```
import time
import os, sys
import numpy as np
import pandas as pd
from tqdm import tqdm
from pathlib import Path
import keras.backend as K
from datetime import datetime

from scripts.classifier import rf

#--------------------------------------------------------------------------------------------------
# HiddenPrints
#--------------------------------------------------------------------------------------------------
class HiddenPrints:
    def __enter__(self):
        self._original_stdout = sys.stdout
        sys.stdout = open(os.devnull, 'w')

    def __exit__(self, exc_type, exc_val, exc_tb):
        sys.stdout.close()
        sys.stdout = self._original_stdout

def from_categorical(x):
    """Returns the class vector for a given one-hot vector.

    Parameters
    ----------
    x: np. One-hot vector.
```

```
    Returns
    -------
    np: np.argmax(x, axis= 1)
    """
    return np.argmax(x, axis= 1)

def accuracy(actual, predicted):
    """Function to calculate accuracy for a list of actual and a list of predicted elements.

    Parameters
    ----------
    actual: list. List with the actual labels.
    predicted: list. List with the predicted labels.

    Returns
    -------
    float: Accuracy
    """

    assert len(actual) == len(predicted)

    correct = 0
    for a,b in zip (actual, predicted):
        if a == b:
            correct += 1

    return correct/len(actual)
```

```
def macro_f1_score(actual, predicted):
    """Function to calculate macro f1_score for a multi class problem.

    Parameters
    ----------
    actual: list. List with the actual labels.
    predicted: list. List with the predicted labels.

    Returns
    -------
    float: macro f1_score
    """

    result = []
    # Calculate score for all classes
    for cl in np.unique(actual):

        x = two_class_f1_score(actual==cl, predicted==cl)
        result.append(x)

    return np.mean(result)

def two_class_f1_score(actual, predicted):
    """Function to calculate f1_score for a two class problem. Lists must given with True/False inside.

    Parameters
    ----------
    actual: list. List with the actual labels. Must contain either True or False.
    predicted: list. List with the predicted labels. Must contain either True or False.
```

```
    Returns
    -------
    float: f1_score
    """
    tp = sum(actual & predicted)
    fp = sum(predicted) - tp
    fn = sum(actual) - tp

    if tp>0:
        precision=float(tp)/(tp+fp)
        recall=float(tp)/(tp+fn)

        return 2*((precision*recall)/(precision+recall))
    else:
        return 0


def leave_one_subject_out(data, subjects, subject_id):
    """Function to split dataset into test and training set. Elements of subject (subject_id) are part of the test set.
    Rest is part of training set. The column including subject IDs is removed finally.

    Parameters
    ----------
    data: list. List of nps. All nps will be splitted in training/testing set.
    subject_id: int. ID of a subject.

    Returns
    -------
```

```
    np: x_train, x_test, y_train, y_test
    """

    testing = subjects==subject_id

    # Extract data/label with testing_indices
    test = [i[testing] if i is not None else None for i in data]

    # Delete thos values from the initial data/label
    train = [i[~testing] if i is not None else None for i in data]

    return train + test

def loso_cross_validation(X, y, aug, hcf, subjects, clf, output_csv = Path("results", "loso.csv"), save_model_summary= False):
    """Function to validate a keras model using leave one subject out validation.

    Args:
        X (Np): X data from dataset.
        hcf (Np): hcf data from dataset.
        y (Np): y data from dataset.
        subjects (Np): subjects data from dataset.
        clf (classifer): Classifier chosen form 'classifier.py'.
        output_csv (path, optional): Path to the output CSV. Defaults to str(Path("results", "5_loso.csv")).
        save_model_summary (bool, optional): Whether to save the models summary in a txt file. Defaults to True.
    """

    start_date= datetime.now()
```

```
    start_time= time.time()


    all_accs = []
    all_fscores = []
    all_actuals = []
    all_predictions= []


    rfe_features = []


    df_importance = pd.DataFrame([])


    for subject in (pbar := tqdm(np.unique(subjects))):


        x_train, y_train, hcf_train, sub_train, x_test, y_test, hcf_test, sub_test = leave_one_subject_out(
                [X, y, hcf, subjects], subjects, subject)


        aug_x_train, aug_y_train, aug_sub_train, aug_x_test, aug_y_test, aug_sub_test =
leave_one_subject_out(
                [aug["X"], aug["y"], aug["subjects"]], aug["subjects"], subject)


        # completly delete old classifier and instantiate new one
        clf = type(clf)(clf.param)


        clf.set_dataset(  train_data= (x_train, y_train),
                    test_data= (x_test, y_test),
                    hcf_data= (hcf_train, hcf_test),
                    sub_data= (sub_train, sub_test),
                    aug_train = (aug_x_train, aug_y_train))
```

```
        clf.data_processing()
        clf.create_model()
        clf.train()

        # Save prediction and actual values
        fold_predictions = list(from_categorical(clf.predict_test()))
        all_predictions.extend([fold_predictions])
        fold_actuals = list(from_categorical(clf.y_test))
        all_actuals.extend([fold_actuals])

        fold_acc = accuracy(fold_actuals, fold_predictions)
        all_accs.append(fold_acc)
        all_fscores.append(macro_f1_score(fold_actuals, fold_predictions))

        # save importance
        if hasattr(clf, "model") and hasattr(clf.model, "feature_importances_"):
            importance = pd.DataFrame(clf.model.feature_importances_.reshape(1,-1),
columns=list(clf.hcf_train.columns))
            df_importance = pd.concat([df_importance, importance], ignore_index=True)

        if "rfe_features" in clf.param:
            rfe_features.extend(clf.param["rfe_features"])

        acc = round(np.nanmean(all_accs, axis= 0) *100, 2)
        pbar.set_description(f"Accuracy '{acc}'")

        K.clear_session()

    if len(rfe_features) != 0:
```

```
        clf.param["rfe_features"] = list(set(clf.param["rfe_features"]))


    aug_method = clf.param["aug_method"]
    aug_factor = clf.param["aug_factor"]


    save_data(clf, aug_method, aug_factor, output_csv, start_date, start_time, all_fscores, all_accs,
df_importance, save_model_summary)


    return          all_fscores, all_accs, all_predictions, all_actuals


def save_data(clf, aug_method, aug_factor, output_csv, start_date, start_time, all_fscores, all_accs,
df_importance, save_model_summary):
    # create output directory if does not exist
    output_dir = output_csv.parent
    if not output_dir.exists():
        os.makedirs(output_dir)


    # --- Make entry in results datasheet
    df =  pd.DataFrame()
    now_date = datetime.now()
    df.loc[0, "Start time"]= start_date
    df.loc[0, "End time"]= now_date
    df.loc[0, "Duration"]= str(now_date-start_date).split('.')[0]
    df.loc[0, "Net"]= clf.name
    df.loc[0, "Aug_method"]= aug_method
    df.loc[0, "Aug_factor"]= aug_factor
    df.loc[0, "F1 mean"] = round(np.nanmean(all_fscores) * 100, 2)
    df.loc[0, "F1 std"] = round(np.std(all_fscores) * 100, 2)
    df.loc[0, "Accuracy mean"] = round(np.nanmean(all_accs) * 100, 2)
```

```
    df.loc[0, "Accuracy std"] = round(np.std(all_accs) * 100, 2)
    df.loc[0, "F1"] = str(all_fscores)
    df.loc[0, "Accs"] = str(all_accs)
    df.loc[0, "Param"] = str(sorted(clf.param.items()))

    df.to_csv(output_csv, sep= ";", mode='a', decimal=',', index= False, header= not output_csv.exists())

    # --- save feature importance
    if hasattr(clf, "model") and hasattr(clf.model, "feature_importances_"):
        output_dir_importance = Path(output_dir, "importance", clf.param["dataset"])
        if not output_dir_importance.exists():
            os.makedirs(output_dir_importance)
        df_importance.index.name = "Subject"
        mean_importance = pd.DataFrame({"Mean": df_importance.mean(axis= 0)}).T
        df_importance = pd.concat([df_importance, mean_importance])
        df_importance = df_importance.astype("float")
        df_importance.to_csv(Path(output_dir_importance,
"{}_importance.csv".format(round(start_time))), sep= ";", decimal=',', index= True)

        best_features = df_importance.loc["Mean"].sort_values()
        best_features.index.name = "Feature"
        best_features.name = "Importance"
        best_features.to_csv(Path(output_dir_importance,
"{}_bestfeatures.csv".format(round(start_time))), sep= ";", decimal=',', index= True)

    # --- Save model summary
    if save_model_summary:
        if hasattr(clf, "model") and hasattr(clf.model, 'summary'):
            file_name = str(now_date).replace(" ", "_").replace(":", "-").partition(".")[0] + ".txt"
```

```
        model_summary_path = Path(output_csv.parent, "keras_summaries", file_name)
        if not Path(model_summary_path).parent.exists():
            os.makedirs(Path(model_summary_path).parent)

        # Create summary txt
        with open(model_summary_path, 'w') as f:
            clf.model.summary(print_fn=lambda x: f.write(x + '\n'))


def five_loso(X, y, aug, hcf, subjects, clf, aug_method, aug_factor, runs= 5, output_csv = Path("results", "5_loso.csv")):
    """Function to validate a keras model using a leave one subject out validation 5 times and computing the mean.

    Args:
        X (Np): X data from dataset.
        hcf (Np): hcf data from dataset.
        y (Np): y data from dataset.
        subjects (Np): subjects data from dataset.
        clf (classifer): Classifier chosen form 'classifier.py'.
        runs (int, optional): Number of runs to evaluate. Defaults to 5.
        output_csv (path, optional): Path to the output CSV. Defaults to str(Path("results", "5_loso.csv")).
    """

    start_date= datetime.now()

    acc_mean = []
    acc_std = []
    f1_mean = []
    f1_std = []
```

```
all_predictions = []
all_actuals = []

for i in tqdm(np.arange(runs)):
    fscores, accs, predictions, actuals = loso_cross_validation(X, y, aug, hcf, subjects, clf)

    acc_mean.append(np.nanmean(accs))
    acc_std.append(np.std(accs))
    f1_mean.append(np.nanmean(fscores))
    f1_std.append(np.std(fscores))

    all_predictions.extend(predictions)
    all_actuals.extend(actuals)

# --- Make entry in results datasheet
df =  pd.DataFrame()
now_date = datetime.now()
df.loc[0, "Start time"]= start_date
df.loc[0, "End time"]= now_date
df.loc[0, "Duration"]= str(now_date-start_date).split('.')[0]
df.loc[0, "Net"]= clf.name
df.loc[0, "Aug_method"]= aug_method
df.loc[0, "Aug_factor"]= aug_factor
df.loc[0, "Max. acc mean "] = round(np.nanmax(acc_mean) * 100, 2)
df.loc[0, "Avg. acc mean"] = round(np.nanmean(acc_mean) * 100, 2)
df.loc[0, "Std. acc mean"] = round(np.std(acc_mean) * 100, 2)
df.loc[0, "Avg. acc std"] = round(np.nanmean(acc_std) * 100, 2)
df.loc[0, "Std. acc std"] = round(np.std(acc_std) * 100, 2)
df.loc[0, "Max. F1 mean"] = round(np.nanmax(f1_mean) * 100, 2)
```

```
df.loc[0, "Avg. F1 mean"] = round(np.nanmean(f1_mean) * 100, 2)
df.loc[0, "Std. F1 mean"] = round(np.std(f1_mean) * 100, 2)
df.loc[0, "Avg. F1 std"] = round(np.nanmean(f1_std) * 100, 2)
df.loc[0, "Std. F1 std"] = round(np.std(f1_std) * 100, 2)
df.loc[0, "All F1 mean"] = str(f1_mean)
df.loc[0, "All F1 std"] = str(f1_std)
df.loc[0, "All Acc mean"] = str(acc_mean)
df.loc[0, "All Acc std"] = str(acc_std)
df.loc[0, "Param"] = str(sorted(clf.param.items()))

if not output_csv.parent.exists():
   os.makedirs(output_csv.parent)

df.to_csv(output_csv, sep= ";", mode='a', decimal=',', index= False, header= not output_csv.exists())

return          f1_mean, acc_mean, all_predictions, all_actuals
```