        raise ValueError("Input must be a pandas DataFrame.")

    # Separate target variable (y) and features (X)
    y_column = 'case'  # Target variable
    X = data.drop(y_column, axis=1)
    y = data[y_column]

    # Calculate mean and standard deviation for each column
    mean_values = X.mean()
    std_values = X.std()

    # Z-score normalize each column
    normalized_X = (X - mean_values) / std_values
```

```
    # Add target variable back to the DataFrame
    normalized_data = pd.concat([y, normalized_X], axis=1)

    return normalized_data
```

```
"""
Notebook for preprocessing the data
"""

import pandas as pd
import openpyxl
import sys
sys.path.append('../../')
from src.data_preprocessing.isolation_forest import remove_outliers
from src.data_preprocessing.train_test_splitter import split_data
from src.data_preprocessing.z_score_normalization import normalize_data

# Load data from Excel file
df = pd.read_excel('../../data/raw/raw_data.xlsx')

# Display the first few rows of the dataframe to inspect the data
df.head()

# Separate target variable (y) and features (x)
y_column = 'case'  # Column name
X_columns = ['AGE', 'BMI', 'density']  # Column names

y = df[y_column] # Seperate target variable y, and features X from dataframe
X = df[X_columns]

# Apply train-test split
data = pd.concat([y, X], axis=1)  # Concatenate X and y to one dataframe

train_data, cv_data, test_data = split_data(data)

# Test number of examples in train, cv and test sets
print("Train data number of examples:")
print(len(train_data))
print("CV data number of examples:")
print(len(cv_data))
print("Test data number of examples:")
print(len(test_data))

# Print the first few rows of the train data to inspect the data
train_data.head()

# Apply Isolation Forest to detect outliers and remove them from each set (save removed data in a separate
dataframe)
train_data, removed_data1 = remove_outliers(train_data)
cv_data, removed_data2 = remove_outliers(cv_data)
```

```
test_data, removed_data3 = remove_outliers(test_data)
removed_data = pd.concat([removed_data1, removed_data2, removed_data3])

# Print number of outliers in each set
print("Train data number of outliers:")
print(len(removed_data1))
print("CV data number of outliers:")
print(len(removed_data2))
print("Test data number of outliers:")
print(len(removed_data3))

# Apply Z-score normalization to scale the features for each set
train_data = normalize_data(train_data)
cv_data = normalize_data(cv_data)
test_data = normalize_data(test_data)
# Print first few rows of normalized data to inspect the data
train_data.head()

# Save the preprocessed data to respective Excel files and the removed outliers to another Excel file
train_data.to_excel('train_data.xlsx', index=False)
cv_data.to_excel('cv_data.xlsx', index=False)
test_data.to_excel('test_data.xlsx', index=False)
removed_data.to_excel('removed_data.xlsx', index=False)
```

```
"""
Notebook for logistic regression
"""

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.impute import SimpleImputer
import copy


train_data = pd.read_excel('../../data/preprocessed/train_data.xlsx')
cv_data = pd.read_excel('../../data/preprocessed/cv_data.xlsx')
test_data = pd.read_excel('../../data/preprocessed/test_data.xlsx')

# Impute missing values with mean
imputer = SimpleImputer(strategy='mean')
train_data = imputer.fit_transform(train_data)
cv_data = imputer.transform(cv_data)
test_data = imputer.transform(test_data)

# split into X and y
X_train = train_data[:, 1:]
y_train = train_data[:, 0]
X_cv = cv_data[:, 1:]
y_cv = cv_data[:, 0]
X_test = test_data[:, 1:]
```

```
y_test = test_data[:, 0]



def compute_cost_logistic(X, y, w, b):
    m = len(y)
    h_theta_x = 1 / (1 + np.exp(-(np.dot(X, w) + b)))
    cost = -np.sum(y * np.log(h_theta_x) + (1 - y) * np.log(1 - h_theta_x)) / m
    return cost



def sigmoid(z):
    return 1/(1+np.exp(-z))

def compute_gradient_logistic(X, y, w, b):
    """
    Computes the gradient for logistic regression

    Args:
      X (ndarray (m,n): Data, m examples with n features
      y (ndarray (m,)): target values
      w (ndarray (n,)): model parameters
      b (scalar)      : model parameter
    Returns
      dj_dw (ndarray (n,)): The gradient of the cost w.r.t. the parameters w.
      dj_db (scalar)      : The gradient of the cost w.r.t. the parameter b.
    """
    m,n = X.shape
    dj_dw = np.zeros((n,))                           #(n,)
    dj_db = 0.

    for i in range(m):
        f_wb_i = sigmoid(np.dot(X[i],w) + b)          #(n,)(n,)=scalar
        err_i  = f_wb_i  - y[i]                       #scalar
        for j in range(n):
            dj_dw[j] = dj_dw[j] + err_i * X[i,j]      #scalar
        dj_db = dj_db + err_i
    dj_dw = dj_dw/m                                   #(n,)
    dj_db = dj_db/m                                   #scalar

    return dj_db, dj_dw

def gradient_descent(X, y, w_in, b_in, alpha, num_iters, cost_threshold):
    """
    Performs batch gradient descent

    Args:
      X (ndarray (m,n)   : Data, m examples with n features
      y (ndarray (m,))   : target values
      w_in (ndarray (n,)): Initial values of model parameters
      b_in (scalar)      : Initial values of model parameter
      alpha (float)      : Learning rate
```

```
      num_iters (scalar) : number of iterations to run gradient descent

    Returns:
      w (ndarray (n,))   : Updated values of parameters
      b (scalar)         : Updated value of parameter
    """
    # An array to store cost J and w's at each iteration primarily for graphing later
    J_history = []
    w = copy.deepcopy(w_in)  #avoid modifying global w within function
    b = b_in

    for i in range(num_iters):
        # Calculate the gradient and update the parameters
        dj_db, dj_dw = compute_gradient_logistic(X, y, w, b)

        # Update Parameters using w, b, alpha and gradient
        w = w - alpha * dj_dw
        b = b - alpha * dj_db

        # Save cost J at each iteration
        if i<100000:      # prevent resource exhaustion
            J_history.append( compute_cost_logistic(X, y, w, b) )

        # Print cost
        print(f"Iteration {i:4d}: Cost {J_history[-1]}   ")

        # Check for convergence
        if i!= 0 and abs(J_history[i-1] - J_history[i]) < cost_threshold:
            print(f"Converged at iteration {i}. Change in cost: {abs(J_history[i-1] - J_history[i])}")
            break

    return w, b, J_history         #return final w,b and J history for graphing

w_tmp  = np.zeros_like(X_train[0])
b_tmp  = 0.
alph = 1
iters = 10000
cost_threshold = 0.001

w_out, b_out, _ = gradient_descent(X_train, y_train, w_tmp, b_tmp, alph, iters, cost_threshold)
print(f"\nupdated parameters: w:{w_out}, b:{b_out}")
iteration_costs = [
    (0, 0.5631982531155912),
    (1, 0.4889821652087534),
    (2, 0.4448418165195707),
    (3, 0.4173401510710706),
    (4, 0.3994739437450503),
    (5, 0.38745365669543164),
    (6, 0.37913017802116894),
    (7, 0.3732284313326753),
    (8, 0.36896090887815275),
```

```
    (9, 0.3658240269162005),
    (10, 0.3634860093323348),
    (11, 0.36172260874619055),
    (12, 0.36037888690021175),
    (13, 0.35934574207421766),
    (14, 0.3585450742616799)
]

# Extract iteration numbers and cost values
iteration_numbers, cost_values = zip(*iteration_costs)

# Plot
plt.plot(iteration_numbers, cost_values, marker='o')
plt.title('Gradient Descent Convergence')
plt.xlabel('Iterations')
plt.ylabel('Cost')
plt.show()

# Evaluate performance on cv and test data
# Create predictions for cv and test data
def predict(X, w, b):
    """
    Predicts the class for each data point in X

    Args:
      X (ndarray (m,n)): Data, m examples with n features
      w (ndarray (n,)): model parameters
      b (scalar)      : model parameter
    Returns
      ndarray (m,): The predicted class for each example
    """
    m = X.shape[0]
    y_pred = np.zeros((m,))
    for i in range(m):
        y_pred[i] = 1 if sigmoid(np.dot(X[i], w) + b) >= 0.5 else 0
    return y_pred

# Compare predictions with ground truth to compute accuracy
def accuracy(y_true, y_pred):
    """
    Computes the accuracy of the predictions

    Args:
      y_true (ndarray (m,)): The true class labels
      y_pred (ndarray (m,)): The predicted class labels
    Returns
      float: The accuracy
    """
    return np.sum(y_true == y_pred) / len(y_true)

# Calculate accuracy on train, cv and test data
```

```
y_train_pred = predict(X_train, w_out, b_out)
y_cv_pred = predict(X_cv, w_out, b_out)
y_test_pred = predict(X_test, w_out, b_out)
print(f"Accuracy on train data: {accuracy(y_train, y_train_pred):.2%}")
print(f"Accuracy on cv data: {accuracy(y_cv, y_cv_pred):.2%}")
print(f"Accuracy on test data: {accuracy(y_test, y_test_pred):.2%}")

# Plot the coefficients
plt.bar(range(len(w_out)), w_out)
plt.title('Coefficients')
plt.xlabel('Coefficient index')
plt.ylabel('Coefficient value')
plt.show()
```

```
"""
Notebook for random forest
"""

from sklearn.ensemble import RandomForestClassifier
from sklearn.impute import SimpleImputer
from sklearn.metrics import accuracy_score
import pandas as pd

# Access the training, cv and test data
train_data = pd.read_excel('../../data/preprocessed/train_data.xlsx')
cv_data = pd.read_excel('../../data/preprocessed/cv_data.xlsx')
test_data = pd.read_excel('../../data/preprocessed/test_data.xlsx')

# Impute missing values with mean
imputer = SimpleImputer(strategy='mean')
train_data = imputer.fit_transform(train_data)
cv_data = imputer.transform(cv_data)
test_data = imputer.transform(test_data)

# split into X and y
X_train = train_data[:, 1:]
y_train = train_data[:, 0]
X_cv = cv_data[:, 1:]
y_cv = cv_data[:, 0]
X_test = test_data[:, 1:]
y_test = test_data[:, 0]

# Initialize the Random Forest model
rf_model = RandomForestClassifier(n_estimators=100, random_state=42)

# Train the model using the training set
rf_model.fit(X_train, y_train)

# Make predictions on the train set
y_train_pred = rf_model.predict(X_train)
```

```python
# Evaluate the model on the train set
print('Train accuracy score: ', accuracy_score(y_train, y_train_pred))


# Make predictions on the validation set
cv_predictions = rf_model.predict(X_cv)

# Evaluate the model on the validation set
cv_accuracy = accuracy_score(y_cv, cv_predictions)
print(f"Validation Accuracy: {cv_accuracy}")

# Make predictions on the test set
test_predictions = rf_model.predict(X_test)

# Evaluate the model on the test set
test_accuracy = accuracy_score(y_test, test_predictions)
print(f"Test Accuracy: {test_accuracy}")


# Feature importance plot
import matplotlib.pyplot as plt
importances = rf_model.feature_importances_

plt.figure(figsize=(10, 6))
plt.bar(range(len(importances)), importances, align="center")
plt.xticks(range(len(importances)), range(len(importances)))
plt.xlabel("Feature Index")
plt.ylabel("Feature Importance")
plt.title("Random Forest Feature Importance")
plt.show()


# Confusion matrix plot
from sklearn.metrics import confusion_matrix
import seaborn as sns

cm = confusion_matrix(y_train, y_train_pred)
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt="d")
plt.title("Confusion Matrix")
plt.xlabel("Predicted Label")
plt.ylabel("True Label")
plt.show()
```