```python
########################################
#                                      #
#               utils.py               #
#                                      #
########################################
import pandas as pd
import numpy as np
from sklearn.compose import ColumnTransformer
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import OrdinalEncoder, StandardScaler, OneHotEncoder
from sklearn.decomposition import PCA
from sklearn.model_selection import train_test_split, GridSearchCV, cross_val_score
import itertools
from sklearn.metrics import f1_score

def clean_noisy_data(dataset,classes = 2):
    """
    @author : Yassir BENDOU
    Clean the data with replacing the miss-filled values by the correct ones and defining the right types for the dataset
    variables.

    Args:
        :param dataset: The dataset to clean.
        :type  dataset: pandas.DataFrame

    Returns:
        :param dataset: Cleaned dataset.
        :type dataset: pandas.DataFrame
    """

    if 'id' in dataset.columns :
        dataset = dataset.drop(columns = ['id']) # Drop id column as it's not relevent for predictions


    for c in dataset.columns :
        if dataset[c].dtype not in ['float32','float64','float','int32','int64','int']:
            dataset[c] = dataset[c].str.replace('\t','')
            dataset[c] = dataset[c].replace('?',np.nan)
            dataset[c] = dataset[c].str.replace(' ','')


    #Changing numerical data into float types
    output_column = dataset.columns[-1]

    string_columns = []
    numerical_columns = []
    output_is_string = False
    for c in dataset.columns :
        try :
            dataset[c] = dataset[c].astype(float) #transform numerical data that is stored as string to float type
            numerical_columns.append(c)
        except ValueError :
            if c == output_column:
                output_is_string = True
            string_columns.append(c)

    #Ordinal encoding of the output variable
    if output_is_string:
        outputs = dataset[output_column].unique()
        assert len(outputs) == classes, 'Error the number of output classes should be the same as the one in the dataset'
        dataset[output_column] = 1*(dataset[output_column] == outputs[0]) #return an ordinal encoding of the output variable
    dataset[output_column] = dataset[output_column].astype(int)

    dataset[numerical_columns] = dataset[numerical_columns].astype(float)
    return dataset
```

```python
def detect_type(data):
    """
    @author : Ilyas BENGHABRIT
    Detecting the type of the variables numerical or categorical

    Args:
        :param data: The dataset.
        :type  data: pandas.DataFrame

    Returns:
        :param num_variables: List of columns with numerical values.
        :type num_variables: list

        :param categ_variables: List of columns with categorical values.
        :type categ_variables: list
    """

    num_variables = []
    categ_variables = []
    columns = list(data.columns)
    n = len(columns)
    for i in range(n):
        if data[columns[i]].dtype in  ['int32','int64','float32','float64','int','float']: #Checking if the variable is numerical
            num_variables.append(columns[i])
        else :
            categ_variables.append(columns[i])
    return num_variables, categ_variables


def replace_missing(data, num_variables, categ_variables, num_strategy = 'mean', categ_strategy = 'most_frequent'):
    """
    @author : Ilyas BENGHABRIT
    Replacing the missing values in categorical variables and numerical variables by 2 corresponding strategies
    (mean for numerical variables and the most frequent value for categorical varibles for example)

    Args:
        :param data: The dataset.
        :type  data: pandas.DataFrame

        :param num_variables: List of columns with numerical values.
        :type num_variables: list

        :param categ_variables: List of columns with categorical values.
        :type categ_variables: list

        :param num_strategy: The defined strategy to replace the numerical missing values, default is mean.
        :type num_strategy: str

        :param categ_strategy: The defined strategy to replace the categorical missing values, default is most_frequent.
        :type categ_strategy: str

    Returns:
        :param data_tr_table: A transformed dataset with missing values filled.
        :type data_tr_table: pandas.DataFrame
    """

    ct = ColumnTransformer([("categ_imput", SimpleImputer(missing_values = np.nan, strategy = categ_strategy),
categ_variables),
                        ("num_imput", SimpleImputer(missing_values = np.nan, strategy = num_strategy), num_variables)])
    data_transformed = ct.fit_transform(data) #Recuperate the transformed array
    columns = categ_variables + num_variables
    data_tr_table = pd.DataFrame(data_transformed, columns = columns)#Putting the transformation into a dataframe
    return data_tr_table

def center_encode(data, num_variables, categ_variables):
    """
```

```
    @author : Ilyas BENGHABRIT
    Centring and normalizing the data. Transforming the categorical variables.

    Args:
        :param data: The dataset.
        :type  data: pandas.DataFrame

        :param num_variables: List of columns with numerical values.
        :type num_variables: list

        :param categ_variables: List of columns with categorical values.
        :type categ_variables: list

    Returns:
        :param data_tr_table: A centered and encoded dataset.
        :type data_tr_table: pandas.DataFrame
    """
    cat_enc = OrdinalEncoder()
    center_norm = StandardScaler()
    if categ_variables != [] and num_variables != []:
        categ_data = data[categ_variables]
        num_data = data[num_variables]
        data_transformed_cat = cat_enc.fit_transform(categ_data)
        data_transformed_num = center_norm.fit_transform(num_data) # center and normalize numerical data
        categ_columns = categ_variables
        columns = list(categ_columns) + num_variables
        data_transformed = np.concatenate((data_transformed_cat, data_transformed_num), axis = 1)
        data_tr_table = pd.DataFrame(data_transformed, columns = columns)
    elif categ_variables != []:
        categ_data = data[categ_variables]
        data_transformed_cat = cat_enc.fit_transform(categ_data)
        columns = categ_variables
        data_tr_table = pd.DataFrame(data_transformed_cat, columns = columns)
    elif num_variables != []:
        num_data = data[num_variables]
        data_transformed_num = center_norm.fit_transform(num_data)
        columns = num_variables
        data_tr_table = pd.DataFrame(data_transformed_num, columns = columns)
    return data_tr_table


def feature_selection(dataset,cut_off_variance=0.95):
    """
    @author : Yassir BENDOU
    Applies Feature selection using PCA. We fix the number of the remaining vectors based on the number of components which
garantees 95% of the original variance of the dataset.

    Args:
        :param dataset: The dataset.
        :type  dataset: pandas.DataFrame

        :param cut_off_variance: The threshold ratio of the explained variance, takes values between 0 and 1, default is 0.95.
        :type cut_off_variance: float

    Returns:
        :param data_compressed: The compressed dataset.
        :type data_compressed: pandas.DataFrame
    """
    X,_ = dataset.values[:,:-1],dataset.values[:,-1]
    pca = PCA(n_components=len(dataset.columns)-1)
    pca.fit(X)
    y = np.cumsum(pca.explained_variance_ratio_) # get the cumulative explained variance

    number_of_features = len(dataset.columns) - sum(y>=cut_off_variance)
    print(f'number of features selected : {number_of_features}')

    new_X = pca.fit_transform(X) # get the projections on the eigen vectors of PCA.
```

```python
    data_compressed = pd.DataFrame(new_X[:,:number_of_features]) # only keep the desired features according to the cut_off_variance
    data_compressed[dataset.columns[-1]] = dataset.iloc[:,-1]

    return data_compressed

def split_data(X, y):
    """
    @author : Hafsa OULED ATTOU
    Split the data into training set and validation set

    Args:
        :param X: The features
        :type  X: numpy.array

        :param y: labels
        :type  y: numpy.array

    Returns: Training set and validation set of the data, training set and validation set of the class
    """
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3, random_state = 42)
    return X_train, X_test, y_train, y_test

def determine_combinations(parameters):
    """
    @author : Yassir BENDOU
    Determine all possible combinaisons to have from a defined set of parameters of a model

    Args:
        :param parameters:  Dictionary of parameters.
        :type  parameters: dict

    Returns:
        :param comb_parameters: List of all combinaisons, each combinaison of parameters defined as a dictionary.
        :type comb_parameters: list
    """
    parameters_values = list(parameters.values())
    combinations = list(itertools.product(*parameters_values))
    comb_parameters = []
    for c in combinations :
        d = {}
        keys = list(parameters.keys())
        for k in range(len(keys)):
            d[keys[k]] = c[k]
        comb_parameters.append(d)
    return comb_parameters

def training(model, parameters, X, y):
    """
    @author : Hafsa OULED ATTOU
    Train the model with defined parameters and returns the cross validation score

    Input : Model, parameters of the model, data, class
    Output : the score of the cross validation

    Args:
        :param model:  the defined model.
        :type  model: object

        :param parameters: parameters of the model.
        :type  parameters: dict

        :param X: features.
        :type  X: numpy.array

        :param y: labels.
        :type  y: numpy.array
```

```
    Returns:
        :param score: the score of the cross validation.
        :type score: float

        :param clf: the trained model.
        :type clf: object
    """
    clf = model(**parameters)
    scores = cross_val_score(clf, X, y, cv=5, scoring='f1_macro')
    clf.fit(X,y)
    score = np.mean(scores)
    return score,clf

def select_params(model, parameters, X, y):
    """
    @author : Ilyas BENGHABRIT
    Selecting the best parameters to take for the model

    Input : Model, dictionary of possible parameters, Data, class
    Output :

    Args:
        :param model:  the defined model.
        :type  model: object

        :param parameters: all possible parameters of the model for the gridsearch, format is a dictionary of lists. \nExample :
'parameters':{'loss':['log', 'squared_hinge', 'perceptron']}}
        :type  parameters: dict

        :param X: features.
        :type  X: numpy.array

        :param y: labels.
        :type  y: numpy.array
    Returns:
        :param chosen_params: Chosen combinaison of parameters, score of the cross validation with this combinaison.
        :type chosen_params: dict

        :param score_max: score of the chosen parameters, which is the highest score.
        :type score_max: float
    """

    comb_parameters = determine_combinations(parameters) # Get all the possible combinations
    total_scores = []
    for i in range(len(comb_parameters)):
        total_scores.append(training(model, comb_parameters[i], X, y)[0])
    score_max = np.max(total_scores)
    ind_max = np.argmax(total_scores)
    chosen_params = comb_parameters[ind_max] # return the parameters with the highest score.

    return chosen_params, score_max


def test_evaluate(model,X,y):
    """
    @author : Hafsa OULED ATTOU
    Evaluate the model on test data and returns the f1 score.
    """
    y_pred = model.predict(X)
    score = f1_score(y, y_pred,average='macro')
    return score
```