# LIBRARIES

# set up backend for ssh -x11 figures

import matplotlib

matplotlib.use('Agg')

# read and write

import os

import sys

import glob

import re

import fnmatch

import csv

import shutil

from datetime import datetime

# maths

import numpy as np

import pandas as pd

import math

import random

# miscellaneous

import warnings

import gc

import timeit

# sklearn

from sklearn.utils import resample

from sklearn.preprocessing import OneHotEncoder

from sklearn.metrics import mean\_squared\_error, mean\_absolute\_error, r2\_score, log\_loss, roc\_auc\_score, \

accuracy\_score, f1\_score, precision\_score, recall\_score, confusion\_matrix, average\_precision\_score

from sklearn.utils.validation import check\_is\_fitted

from sklearn.model\_selection import KFold, PredefinedSplit, cross\_validate

from sklearn.pipeline import Pipeline

from sklearn.linear\_model import LinearRegression, ElasticNet

from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor

from sklearn.neural\_network import MLPRegressor

from sklearn.preprocessing import StandardScaler

# Statistics

from scipy.stats import pearsonr, ttest\_rel, norm

# Other tools for ensemble models building (Samuel Diai's InnerCV class)

from hyperopt import fmin, tpe, space\_eval, Trials, hp, STATUS\_OK

from xgboost import XGBRegressor

from lightgbm import LGBMRegressor

# CPUs

from multiprocessing import Pool

# GPUs

from GPUtil import GPUtil

# tensorflow

import tensorflow as tf

# keras

from keras\_preprocessing.image import ImageDataGenerator, Iterator

from keras\_preprocessing.image.utils import load\_img, img\_to\_array, array\_to\_img

from tensorflow.keras.utils import Sequence

from tensorflow.keras.models import Model, Sequential

from tensorflow.keras.layers import Flatten, Dense, Dropout, GlobalAveragePooling2D, concatenate

from tensorflow.keras import regularizers

from tensorflow.keras.optimizers import Adam, RMSprop, Adadelta

from tensorflow.keras.callbacks import Callback, EarlyStopping, ReduceLROnPlateau, ModelCheckpoint, CSVLogger

from tensorflow.keras.losses import MeanSquaredError, BinaryCrossentropy

from tensorflow.keras.metrics import RootMeanSquaredError, MeanAbsoluteError, AUC, BinaryAccuracy, Precision, Recall, \

TruePositives, FalsePositives, FalseNegatives, TrueNegatives

from tensorflow\_addons.metrics import RSquare, F1Score

# Plots

import matplotlib.pyplot as plt

import matplotlib.cm as cm

from PIL import Image

from bioinfokit import visuz

# Model's attention

from keract import get\_activations, get\_gradients\_of\_activations

from scipy.ndimage.interpolation import zoom

# Survival

from lifelines.utils import concordance\_index

# Necessary to define MyCSVLogger

import collections

import csv

import io

import six

from tensorflow.python.lib.io import file\_io

from tensorflow.python.util.compat import collections\_abc

from tensorflow.keras.backend import eval

# Set display parameters

pd.set\_option('display.max\_rows', 200)

# CLASSES

class Basics:

"""

Root class herited by most other class. Includes handy helper functions

"""

def \_\_init\_\_(self):

# seeds for reproducibility

self.seed = 0

os.environ['PYTHONHASHSEED'] = str(self.seed)

np.random.seed(self.seed)

random.seed(self.seed)

# other parameters

self.path\_data = '../data/'

self.folds = ['train', 'val', 'test']

self.n\_CV\_outer\_folds = 10

self.outer\_folds = [str(x) for x in list(range(self.n\_CV\_outer\_folds))]

self.modes = ['', '\_sd', '\_str']

self.id\_vars = ['id', 'eid', 'instance', 'outer\_fold']

self.instances = ['0', '1', '1.5', '1.51', '1.52', '1.53', '1.54', '2', '3']

self.ethnicities\_vars\_forgot\_Other = \

['Ethnicity.White', 'Ethnicity.British', 'Ethnicity.Irish', 'Ethnicity.White\_Other', 'Ethnicity.Mixed',

'Ethnicity.White\_and\_Black\_Caribbean', 'Ethnicity.White\_and\_Black\_African', 'Ethnicity.White\_and\_Asian',

'Ethnicity.Mixed\_Other', 'Ethnicity.Asian', 'Ethnicity.Indian', 'Ethnicity.Pakistani',

'Ethnicity.Bangladeshi', 'Ethnicity.Asian\_Other', 'Ethnicity.Black', 'Ethnicity.Caribbean',

'Ethnicity.African', 'Ethnicity.Black\_Other', 'Ethnicity.Chinese', 'Ethnicity.Other\_ethnicity',

'Ethnicity.Do\_not\_know', 'Ethnicity.Prefer\_not\_to\_answer', 'Ethnicity.NA']

self.ethnicities\_vars = \

['Ethnicity.White', 'Ethnicity.British', 'Ethnicity.Irish', 'Ethnicity.White\_Other', 'Ethnicity.Mixed',

'Ethnicity.White\_and\_Black\_Caribbean', 'Ethnicity.White\_and\_Black\_African', 'Ethnicity.White\_and\_Asian',

'Ethnicity.Mixed\_Other', 'Ethnicity.Asian', 'Ethnicity.Indian', 'Ethnicity.Pakistani',

'Ethnicity.Bangladeshi', 'Ethnicity.Asian\_Other', 'Ethnicity.Black', 'Ethnicity.Caribbean',

'Ethnicity.African', 'Ethnicity.Black\_Other', 'Ethnicity.Chinese', 'Ethnicity.Other',

'Ethnicity.Other\_ethnicity', 'Ethnicity.Do\_not\_know', 'Ethnicity.Prefer\_not\_to\_answer', 'Ethnicity.NA']

self.demographic\_vars = ['Age', 'Sex'] + self.ethnicities\_vars

self.names\_model\_parameters = ['target', 'organ', 'view', 'transformation', 'architecture', 'n\_fc\_layers',

'n\_fc\_nodes', 'optimizer', 'learning\_rate', 'weight\_decay', 'dropout\_rate',

'data\_augmentation\_factor']

self.targets\_regression = ['Age']

self.targets\_binary = ['Sex']

self.models\_types = ['', '\_bestmodels']

self.dict\_prediction\_types = {'Age': 'regression', 'Sex': 'binary'}

self.dict\_side\_predictors = {'Age': ['Sex'] + self.ethnicities\_vars\_forgot\_Other,

'Sex': ['Age'] + self.ethnicities\_vars\_forgot\_Other}

self.organs = ['Brain', 'Eyes', 'Arterial', 'Heart', 'Abdomen', 'Musculoskeletal']

self.left\_right\_organs\_views = ['Eyes\_Fundus', 'Eyes\_OCT', 'Arterial\_Carotids', 'Musculoskeletal\_Hips',

'Musculoskeletal\_Knees']

self.dict\_organs\_to\_views = {'Brain': ['MRI'],

'Eyes': ['Fundus', 'OCT'],

'Arterial': ['Carotids'],

'Heart': ['MRI'],

'Abdomen': ['Liver', 'Pancreas'],

'Musculoskeletal': ['Spine', 'Hips', 'Knees', 'FullBody'],

'PhysicalActivity': ['FullWeek']}

self.dict\_organsviews\_to\_transformations = \

{'Brain\_MRI': ['SagittalRaw', 'SagittalReference', 'CoronalRaw', 'CoronalReference', 'TransverseRaw',

'TransverseReference'],

'Arterial\_Carotids': ['Mixed', 'LongAxis', 'CIMT120', 'CIMT150', 'ShortAxis'],

'Heart\_MRI': ['2chambersRaw', '2chambersContrast', '3chambersRaw', '3chambersContrast', '4chambersRaw',

'4chambersContrast'],

'Musculoskeletal\_Spine': ['Sagittal', 'Coronal'],

'Musculoskeletal\_FullBody': ['Mixed', 'Figure', 'Skeleton', 'Flesh'],

'PhysicalActivity\_FullWeek': ['GramianAngularField1minDifference', 'GramianAngularField1minSummation',

'MarkovTransitionField1min', 'RecurrencePlots1min']}

self.dict\_organsviews\_to\_transformations.update(dict.fromkeys(['Eyes\_Fundus', 'Eyes\_OCT'], ['Raw']))

self.dict\_organsviews\_to\_transformations.update(

dict.fromkeys(['Abdomen\_Liver', 'Abdomen\_Pancreas'], ['Raw', 'Contrast']))

self.dict\_organsviews\_to\_transformations.update(

dict.fromkeys(['Musculoskeletal\_Hips', 'Musculoskeletal\_Knees'], ['MRI']))

self.organsviews\_not\_to\_augment = []

self.organs\_instances23 = ['Brain', 'Eyes', 'Arterial', 'Heart', 'Abdomen', 'Musculoskeletal',

'PhysicalActivity']

self.organs\_XWAS = \

['\*', '\*instances01', '\*instances1.5x', '\*instances23', 'Brain', 'BrainCognitive', 'BrainMRI', 'Eyes',

'EyesFundus', 'EyesOCT', 'Hearing', 'Lungs', 'Arterial', 'ArterialPulseWaveAnalysis', 'ArterialCarotids',

'Heart', 'HeartECG', 'HeartMRI', 'Abdomen', 'AbdomenLiver', 'AbdomenPancreas', 'Musculoskeletal',

'MusculoskeletalSpine', 'MusculoskeletalHips', 'MusculoskeletalKnees', 'MusculoskeletalFullBody',

'MusculoskeletalScalars', 'PhysicalActivity', 'Biochemistry', 'BiochemistryUrine', 'BiochemistryBlood',

'ImmuneSystem']

# Others

if '/Users/Alan/' in os.getcwd():

os.chdir('/Users/Alan/Desktop/Aging/Medical\_Images/scripts/')

else:

os.chdir('/n/groups/patel/Alan/Aging/Medical\_Images/scripts/')

gc.enable() # garbage collector

warnings.filterwarnings('ignore')

def \_version\_to\_parameters(self, model\_name):

parameters = {}

parameters\_list = model\_name.split('\_')

for i, parameter in enumerate(self.names\_model\_parameters):

parameters[parameter] = parameters\_list[i]

if len(parameters\_list) > 11:

parameters['outer\_fold'] = parameters\_list[11]

return parameters

@staticmethod

def \_parameters\_to\_version(parameters):

return '\_'.join(parameters.values())

@staticmethod

def convert\_string\_to\_boolean(string):

if string == 'True':

boolean = True

elif string == 'False':

boolean = False

else:

print('ERROR: string must be either \'True\' or \'False\'')

sys.exit(1)

return boolean

class Metrics(Basics):

"""

Helper class defining dictionaries of metrics and custom metrics

"""

def \_\_init\_\_(self):

# Parameters

Basics.\_\_init\_\_(self)

self.metrics\_displayed\_in\_int = ['True-Positives', 'True-Negatives', 'False-Positives', 'False-Negatives']

self.metrics\_needing\_classpred = ['F1-Score', 'Binary-Accuracy', 'Precision', 'Recall']

self.dict\_metrics\_names\_K = {'regression': ['RMSE'], # For now, R-Square is buggy. Try again in a few months.

'binary': ['ROC-AUC', 'PR-AUC', 'F1-Score', 'Binary-Accuracy', 'Precision',

'Recall', 'True-Positives', 'False-Positives', 'False-Negatives',

'True-Negatives'],

'multiclass': ['Categorical-Accuracy']}

self.dict\_metrics\_names = {'regression': ['RMSE', 'MAE', 'R-Squared', 'Pearson-Correlation'],

'binary': ['ROC-AUC', 'F1-Score', 'PR-AUC', 'Binary-Accuracy', 'Sensitivity',

'Specificity', 'Precision', 'Recall', 'True-Positives', 'False-Positives',

'False-Negatives', 'True-Negatives'],

'multiclass': ['Categorical-Accuracy']}

self.dict\_losses\_names = {'regression': 'MSE', 'binary': 'Binary-Crossentropy',

'multiclass': 'categorical\_crossentropy'}

self.dict\_main\_metrics\_names\_K = {'Age': 'MAE', 'Sex': 'PR-AUC', 'imbalanced\_binary\_placeholder': 'PR-AUC'}

self.dict\_main\_metrics\_names = {'Age': 'R-Squared', 'Sex': 'ROC-AUC',

'imbalanced\_binary\_placeholder': 'PR-AUC'}

self.main\_metrics\_modes = {'loss': 'min', 'R-Squared': 'max', 'Pearson-Correlation': 'max', 'RMSE': 'min',

'MAE': 'min', 'ROC-AUC': 'max', 'PR-AUC': 'max', 'F1-Score': 'max', 'C-Index': 'max',

'C-Index-difference': 'max'}

self.n\_bootstrap\_iterations = 1000

def rmse(y\_true, y\_pred):

return math.sqrt(mean\_squared\_error(y\_true, y\_pred))

def sensitivity\_score(y, pred):

\_, \_, fn, tp = confusion\_matrix(y, pred.round()).ravel()

return tp / (tp + fn)

def specificity\_score(y, pred):

tn, fp, \_, \_ = confusion\_matrix(y, pred.round()).ravel()

return tn / (tn + fp)

def true\_positives\_score(y, pred):

\_, \_, \_, tp = confusion\_matrix(y, pred.round()).ravel()

return tp

def false\_positives\_score(y, pred):

\_, fp, \_, \_ = confusion\_matrix(y, pred.round()).ravel()

return fp

def false\_negatives\_score(y, pred):

\_, \_, fn, \_ = confusion\_matrix(y, pred.round()).ravel()

return fn

def true\_negatives\_score(y, pred):

tn, \_, \_, \_ = confusion\_matrix(y, pred.round()).ravel()

return tn

self.dict\_metrics\_sklearn = {'mean\_squared\_error': mean\_squared\_error,

'mean\_absolute\_error': mean\_absolute\_error,

'RMSE': rmse,

'Pearson-Correlation': pearsonr,

'R-Squared': r2\_score,

'Binary-Crossentropy': log\_loss,

'ROC-AUC': roc\_auc\_score,

'F1-Score': f1\_score,

'PR-AUC': average\_precision\_score,

'Binary-Accuracy': accuracy\_score,

'Sensitivity': sensitivity\_score,

'Specificity': specificity\_score,

'Precision': precision\_score,

'Recall': recall\_score,

'True-Positives': true\_positives\_score,

'False-Positives': false\_positives\_score,

'False-Negatives': false\_negatives\_score,

'True-Negatives': true\_negatives\_score}

def \_bootstrap(self, data, function):

results = []

for i in range(self.n\_bootstrap\_iterations):

data\_i = resample(data, replace=True, n\_samples=len(data.index))

results.append(function(data\_i['y'], data\_i['pred']))

return np.mean(results), np.std(results)

class PreprocessingMain(Basics):

"""

This class executes the code for step 01. It preprocesses the main dataframe by:

- reformating the rows and columns

- splitting the dataset into folds for the future cross validations

- imputing key missing data

- adding a new UKB instance for physical activity data

- formating the demographics columns (age, sex and ethnicity)

- reformating the dataframe so that different instances of the same participant are treated as different rows

- saving the dataframe

"""

def \_\_init\_\_(self):

Basics.\_\_init\_\_(self)

self.data\_raw = None

self.data\_features = None

self.data\_features\_eids = None

def \_add\_outer\_folds(self):

outer\_folds\_split = pd.read\_csv(self.path\_data + 'All\_eids.csv')

outer\_folds\_split.rename(columns={'fold': 'outer\_fold'}, inplace=True)

outer\_folds\_split['eid'] = outer\_folds\_split['eid'].astype('str')

outer\_folds\_split['outer\_fold'] = outer\_folds\_split['outer\_fold'].astype('str')

outer\_folds\_split.set\_index('eid', inplace=True)

self.data\_raw = self.data\_raw.join(outer\_folds\_split)

def \_impute\_missing\_ecg\_instances(self):

data\_ecgs = pd.read\_csv('/n/groups/patel/Alan/Aging/TimeSeries/scripts/age\_analysis/missing\_samples.csv')

data\_ecgs['eid'] = data\_ecgs['eid'].astype(str)

data\_ecgs['instance'] = data\_ecgs['instance'].astype(str)

for \_, row in data\_ecgs.iterrows():

self.data\_raw.loc[row['eid'], 'Date\_attended\_center\_' + row['instance']] = row['observation\_date']

def \_add\_physicalactivity\_instances(self):

data\_pa = pd.read\_csv(

'/n/groups/patel/Alan/Aging/TimeSeries/series/PhysicalActivity/90001/features/PA\_visit\_date.csv')

data\_pa['eid'] = data\_pa['eid'].astype(str)

data\_pa.set\_index('eid', drop=False, inplace=True)

data\_pa.index.name = 'column\_names'

self.data\_raw = self.data\_raw.merge(data\_pa, on=['eid'], how='outer')

self.data\_raw.set\_index('eid', drop=False, inplace=True)

def \_compute\_sex(self):

# Use genetic sex when available

self.data\_raw['Sex\_genetic'][self.data\_raw['Sex\_genetic'].isna()] = \

self.data\_raw['Sex'][self.data\_raw['Sex\_genetic'].isna()]

self.data\_raw.drop(['Sex'], axis=1, inplace=True)

self.data\_raw.rename(columns={'Sex\_genetic': 'Sex'}, inplace=True)

self.data\_raw.dropna(subset=['Sex'], inplace=True)

def \_compute\_age(self):

# Recompute age with greater precision by leveraging the month of birth

self.data\_raw['Year\_of\_birth'] = self.data\_raw['Year\_of\_birth'].astype(int)

self.data\_raw['Month\_of\_birth'] = self.data\_raw['Month\_of\_birth'].astype(int)

self.data\_raw['Date\_of\_birth'] = self.data\_raw.apply(

lambda row: datetime(row.Year\_of\_birth, row.Month\_of\_birth, 15), axis=1)

for i in self.instances:

self.data\_raw['Date\_attended\_center\_' + i] = \

self.data\_raw['Date\_attended\_center\_' + i].apply(

lambda x: pd.NaT if pd.isna(x) else datetime.strptime(x, '%Y-%m-%d'))

self.data\_raw['Age\_' + i] = self.data\_raw['Date\_attended\_center\_' + i] - self.data\_raw['Date\_of\_birth']

self.data\_raw['Age\_' + i] = self.data\_raw['Age\_' + i].dt.days / 365.25

self.data\_raw.drop(['Date\_attended\_center\_' + i], axis=1, inplace=True)

self.data\_raw.drop(['Year\_of\_birth', 'Month\_of\_birth', 'Date\_of\_birth'], axis=1, inplace=True)

self.data\_raw.dropna(how='all', subset=['Age\_0', 'Age\_1', 'Age\_1.5', 'Age\_1.51', 'Age\_1.52', 'Age\_1.53',

'Age\_1.54', 'Age\_2', 'Age\_3'], inplace=True)

def \_encode\_ethnicity(self):

# Fill NAs for ethnicity on instance 0 if available in other instances

eids\_missing\_ethnicity = self.data\_raw['eid'][self.data\_raw['Ethnicity'].isna()]

for eid in eids\_missing\_ethnicity:

sample = self.data\_raw.loc[eid, :]

if not math.isnan(sample['Ethnicity\_1']):

self.data\_raw.loc[eid, 'Ethnicity'] = self.data\_raw.loc[eid, 'Ethnicity\_1']

elif not math.isnan(sample['Ethnicity\_2']):

self.data\_raw.loc[eid, 'Ethnicity'] = self.data\_raw.loc[eid, 'Ethnicity\_2']

self.data\_raw.drop(['Ethnicity\_1', 'Ethnicity\_2'], axis=1, inplace=True)

# One hot encode ethnicity

dict\_ethnicity\_codes = {'1': 'Ethnicity.White', '1001': 'Ethnicity.British', '1002': 'Ethnicity.Irish',

'1003': 'Ethnicity.White\_Other',

'2': 'Ethnicity.Mixed', '2001': 'Ethnicity.White\_and\_Black\_Caribbean',

'2002': 'Ethnicity.White\_and\_Black\_African',

'2003': 'Ethnicity.White\_and\_Asian', '2004': 'Ethnicity.Mixed\_Other',

'3': 'Ethnicity.Asian', '3001': 'Ethnicity.Indian', '3002': 'Ethnicity.Pakistani',

'3003': 'Ethnicity.Bangladeshi', '3004': 'Ethnicity.Asian\_Other',

'4': 'Ethnicity.Black', '4001': 'Ethnicity.Caribbean', '4002': 'Ethnicity.African',

'4003': 'Ethnicity.Black\_Other',

'5': 'Ethnicity.Chinese',

'6': 'Ethnicity.Other\_ethnicity',

'-1': 'Ethnicity.Do\_not\_know',

'-3': 'Ethnicity.Prefer\_not\_to\_answer',

'-5': 'Ethnicity.NA'}

self.data\_raw['Ethnicity'] = self.data\_raw['Ethnicity'].fillna(-5).astype(int).astype(str)

ethnicities = pd.get\_dummies(self.data\_raw['Ethnicity'])

self.data\_raw.drop(['Ethnicity'], axis=1, inplace=True)

ethnicities.rename(columns=dict\_ethnicity\_codes, inplace=True)

ethnicities['Ethnicity.White'] = ethnicities['Ethnicity.White'] + ethnicities['Ethnicity.British'] + \

ethnicities['Ethnicity.Irish'] + ethnicities['Ethnicity.White\_Other']

ethnicities['Ethnicity.Mixed'] = ethnicities['Ethnicity.Mixed'] + \

ethnicities['Ethnicity.White\_and\_Black\_Caribbean'] + \

ethnicities['Ethnicity.White\_and\_Black\_African'] + \

ethnicities['Ethnicity.White\_and\_Asian'] + \

ethnicities['Ethnicity.Mixed\_Other']

ethnicities['Ethnicity.Asian'] = ethnicities['Ethnicity.Asian'] + ethnicities['Ethnicity.Indian'] + \

ethnicities['Ethnicity.Pakistani'] + ethnicities['Ethnicity.Bangladeshi'] + \

ethnicities['Ethnicity.Asian\_Other']

ethnicities['Ethnicity.Black'] = ethnicities['Ethnicity.Black'] + ethnicities['Ethnicity.Caribbean'] + \

ethnicities['Ethnicity.African'] + ethnicities['Ethnicity.Black\_Other']

ethnicities['Ethnicity.Other'] = ethnicities['Ethnicity.Other\_ethnicity'] + \

ethnicities['Ethnicity.Do\_not\_know'] + \

ethnicities['Ethnicity.Prefer\_not\_to\_answer'] + \

ethnicities['Ethnicity.NA']

self.data\_raw = self.data\_raw.join(ethnicities)

def generate\_data(self):

# Preprocessing

dict\_UKB\_fields\_to\_names = {'34-0.0': 'Year\_of\_birth', '52-0.0': 'Month\_of\_birth',

'53-0.0': 'Date\_attended\_center\_0', '53-1.0': 'Date\_attended\_center\_1',

'53-2.0': 'Date\_attended\_center\_2', '53-3.0': 'Date\_attended\_center\_3',

'31-0.0': 'Sex', '22001-0.0': 'Sex\_genetic', '21000-0.0': 'Ethnicity',

'21000-1.0': 'Ethnicity\_1', '21000-2.0': 'Ethnicity\_2',

'22414-2.0': 'Abdominal\_images\_quality'}

self.data\_raw = pd.read\_csv('/n/groups/patel/uk\_biobank/project\_52887\_41230/ukb41230.csv',

usecols=['eid', '31-0.0', '22001-0.0', '21000-0.0', '21000-1.0', '21000-2.0',

'34-0.0', '52-0.0', '53-0.0', '53-1.0', '53-2.0', '53-3.0', '22414-2.0'])

# Formatting

self.data\_raw.rename(columns=dict\_UKB\_fields\_to\_names, inplace=True)

self.data\_raw['eid'] = self.data\_raw['eid'].astype(str)

self.data\_raw.set\_index('eid', drop=False, inplace=True)

self.data\_raw.index.name = 'column\_names'

self.\_add\_outer\_folds()

self.\_impute\_missing\_ecg\_instances()

self.\_add\_physicalactivity\_instances()

self.\_compute\_sex()

self.\_compute\_age()

self.\_encode\_ethnicity()

# Concatenate the data from the different instances

self.data\_features = None

for i in self.instances:

print('Preparing the samples for instance ' + i)

df\_i = self.data\_raw[['eid', 'outer\_fold', 'Age\_' + i, 'Sex'] + self.ethnicities\_vars +

['Abdominal\_images\_quality']].dropna(subset=['Age\_' + i])

print(str(len(df\_i.index)) + ' samples found in instance ' + i)

df\_i.rename(columns={'Age\_' + i: 'Age'}, inplace=True)

df\_i['instance'] = i

df\_i['id'] = df\_i['eid'] + '\_' + df\_i['instance']

df\_i = df\_i[self.id\_vars + self.demographic\_vars + ['Abdominal\_images\_quality']]

if i != '2':

df\_i['Abdominal\_images\_quality'] = np.nan # not defined for instance 3, not relevant for instances 0, 1

if self.data\_features is None:

self.data\_features = df\_i

else:

self.data\_features = self.data\_features.append(df\_i)

print('The size of the full concatenated dataframe is now ' + str(len(self.data\_features.index)))

# Save age as a float32 instead of float64

self.data\_features['Age'] = np.float32(self.data\_features['Age'])

# Shuffle the rows before saving the dataframe

self.data\_features = self.data\_features.sample(frac=1)

# Generate dataframe for eids pipeline as opposed to instances pipeline

self.data\_features\_eids = self.data\_features[self.data\_features.instance == '0']

self.data\_features\_eids['instance'] = '\*'

self.data\_features\_eids['id'] = [ID.replace('\_0', '\_\*') for ID in self.data\_features\_eids['id'].values]

def save\_data(self):

self.data\_features.to\_csv(self.path\_data + 'data-features\_instances.csv', index=False)

self.data\_features\_eids.to\_csv(self.path\_data + 'data-features\_eids.csv', index=False)

class PreprocessingImagesIDs(Basics):

"""

Splits the different images datasets into folds for the future cross validation

"""

def \_\_init\_\_(self):

Basics.\_\_init\_\_(self)

# Instances 2 and 3 datasets (most medical images, mostly medical images)

self.instances23\_eids = None

self.HEART\_EIDs = None

self.heart\_eids = None

self.FOLDS\_23\_EIDS = None

def \_load\_23\_eids(self):

data\_features = pd.read\_csv(self.path\_data + 'data-features\_instances.csv')

images\_eids = data\_features['eid'][data\_features['instance'].isin([2, 3])]

self.images\_eids = list(set(images\_eids))

def \_load\_heart\_eids(self):

# IDs already used in Heart videos

HEART\_EIDS = {}

heart\_eids = []

for i in range(10):

# Important: The i's data fold is used as \*validation\* fold for outer fold i.

data\_i = pd.read\_csv(

"/n/groups/patel/JbProst/Heart/Data/FoldsAugmented/data-features\_Heart\_20208\_Augmented\_Age\_val\_" + str(

i) + ".csv")

HEART\_EIDS[i] = list(set([int(str(ID)[:7]) for ID in data\_i['eid']]))

heart\_eids = heart\_eids + HEART\_EIDS[i]

self.HEART\_EIDS = HEART\_EIDS

self.heart\_eids = heart\_eids

def \_split\_23\_eids\_folds(self):

self.\_load\_23\_eids()

self.\_load\_heart\_eids()

# List extra images ids, and split them between the different folds.

extra\_eids = [eid for eid in self.images\_eids if eid not in self.heart\_eids]

random.shuffle(extra\_eids)

n\_samples = len(extra\_eids)

n\_samples\_by\_fold = n\_samples / self.n\_CV\_outer\_folds

FOLDS\_EXTRAEIDS = {}

FOLDS\_EIDS = {}

for outer\_fold in self.outer\_folds:

FOLDS\_EXTRAEIDS[outer\_fold] = \

extra\_eids[int((int(outer\_fold)) \* n\_samples\_by\_fold):int((int(outer\_fold) + 1) \* n\_samples\_by\_fold)]

FOLDS\_EIDS[outer\_fold] = self.HEART\_EIDS[int(outer\_fold)] + FOLDS\_EXTRAEIDS[outer\_fold]

self.FOLDS\_23\_EIDS = FOLDS\_EIDS

def \_save\_23\_eids\_folds(self):

for outer\_fold in self.outer\_folds:

with open(self.path\_data + 'instances23\_eids\_' + outer\_fold + '.csv', 'w', newline='') as myfile:

wr = csv.writer(myfile, quoting=csv.QUOTE\_ALL)

wr.writerow(self.FOLDS\_23\_EIDS[outer\_fold])

def generate\_eids\_splits(self):

print("Generating eids split for organs on instances 2 and 3")

self.\_split\_23\_eids\_folds()

self.\_save\_23\_eids\_folds()

class PreprocessingFolds(Metrics):

"""

Splits the data into training, validation and testing sets for all CV folds

"""

def \_\_init\_\_(self, target, organ, regenerate\_data):

Metrics.\_\_init\_\_(self)

self.target = target

self.organ = organ

self.list\_ids\_per\_view\_transformation = None

# Check if these folds have already been generated

if not regenerate\_data:

if len(glob.glob(self.path\_data + 'data-features\_' + organ + '\_\*\_' + target + '\_\*.csv')) > 0:

print("Error: The files already exist! Either change regenerate\_data to True or delete the previous"

" version.")

sys.exit(1)

self.side\_predictors = self.dict\_side\_predictors[target]

self.variables\_to\_normalize = self.side\_predictors

if target in self.targets\_regression:

self.variables\_to\_normalize.append(target)

self.dict\_image\_quality\_col = {'Liver': 'Abdominal\_images\_quality'}

self.dict\_image\_quality\_col.update(

dict.fromkeys(['Brain', 'Eyes', 'Arterial', 'Heart', 'Abdomen', 'Musculoskeletal', 'PhysicalActivity'],

None))

self.image\_quality\_col = self.dict\_image\_quality\_col[organ]

self.views = self.dict\_organs\_to\_views[organ]

self.list\_ids = None

self.list\_ids\_per\_view = {}

self.data = None

self.EIDS = None

self.EIDS\_per\_view = {'train': {}, 'val': {}, 'test': {}}

self.data\_fold = None

def \_get\_list\_ids(self):

self.list\_ids\_per\_view\_transformation = {}

list\_ids = []

# if different views are available, take the union of the ids

for view in self.views:

self.list\_ids\_per\_view\_transformation[view] = {}

for transformation in self.dict\_organsviews\_to\_transformations[self.organ + '\_' + view]:

list\_ids\_transformation = []

path = '../images/' + self.organ + '/' + view + '/' + transformation + '/'

# for paired organs, take the unions of the ids available on the right and the left sides

if self.organ + '\_' + view in self.left\_right\_organs\_views:

for side in ['right', 'left']:

list\_ids\_transformation += os.listdir(path + side + '/')

list\_ids\_transformation = np.unique(list\_ids\_transformation).tolist()

else:

list\_ids\_transformation += os.listdir(path)

self.list\_ids\_per\_view\_transformation[view][transformation] = \

[im.replace('.jpg', '') for im in list\_ids\_transformation]

list\_ids += self.list\_ids\_per\_view\_transformation[view][transformation]

self.list\_ids = np.unique(list\_ids).tolist()

self.list\_ids.sort()

def \_filter\_and\_format\_data(self):

"""

Clean the data before it can be split between the rows

"""

cols\_data = self.id\_vars + self.demographic\_vars

if self.image\_quality\_col is not None:

cols\_data.append(self.dict\_image\_quality\_col[self.organ])

data = pd.read\_csv(self.path\_data + 'data-features\_instances.csv', usecols=cols\_data)

data.rename(columns={self.dict\_image\_quality\_col[self.organ]: 'Data\_quality'}, inplace=True)

for col\_name in self.id\_vars:

data[col\_name] = data[col\_name].astype(str)

data.set\_index('id', drop=False, inplace=True)

if self.image\_quality\_col is not None:

data = data[data['Data\_quality'] != np.nan]

data.drop('Data\_quality', axis=1, inplace=True)

# get rid of samples with NAs

data.dropna(inplace=True)

# list the samples' ids for which images are available

data = data.loc[self.list\_ids]

self.data = data

def \_split\_data(self):

# Generate the data for each outer\_fold

for i, outer\_fold in enumerate(self.outer\_folds):

of\_val = outer\_fold

of\_test = str((int(outer\_fold) + 1) % len(self.outer\_folds))

DATA = {

'train': self.data[~self.data['outer\_fold'].isin([of\_val, of\_test])],

'val': self.data[self.data['outer\_fold'] == of\_val],

'test': self.data[self.data['outer\_fold'] == of\_test]

}

# Generate the data for the different views and transformations

for view in self.views:

for transformation in self.dict\_organsviews\_to\_transformations[self.organ + '\_' + view]:

print('Splitting data for view ' + view + ', and transformation ' + transformation)

DF = {}

for fold in self.folds:

idx = DATA[fold]['id'].isin(self.list\_ids\_per\_view\_transformation[view][transformation]).values

DF[fold] = DATA[fold].iloc[idx, :]

# compute values for scaling of variables

normalizing\_values = {}

for var in self.variables\_to\_normalize:

var\_mean = DF['train'][var].mean()

if len(DF['train'][var].unique()) < 2:

print('Variable ' + var + ' has a single value in fold ' + outer\_fold +

'. Using 1 as std for normalization.')

var\_std = 1

else:

var\_std = DF['train'][var].std()

normalizing\_values[var] = {'mean': var\_mean, 'std': var\_std}

# normalize the variables

for fold in self.folds:

for var in self.variables\_to\_normalize:

DF[fold][var + '\_raw'] = DF[fold][var]

DF[fold][var] = (DF[fold][var] - normalizing\_values[var]['mean']) \

/ normalizing\_values[var]['std']

# report issue if NAs were detected (most likely comes from a sample whose id did not match)

n\_mismatching\_samples = DF[fold].isna().sum().max()

if n\_mismatching\_samples > 0:

print(DF[fold][DF[fold].isna().any(axis=1)])

print('/!\\ WARNING! ' + str(n\_mismatching\_samples) + ' ' + fold + ' images ids out of ' +

str(len(DF[fold].index)) + ' did not match the dataframe!')

# save the data

DF[fold].to\_csv(self.path\_data + 'data-features\_' + self.organ + '\_' + view + '\_' +

transformation + '\_' + self.target + '\_' + fold + '\_' + outer\_fold + '.csv',

index=False)

print('For outer\_fold ' + outer\_fold + ', the ' + fold + ' fold has a sample size of ' +

str(len(DF[fold].index)))

def generate\_folds(self):

self.\_get\_list\_ids()

self.\_filter\_and\_format\_data()

self.\_split\_data()

class PreprocessingSurvival(Basics):

"""

Preprocesses the main dataframe for survival purposes.

Mirrors the PreprocessingMain class, but computes Death time and FollowTime for the future survival analysis

"""

def \_\_init\_\_(self):

Basics.\_\_init\_\_(self)

self.data\_raw = None

self.data\_features = None

self.data\_features\_eids = None

self.survival\_vars = ['FollowUpTime', 'Death']

def \_preprocessing(self):

usecols = ['eid', '40000-0.0', '34-0.0', '52-0.0', '53-0.0', '53-1.0', '53-2.0', '53-3.0']

self.data\_raw = pd.read\_csv('/n/groups/patel/uk\_biobank/project\_52887\_41230/ukb41230.csv', usecols=usecols)

dict\_UKB\_fields\_to\_names = {'40000-0.0': 'FollowUpDate', '34-0.0': 'Year\_of\_birth', '52-0.0': 'Month\_of\_birth',

'53-0.0': 'Date\_attended\_center\_0', '53-1.0': 'Date\_attended\_center\_1',

'53-2.0': 'Date\_attended\_center\_2', '53-3.0': 'Date\_attended\_center\_3'}

self.data\_raw.rename(columns=dict\_UKB\_fields\_to\_names, inplace=True)

self.data\_raw['eid'] = self.data\_raw['eid'].astype(str)

self.data\_raw.set\_index('eid', drop=False, inplace=True)

self.data\_raw.index.name = 'column\_names'

# Format survival data

self.data\_raw['Death'] = ~self.data\_raw['FollowUpDate'].isna()

self.data\_raw['FollowUpDate'][self.data\_raw['FollowUpDate'].isna()] = '2020-04-27'

self.data\_raw['FollowUpDate'] = self.data\_raw['FollowUpDate'].apply(

lambda x: pd.NaT if pd.isna(x) else datetime.strptime(x, '%Y-%m-%d'))

assert ('FollowUpDate.1' not in self.data\_raw.columns)

def \_add\_physicalactivity\_instances(self):

data\_pa = pd.read\_csv(

'/n/groups/patel/Alan/Aging/TimeSeries/series/PhysicalActivity/90001/features/PA\_visit\_date.csv')

data\_pa['eid'] = data\_pa['eid'].astype(str)

data\_pa.set\_index('eid', drop=False, inplace=True)

data\_pa.index.name = 'column\_names'

self.data\_raw = self.data\_raw.merge(data\_pa, on=['eid'], how='outer')

self.data\_raw.set\_index('eid', drop=False, inplace=True)

def \_compute\_age(self):

# Recompute age with greater precision by leveraging the month of birth

self.data\_raw.dropna(subset=['Year\_of\_birth'], inplace=True)

self.data\_raw['Year\_of\_birth'] = self.data\_raw['Year\_of\_birth'].astype(int)

self.data\_raw['Month\_of\_birth'] = self.data\_raw['Month\_of\_birth'].astype(int)

self.data\_raw['Date\_of\_birth'] = self.data\_raw.apply(

lambda row: datetime(row.Year\_of\_birth, row.Month\_of\_birth, 15), axis=1)

for i in self.instances:

self.data\_raw['Date\_attended\_center\_' + i] = self.data\_raw['Date\_attended\_center\_' + i].apply(

lambda x: pd.NaT if pd.isna(x) else datetime.strptime(x, '%Y-%m-%d'))

self.data\_raw['Age\_' + i] = self.data\_raw['Date\_attended\_center\_' + i] - self.data\_raw['Date\_of\_birth']

self.data\_raw['Age\_' + i] = self.data\_raw['Age\_' + i].dt.days / 365.25

self.data\_raw['FollowUpTime\_' + i] = self.data\_raw['FollowUpDate'] - self.data\_raw[

'Date\_attended\_center\_' + i]

self.data\_raw['FollowUpTime\_' + i] = self.data\_raw['FollowUpTime\_' + i].dt.days / 365.25

self.data\_raw.drop(['Date\_attended\_center\_' + i], axis=1, inplace=True)

self.data\_raw.drop(['Year\_of\_birth', 'Month\_of\_birth', 'Date\_of\_birth', 'FollowUpDate'], axis=1, inplace=True)

self.data\_raw.dropna(how='all', subset=['Age\_0', 'Age\_1', 'Age\_1.5', 'Age\_1.51', 'Age\_1.52', 'Age\_1.53',

'Age\_1.54', 'Age\_2', 'Age\_3'], inplace=True)

def \_concatenate\_instances(self):

self.data\_features = None

for i in self.instances:

print('Preparing the samples for instance ' + i)

df\_i = self.data\_raw.dropna(subset=['Age\_' + i])

print(str(len(df\_i.index)) + ' samples found in instance ' + i)

dict\_names = {}

features = ['Age', 'FollowUpTime']

for feature in features:

dict\_names[feature + '\_' + i] = feature

self.dict\_names = dict\_names

df\_i.rename(columns=dict\_names, inplace=True)

df\_i['instance'] = i

df\_i['id'] = df\_i['eid'] + '\_' + df\_i['instance']

df\_i = df\_i[['id', 'eid', 'instance'] + self.survival\_vars]

if self.data\_features is None:

self.data\_features = df\_i

else:

self.data\_features = self.data\_features.append(df\_i)

print('The size of the full concatenated dataframe is now ' + str(len(self.data\_features.index)))

# Add \* instance for eids

survival\_eids = self.data\_features[self.data\_features['instance'] == '0']

survival\_eids['instance'] = '\*'

survival\_eids['id'] = survival\_eids['eid'] + '\_' + survival\_eids['instance']

self.data\_features = self.data\_features.append(survival\_eids)

def generate\_data(self):

# Formatting

self.\_preprocessing()

self.\_add\_physicalactivity\_instances()

self.\_compute\_age()

self.\_concatenate\_instances()

# save data

self.data\_features.to\_csv('../data/data\_survival.csv', index=False)

class MyImageDataGenerator(Basics, Sequence, ImageDataGenerator):

"""

Helper class: custom data generator for images.

It handles several custom features such as:

- provides batches of not only images, but also the scalar data (e.g demographics) that correspond to it

- it performs random shuffling while making sure that no leftover data (the remainder of the modulo batch size)

is being unused

- it can handle paired data for paired organs (e.g left/right eyes)

"""

def \_\_init\_\_(self, target=None, organ=None, view=None, data\_features=None, n\_samples\_per\_subepoch=None,

batch\_size=None, training\_mode=None, side\_predictors=None, dir\_images=None, images\_width=None,

images\_height=None, data\_augmentation=False, data\_augmentation\_factor=None, seed=None):

# Parameters

Basics.\_\_init\_\_(self)

self.target = target

if target in self.targets\_regression:

self.labels = data\_features[target]

else:

self.labels = data\_features[target + '\_raw']

self.organ = organ

self.view = view

self.training\_mode = training\_mode

self.data\_features = data\_features

self.list\_ids = data\_features.index.values

self.batch\_size = batch\_size

# for paired organs, take twice fewer ids (two images for each id), and add organ\_side as side predictor

if organ + '\_' + view in self.left\_right\_organs\_views:

self.data\_features['organ\_side'] = np.nan

self.n\_ids\_batch = batch\_size // 2

else:

self.n\_ids\_batch = batch\_size

if self.training\_mode & (n\_samples\_per\_subepoch is not None): # during training, 1 epoch = number of samples

self.steps = math.ceil(n\_samples\_per\_subepoch / batch\_size)

else: # during prediction and other tasks, an epoch is defined as all the samples being seen once and only once

self.steps = math.ceil(len(self.list\_ids) / self.n\_ids\_batch)

# learning\_rate\_patience

if n\_samples\_per\_subepoch is not None:

self.n\_subepochs\_per\_epoch = math.ceil(len(self.data\_features.index) / n\_samples\_per\_subepoch)

# initiate the indices and shuffle the ids

self.shuffle = training\_mode # Only shuffle if the model is being trained. Otherwise no need.

self.indices = np.arange(len(self.list\_ids))

self.idx\_end = 0 # Keep track of last indice to permute indices accordingly at the end of epoch.

if self.shuffle:

np.random.shuffle(self.indices)

# Input for side NN and CNN

self.side\_predictors = side\_predictors

self.dir\_images = dir\_images

self.images\_width = images\_width

self.images\_height = images\_height

# Data augmentation

self.data\_augmentation = data\_augmentation

self.data\_augmentation\_factor = data\_augmentation\_factor

self.seed = seed

# Parameters for data augmentation: (rotation range, width shift range, height shift range, zoom range)

self.augmentation\_parameters = \

pd.DataFrame(index=['Brain\_MRI', 'Eyes\_Fundus', 'Eyes\_OCT', 'Arterial\_Carotids', 'Heart\_MRI',

'Abdomen\_Liver', 'Abdomen\_Pancreas', 'Musculoskeletal\_Spine', 'Musculoskeletal\_Hips',

'Musculoskeletal\_Knees', 'Musculoskeletal\_FullBody', 'PhysicalActivity\_FullWeek',

'PhysicalActivity\_Walking'],

columns=['rotation', 'width\_shift', 'height\_shift', 'zoom'])

self.augmentation\_parameters.loc['Brain\_MRI', :] = [10, 0.05, 0.1, 0.0]

self.augmentation\_parameters.loc['Eyes\_Fundus', :] = [20, 0.02, 0.02, 0]

self.augmentation\_parameters.loc['Eyes\_OCT', :] = [30, 0.1, 0.2, 0]

self.augmentation\_parameters.loc[['Arterial\_Carotids'], :] = [0, 0.2, 0.0, 0.0]

self.augmentation\_parameters.loc[['Heart\_MRI', 'Abdomen\_Liver', 'Abdomen\_Pancreas',

'Musculoskeletal\_Spine'], :] = [10, 0.1, 0.1, 0.0]

self.augmentation\_parameters.loc[['Musculoskeletal\_Hips', 'Musculoskeletal\_Knees'], :] = [10, 0.1, 0.1, 0.1]

self.augmentation\_parameters.loc[['Musculoskeletal\_FullBody'], :] = [10, 0.05, 0.02, 0.0]

self.augmentation\_parameters.loc[['PhysicalActivity\_FullWeek'], :] = [0, 0, 0, 0.0]

organ\_view = organ + '\_' + view

ImageDataGenerator.\_\_init\_\_(self, rescale=1. / 255.,

rotation\_range=self.augmentation\_parameters.loc[organ\_view, 'rotation'],

width\_shift\_range=self.augmentation\_parameters.loc[organ\_view, 'width\_shift'],

height\_shift\_range=self.augmentation\_parameters.loc[organ\_view, 'height\_shift'],

zoom\_range=self.augmentation\_parameters.loc[organ\_view, 'zoom'])

def \_\_len\_\_(self):

return self.steps

def on\_epoch\_end(self):

\_ = gc.collect()

self.indices = np.concatenate([self.indices[self.idx\_end:], self.indices[:self.idx\_end]])

def \_generate\_image(self, path\_image):

img = load\_img(path\_image, target\_size=(self.images\_width, self.images\_height), color\_mode='rgb')

Xi = img\_to\_array(img)

if hasattr(img, 'close'):

img.close()

if self.data\_augmentation:

params = self.get\_random\_transform(Xi.shape)

Xi = self.apply\_transform(Xi, params)

Xi = self.standardize(Xi)

return Xi

def \_data\_generation(self, list\_ids\_batch):

# initialize empty matrices

n\_samples\_batch = min(len(list\_ids\_batch), self.batch\_size)

X = np.empty((n\_samples\_batch, self.images\_width, self.images\_height, 3)) \* np.nan

x = np.empty((n\_samples\_batch, len(self.side\_predictors))) \* np.nan

y = np.empty((n\_samples\_batch, 1)) \* np.nan

# fill the matrices sample by sample

for i, ID in enumerate(list\_ids\_batch):

y[i] = self.labels[ID]

x[i] = self.data\_features.loc[ID, self.side\_predictors]

if self.organ + '\_' + self.view in self.left\_right\_organs\_views:

if i % 2 == 0:

path = self.dir\_images + 'right/'

x[i][-1] = 0

else:

path = self.dir\_images + 'left/'

x[i][-1] = 1

if not os.path.exists(path + ID + '.jpg'):

path = path.replace('/right/', '/left/') if i % 2 == 0 else path.replace('/left/', '/right/')

x[i][-1] = 1 - x[i][-1]

else:

path = self.dir\_images

X[i, :, :, :] = self.\_generate\_image(path\_image=path + ID + '.jpg')

return [X, x], y

def \_\_getitem\_\_(self, index):

# Select the indices

idx\_start = (index \* self.n\_ids\_batch) % len(self.list\_ids)

idx\_end = (((index + 1) \* self.n\_ids\_batch) - 1) % len(self.list\_ids) + 1

if idx\_start > idx\_end:

# If this happens outside of training, that is a mistake

if not self.training\_mode:

print('\nERROR: Outside of training, every sample should only be predicted once!')

sys.exit(1)

# Select part of the indices from the end of the epoch

indices = self.indices[idx\_start:]

# Generate a new set of indices

# print('\nThe end of the data was reached within this batch, looping.')

if self.shuffle:

np.random.shuffle(self.indices)

# Complete the batch with samples from the new indices

indices = np.concatenate([indices, self.indices[:idx\_end]])

else:

indices = self.indices[idx\_start: idx\_end]

if idx\_end == len(self.list\_ids) & self.shuffle:

# print('\nThe end of the data was reached. Shuffling for the next epoch.')

np.random.shuffle(self.indices)

# Keep track of last indice for end of subepoch

self.idx\_end = idx\_end

# Select the corresponding ids

list\_ids\_batch = [self.list\_ids[i] for i in indices]

# For paired organs, two images (left, right eyes) are selected for each id.

if self.organ + '\_' + self.view in self.left\_right\_organs\_views:

list\_ids\_batch = [ID for ID in list\_ids\_batch for \_ in ('right', 'left')]

return self.\_data\_generation(list\_ids\_batch)

class MyCSVLogger(Callback):

"""

Custom CSV Logger callback class for Keras training: append to existing file if can be found. Allows to keep track

of training over several jobs.

"""

def \_\_init\_\_(self, filename, separator=',', append=False):

self.sep = separator

self.filename = filename

self.append = append

self.writer = None

self.keys = None

self.append\_header = True

self.csv\_file = None

if six.PY2:

self.file\_flags = 'b'

self.\_open\_args = {}

else:

self.file\_flags = ''

self.\_open\_args = {'newline': '\n'}

Callback.\_\_init\_\_(self)

def on\_train\_begin(self, logs=None):

if self.append:

if file\_io.file\_exists(self.filename):

with open(self.filename, 'r' + self.file\_flags) as f:

self.append\_header = not bool(len(f.readline()))

mode = 'a'

else:

mode = 'w'

self.csv\_file = io.open(self.filename, mode + self.file\_flags, \*\*self.\_open\_args)

def on\_epoch\_end(self, epoch, logs=None):

logs = logs or {}

def handle\_value(k):

is\_zero\_dim\_ndarray = isinstance(k, np.ndarray) and k.ndim == 0

if isinstance(k, six.string\_types):

return k

elif isinstance(k, collections\_abc.Iterable) and not is\_zero\_dim\_ndarray:

return '"[%s]"' % (', '.join(map(str, k)))

else:

return k

if self.keys is None:

self.keys = sorted(logs.keys())

if self.model.stop\_training:

# We set NA so that csv parsers do not fail for this last epoch.

logs = dict([(k, logs[k]) if k in logs else (k, 'NA') for k in self.keys])

if not self.writer:

class CustomDialect(csv.excel):

delimiter = self.sep

fieldnames = ['epoch', 'learning\_rate'] + self.keys

if six.PY2:

fieldnames = [unicode(x) for x in fieldnames]

self.writer = csv.DictWriter(

self.csv\_file,

fieldnames=fieldnames,

dialect=CustomDialect)

if self.append\_header:

self.writer.writeheader()

row\_dict = collections.OrderedDict({'epoch': epoch, 'learning\_rate': eval(self.model.optimizer.lr)})

row\_dict.update((key, handle\_value(logs[key])) for key in self.keys)

self.writer.writerow(row\_dict)

self.csv\_file.flush()

def on\_train\_end(self, logs=None):

self.csv\_file.close()

self.writer = None

class MyModelCheckpoint(ModelCheckpoint):

"""

Custom checkpoint callback class for Keras training. Handles a baseline performance.

"""

def \_\_init\_\_(self, filepath, monitor='val\_loss', baseline=-np.Inf, verbose=0, save\_best\_only=False,

save\_weights\_only=False, mode='auto', save\_freq='epoch'):

# Parameters

ModelCheckpoint.\_\_init\_\_(self, filepath, monitor=monitor, verbose=verbose, save\_best\_only=save\_best\_only,

save\_weights\_only=save\_weights\_only, mode=mode, save\_freq=save\_freq)

if mode == 'min':

self.monitor\_op = np.less

self.best = baseline

elif mode == 'max':

self.monitor\_op = np.greater

self.best = baseline

else:

print('Error. mode for metric must be either min or max')

sys.exit(1)

class DeepLearning(Metrics):

"""

Core helper class to train models. Used to:

- build the data generators

- generate the CNN architectures

- load the weights

"""

def \_\_init\_\_(self, target=None, organ=None, view=None, transformation=None, architecture=None, n\_fc\_layers=None,

n\_fc\_nodes=None, optimizer=None, learning\_rate=None, weight\_decay=None, dropout\_rate=None,

data\_augmentation\_factor=None, debug\_mode=False):

# Initialization

Metrics.\_\_init\_\_(self)

tf.random.set\_seed(self.seed)

# Model's version

self.target = target

self.organ = organ

self.view = view

self.transformation = transformation

self.architecture = architecture

self.n\_fc\_layers = int(n\_fc\_layers)

self.n\_fc\_nodes = int(n\_fc\_nodes)

self.optimizer = optimizer

self.learning\_rate = float(learning\_rate)

self.weight\_decay = float(weight\_decay)

self.dropout\_rate = float(dropout\_rate)

self.data\_augmentation\_factor = float(data\_augmentation\_factor)

self.outer\_fold = None

self.version = target + '\_' + organ + '\_' + view + '\_' + transformation + '\_' + architecture + '\_' + \

n\_fc\_layers + '\_' + n\_fc\_nodes + '\_' + optimizer + '\_' + learning\_rate + '\_' + weight\_decay + \

'\_' + dropout\_rate + '\_' + data\_augmentation\_factor

# NNet's architecture and weights

self.side\_predictors = self.dict\_side\_predictors[target]

if self.organ + '\_' + self.view in self.left\_right\_organs\_views:

self.side\_predictors.append('organ\_side')

self.dict\_final\_activations = {'regression': 'linear', 'binary': 'sigmoid', 'multiclass': 'softmax',

'saliency': 'linear'}

self.path\_load\_weights = None

self.keras\_weights = None

# Generators

self.debug\_mode = debug\_mode

self.debug\_fraction = 0.005

self.DATA\_FEATURES = {}

self.mode = None

self.n\_cpus = len(os.sched\_getaffinity(0))

self.dir\_images = '../images/' + organ + '/' + view + '/' + transformation + '/'

# define dictionary to fit the architecture's input size to the images sizes (take min (height, width))

self.dict\_organ\_view\_transformation\_to\_image\_size = {

'Eyes\_Fundus\_Raw': (316, 316), # initial size (1388, 1388)

'Eyes\_OCT\_Raw': (312, 320), # initial size (500, 512)

'Musculoskeletal\_Spine\_Sagittal': (466, 211), # initial size (1513, 684)

'Musculoskeletal\_Spine\_Coronal': (315, 313), # initial size (724, 720)

'Musculoskeletal\_Hips\_MRI': (329, 303), # initial size (626, 680)

'Musculoskeletal\_Knees\_MRI': (347, 286) # initial size (851, 700)

}

self.dict\_organ\_view\_transformation\_to\_image\_size.update(

dict.fromkeys(['Brain\_MRI\_SagittalRaw', 'Brain\_MRI\_SagittalReference', 'Brain\_MRI\_CoronalRaw',

'Brain\_MRI\_CoronalReference', 'Brain\_MRI\_TransverseRaw', 'Brain\_MRI\_TransverseReference'],

(316, 316))) # initial size (88, 88)

self.dict\_organ\_view\_transformation\_to\_image\_size.update(

dict.fromkeys(['Arterial\_Carotids\_Mixed', 'Arterial\_Carotids\_LongAxis', 'Arterial\_Carotids\_CIMT120',

'Arterial\_Carotids\_CIMT150', 'Arterial\_Carotids\_ShortAxis'],

(337, 291))) # initial size (505, 436)

self.dict\_organ\_view\_transformation\_to\_image\_size.update(

dict.fromkeys(['Heart\_MRI\_2chambersRaw', 'Heart\_MRI\_2chambersContrast', 'Heart\_MRI\_3chambersRaw',

'Heart\_MRI\_3chambersContrast', 'Heart\_MRI\_4chambersRaw', 'Heart\_MRI\_4chambersContrast'],

(316, 316))) # initial size (200, 200)

self.dict\_organ\_view\_transformation\_to\_image\_size.update(

dict.fromkeys(['Abdomen\_Liver\_Raw', 'Abdomen\_Liver\_Contrast'], (288, 364))) # initial size (288, 364)

self.dict\_organ\_view\_transformation\_to\_image\_size.update(

dict.fromkeys(['Abdomen\_Pancreas\_Raw', 'Abdomen\_Pancreas\_Contrast'], (288, 350))) # initial size (288, 350)

self.dict\_organ\_view\_transformation\_to\_image\_size.update(

dict.fromkeys(['Musculoskeletal\_FullBody\_Figure', 'Musculoskeletal\_FullBody\_Skeleton',

'Musculoskeletal\_FullBody\_Flesh', 'Musculoskeletal\_FullBody\_Mixed'],

(541, 181))) # initial size (811, 272)

self.dict\_organ\_view\_transformation\_to\_image\_size.update(

dict.fromkeys(['PhysicalActivity\_FullWeek\_GramianAngularField1minDifference',

'PhysicalActivity\_FullWeek\_GramianAngularField1minSummation',

'PhysicalActivity\_FullWeek\_MarkovTransitionField1min',

'PhysicalActivity\_FullWeek\_RecurrencePlots1min'],

(316, 316))) # initial size (316, 316)

self.dict\_architecture\_to\_image\_size = {'MobileNet': (224, 224), 'MobileNetV2': (224, 224),

'NASNetMobile': (224, 224), 'NASNetLarge': (331, 331)}

if self.architecture in ['MobileNet', 'MobileNetV2', 'NASNetMobile', 'NASNetLarge']:

self.image\_width, self.image\_height = self.dict\_architecture\_to\_image\_size[architecture]

else:

self.image\_width, self.image\_height = \

self.dict\_organ\_view\_transformation\_to\_image\_size[organ + '\_' + view + '\_' + transformation]

# define dictionary of batch sizes to fit as many samples as the model's architecture allows

self.dict\_batch\_sizes = {

# Default, applies to all images with resized input ~100,000 pixels

'Default': {'VGG16': 32, 'VGG19': 32, 'DenseNet121': 16, 'DenseNet169': 16, 'DenseNet201': 16,

'Xception': 32, 'InceptionV3': 32, 'InceptionResNetV2': 8, 'ResNet50': 32, 'ResNet101': 16,

'ResNet152': 16, 'ResNet50V2': 32, 'ResNet101V2': 16, 'ResNet152V2': 16, 'ResNeXt50': 4,

'ResNeXt101': 8, 'EfficientNetB7': 4,

'MobileNet': 128, 'MobileNetV2': 64, 'NASNetMobile': 64, 'NASNetLarge': 4}}

# Define batch size

if organ + '\_' + view in self.dict\_batch\_sizes.keys():

randoself.batch\_size = self.dict\_batch\_sizes[organ + '\_' + view][architecture]

else:

self.batch\_size = self.dict\_batch\_sizes['Default'][architecture]

# double the batch size for the teslaM40 cores that have bigger memory

if len(GPUtil.getGPUs()) > 0: # make sure GPUs are available (not truesometimes for debugging)

if GPUtil.getGPUs()[0].memoryTotal > 20000:

self.batch\_size \*= 2

# Define number of ids per batch (twice fewer for paired organs, because left and right samples)

self.n\_ids\_batch = self.batch\_size

if organ + '\_' + view in self.left\_right\_organs\_views:

self.n\_ids\_batch //= 2

# Define number of samples per subepoch

if debug\_mode:

self.n\_samples\_per\_subepoch = self.batch\_size \* 4

else:

self.n\_samples\_per\_subepoch = 32768

if organ + '\_' + view in self.left\_right\_organs\_views:

self.n\_samples\_per\_subepoch //= 2

# dict to decide which field is used to generate the ids when several targets share the same ids

self.dict\_target\_to\_ids = dict.fromkeys(['Age', 'Sex'], 'Age')

# Note: R-Squared and F1-Score are not available, because their batch based values are misleading.

# For some reason, Sensitivity and Specificity are not available either. Might implement later.

self.dict\_losses\_K = {'MSE': MeanSquaredError(name='MSE'),

'Binary-Crossentropy': BinaryCrossentropy(name='Binary-Crossentropy')}

self.dict\_metrics\_K = {'R-Squared': RSquare(name='R-Squared', y\_shape=(1,)),

'RMSE': RootMeanSquaredError(name='RMSE'),

'F1-Score': F1Score(name='F1-Score', num\_classes=1, dtype=tf.float32),

'ROC-AUC': AUC(curve='ROC', name='ROC-AUC'),

'PR-AUC': AUC(curve='PR', name='PR-AUC'),

'Binary-Accuracy': BinaryAccuracy(name='Binary-Accuracy'),

'Precision': Precision(name='Precision'),

'Recall': Recall(name='Recall'),

'True-Positives': TruePositives(name='True-Positives'),

'False-Positives': FalsePositives(name='False-Positives'),

'False-Negatives': FalseNegatives(name='False-Negatives'),

'True-Negatives': TrueNegatives(name='True-Negatives')}

# Metrics

self.prediction\_type = self.dict\_prediction\_types[target]

self.loss\_name = self.dict\_losses\_names[self.prediction\_type]

self.loss\_function = self.dict\_losses\_K[self.loss\_name]

self.main\_metric\_name = self.dict\_main\_metrics\_names\_K[target]

self.main\_metric\_mode = self.main\_metrics\_modes[self.main\_metric\_name]

self.main\_metric = self.dict\_metrics\_K[self.main\_metric\_name]

self.metrics\_names = [self.main\_metric\_name]

self.metrics = [self.dict\_metrics\_K[metric\_name] for metric\_name in self.metrics\_names]

# Optimizers

self.optimizers = {'Adam': Adam, 'RMSprop': RMSprop, 'Adadelta': Adadelta}

# Model

self.model = None

@staticmethod

def \_append\_ext(fn):

return fn + ".jpg"

def \_load\_data\_features(self):

for fold in self.folds:

self.DATA\_FEATURES[fold] = pd.read\_csv(

self.path\_data + 'data-features\_' + self.organ + '\_' + self.view + '\_' + self.transformation + '\_' +

self.dict\_target\_to\_ids[self.target] + '\_' + fold + '\_' + self.outer\_fold + '.csv')

for col\_name in self.id\_vars:

self.DATA\_FEATURES[fold][col\_name] = self.DATA\_FEATURES[fold][col\_name].astype(str)

self.DATA\_FEATURES[fold].set\_index('id', drop=False, inplace=True)

def \_take\_subset\_to\_debug(self):

for fold in self.folds:

# use +1 or +2 to test the leftovers pipeline

leftovers\_extra = {'train': 0, 'val': 1, 'test': 2}

n\_batches = 2

n\_limit\_fold = leftovers\_extra[fold] + self.batch\_size \* n\_batches

self.DATA\_FEATURES[fold] = self.DATA\_FEATURES[fold].iloc[:n\_limit\_fold, :]

def \_generate\_generators(self, DATA\_FEATURES):

GENERATORS = {}

for fold in self.folds:

# do not generate a generator if there are no samples (can happen for leftovers generators)

if fold not in DATA\_FEATURES.keys():

continue

# parameters

training\_mode = True if self.mode == 'model\_training' else False

if (fold == 'train') & (self.mode == 'model\_training') & \

(self.organ + '\_' + self.view not in self.organsviews\_not\_to\_augment):

data\_augmentation = True

else:

data\_augmentation = False

# define batch size for testing: data is split between a part that fits in batches, and leftovers

if self.mode == 'model\_testing':

if self.organ + '\_' + self.view in self.left\_right\_organs\_views:

n\_samples = len(DATA\_FEATURES[fold].index) \* 2

else:

n\_samples = len(DATA\_FEATURES[fold].index)

batch\_size\_fold = min(self.batch\_size, n\_samples)

else:

batch\_size\_fold = self.batch\_size

if (fold == 'train') & (self.mode == 'model\_training'):

n\_samples\_per\_subepoch = self.n\_samples\_per\_subepoch

else:

n\_samples\_per\_subepoch = None

# generator

GENERATORS[fold] = \

MyImageDataGenerator(target=self.target, organ=self.organ, view=self.view,

data\_features=DATA\_FEATURES[fold], n\_samples\_per\_subepoch=n\_samples\_per\_subepoch,

batch\_size=batch\_size\_fold, training\_mode=training\_mode,

side\_predictors=self.side\_predictors, dir\_images=self.dir\_images,

images\_width=self.image\_width, images\_height=self.image\_height,

data\_augmentation=data\_augmentation,

data\_augmentation\_factor=self.data\_augmentation\_factor, seed=self.seed)

return GENERATORS

def \_generate\_class\_weights(self):

if self.dict\_prediction\_types[self.target] == 'binary':

self.class\_weights = {}

counts = self.DATA\_FEATURES['train'][self.target + '\_raw'].value\_counts()

n\_total = counts.sum()

# weighting the samples for each class inversely proportional to their prevalence, with order of magnitude 1

for i in counts.index.values:

self.class\_weights[i] = n\_total / (counts.loc[i] \* len(counts.index))

def \_generate\_cnn(self):

# define the arguments

# take special initial weights for EfficientNetB7 (better)

if (self.architecture == 'EfficientNetB7') & (self.keras\_weights == 'imagenet'):

w = 'noisy-student'

else:

w = self.keras\_weights

kwargs = {"include\_top": False, "weights": w, "input\_shape": (self.image\_width, self.image\_height, 3)}

if self.architecture in ['ResNet50', 'ResNet101', 'ResNet152', 'ResNet50V2', 'ResNet101V2', 'ResNet152V2',

'ResNeXt50', 'ResNeXt101']:

import tensorflow.keras

kwargs.update(

{"backend": tensorflow.keras.backend, "layers": tensorflow.keras.layers,

"models": tensorflow.keras.models, "utils": tensorflow.keras.utils})

# load the architecture builder

if self.architecture == 'VGG16':

from tensorflow.keras.applications.vgg16 import VGG16 as ModelBuilder

elif self.architecture == 'VGG19':

from tensorflow.keras.applications.vgg19 import VGG19 as ModelBuilder

elif self.architecture == 'DenseNet121':

from tensorflow.keras.applications.densenet import DenseNet121 as ModelBuilder

elif self.architecture == 'DenseNet169':

from tensorflow.keras.applications.densenet import DenseNet169 as ModelBuilder

elif self.architecture == 'DenseNet201':

from tensorflow.keras.applications.densenet import DenseNet201 as ModelBuilder

elif self.architecture == 'Xception':

from tensorflow.keras.applications.xception import Xception as ModelBuilder

elif self.architecture == 'InceptionV3':

from tensorflow.keras.applications.inception\_v3 import InceptionV3 as ModelBuilder

elif self.architecture == 'InceptionResNetV2':

from tensorflow.keras.applications.inception\_resnet\_v2 import InceptionResNetV2 as ModelBuilder

elif self.architecture == 'ResNet50':

from keras\_applications.resnet import ResNet50 as ModelBuilder

elif self.architecture == 'ResNet101':

from keras\_applications.resnet import ResNet101 as ModelBuilder

elif self.architecture == 'ResNet152':

from keras\_applications.resnet import ResNet152 as ModelBuilder

elif self.architecture == 'ResNet50V2':

from keras\_applications.resnet\_v2 import ResNet50V2 as ModelBuilder

elif self.architecture == 'ResNet101V2':

from keras\_applications.resnet\_v2 import ResNet101V2 as ModelBuilder

elif self.architecture == 'ResNet152V2':

from keras\_applications.resnet\_v2 import ResNet152V2 as ModelBuilder

elif self.architecture == 'ResNeXt50':

from keras\_applications.resnext import ResNeXt50 as ModelBuilder

elif self.architecture == 'ResNeXt101':

from keras\_applications.resnext import ResNeXt101 as ModelBuilder

elif self.architecture == 'EfficientNetB7':

from efficientnet.tfkeras import EfficientNetB7 as ModelBuilder

# The following model have a fixed input size requirement

elif self.architecture == 'NASNetMobile':

from tensorflow.keras.applications.nasnet import NASNetMobile as ModelBuilder

elif self.architecture == 'NASNetLarge':

from tensorflow.keras.applications.nasnet import NASNetLarge as ModelBuilder

elif self.architecture == 'MobileNet':

from tensorflow.keras.applications.mobilenet import MobileNet as ModelBuilder

elif self.architecture == 'MobileNetV2':

from tensorflow.keras.applications.mobilenet\_v2 import MobileNetV2 as ModelBuilder

else:

print('Architecture does not exist.')

sys.exit(1)

# build the model's base

cnn = ModelBuilder(\*\*kwargs)

x = cnn.output

# complete the model's base

if self.architecture in ['VGG16', 'VGG19']:

x = Flatten()(x)

x = Dense(4096, activation='relu', kernel\_regularizer=regularizers.l2(self.weight\_decay))(x)

x = Dropout(self.dropout\_rate)(x)

x = Dense(4096, activation='relu', kernel\_regularizer=regularizers.l2(self.weight\_decay))(x)

x = Dropout(self.dropout\_rate)(x)

else:

x = GlobalAveragePooling2D()(x)

if self.architecture == 'EfficientNetB7':

x = Dropout(self.dropout\_rate)(x)

cnn\_output = x

return cnn.input, cnn\_output

def \_generate\_side\_nn(self):

side\_nn = Sequential()

side\_nn.add(Dense(16, input\_dim=len(self.side\_predictors), activation="relu",

kernel\_regularizer=regularizers.l2(self.weight\_decay)))

return side\_nn.input, side\_nn.output

def \_complete\_architecture(self, cnn\_input, cnn\_output, side\_nn\_input, side\_nn\_output):

x = concatenate([cnn\_output, side\_nn\_output])

x = Dropout(self.dropout\_rate)(x)

for n in [int(self.n\_fc\_nodes \* (2 \*\* (2 \* (self.n\_fc\_layers - 1 - i)))) for i in range(self.n\_fc\_layers)]:

x = Dense(n, activation='relu', kernel\_regularizer=regularizers.l2(self.weight\_decay))(x)

# scale the dropout proportionally to the number of nodes in a layer. No dropout for the last layers

if n > 16:

x = Dropout(self.dropout\_rate \* n / 1024)(x)

predictions = Dense(1, activation=self.dict\_final\_activations[self.prediction\_type],

kernel\_regularizer=regularizers.l2(self.weight\_decay))(x)

self.model = Model(inputs=[cnn\_input, side\_nn\_input], outputs=predictions)

def \_generate\_architecture(self):

cnn\_input, cnn\_output = self.\_generate\_cnn()

side\_nn\_input, side\_nn\_output = self.\_generate\_side\_nn()

self.\_complete\_architecture(cnn\_input=cnn\_input, cnn\_output=cnn\_output, side\_nn\_input=side\_nn\_input,

side\_nn\_output=side\_nn\_output)

def \_load\_model\_weights(self):

try:

self.model.load\_weights(self.path\_load\_weights)

except (FileNotFoundError, TypeError):

# load backup weights if the main weights are corrupted

try:

self.model.load\_weights(self.path\_load\_weights.replace('model-weights', 'backup-model-weights'))

except FileNotFoundError:

print('Error. No file was found. imagenet weights should have been used. Bug somewhere.')

sys.exit(1)

@staticmethod

def clean\_exit():

# exit

print('\nDone.\n')

print('Killing JOB PID with kill...')

os.system('touch ../eo/' + os.environ['SLURM\_JOBID'])

os.system('kill ' + str(os.getpid()))

time.sleep(60)

print('Escalating to kill JOB PID with kill -9...')

os.system('kill -9 ' + str(os.getpid()))

time.sleep(60)

print('Escalating to kill JOB ID')

os.system('scancel ' + os.environ['SLURM\_JOBID'])

time.sleep(60)

print('Everything failed to kill the job. Hanging there until hitting walltime...')

class Training(DeepLearning):

"""

Class to train CNN models:

- Generates the architecture

- Loads the best last weights so that a model can be trained over several jobs

- Generates the callbacks

- Compiles the model

- Trains the model

"""

def \_\_init\_\_(self, target=None, organ=None, view=None, transformation=None, architecture=None, n\_fc\_layers=None,

n\_fc\_nodes=None, optimizer=None, learning\_rate=None, weight\_decay=None, dropout\_rate=None,

data\_augmentation\_factor=None, outer\_fold=None, debug\_mode=False, transfer\_learning=None,

continue\_training=True, display\_full\_metrics=True):

# parameters

DeepLearning.\_\_init\_\_(self, target, organ, view, transformation, architecture, n\_fc\_layers, n\_fc\_nodes,

optimizer, learning\_rate, weight\_decay, dropout\_rate, data\_augmentation\_factor,

debug\_mode)

self.outer\_fold = outer\_fold

self.version = self.version + '\_' + str(outer\_fold)

# NNet's architecture's weights

self.continue\_training = continue\_training

self.transfer\_learning = transfer\_learning

self.list\_parameters\_to\_match = ['organ', 'transformation', 'view']

# dict to decide in which order targets should be used when trying to transfer weight from a similar model

self.dict\_alternative\_targets\_for\_transfer\_learning = {'Age': ['Age', 'Sex'], 'Sex': ['Sex', 'Age']}

# Generators

self.folds = ['train', 'val']

self.mode = 'model\_training'

self.class\_weights = None

self.GENERATORS = None

# Metrics

self.baseline\_performance = None

if display\_full\_metrics:

self.metrics\_names = self.dict\_metrics\_names\_K[self.prediction\_type]

# Model

self.path\_load\_weights = self.path\_data + 'model-weights\_' + self.version + '.h5'

if debug\_mode:

self.path\_save\_weights = self.path\_data + 'model-weights-debug.h5'

else:

self.path\_save\_weights = self.path\_data + 'model-weights\_' + self.version + '.h5'

self.n\_epochs\_max = 100000

self.callbacks = None

# Load and preprocess the data, build the generators

def data\_preprocessing(self):

self.\_load\_data\_features()

if self.debug\_mode:

self.\_take\_subset\_to\_debug()

self.\_generate\_class\_weights()

self.GENERATORS = self.\_generate\_generators(self.DATA\_FEATURES)

# Determine which weights to load, if any.

def \_weights\_for\_transfer\_learning(self):

print('Looking for models to transfer weights from...')

# define parameters

parameters = self.\_version\_to\_parameters(self.version)

# continue training if possible

if self.continue\_training and os.path.exists(self.path\_load\_weights):

print('Loading the weights from the model\'s previous training iteration.')

return

# Initialize the weights using other the weights from other successful hyperparameters combinations

if self.transfer\_learning == 'hyperparameters':

# Check if the same model with other hyperparameters have already been trained. Pick the best for transfer.

params = self.version.split('\_')

params\_tl\_idx = \

[i for i in range(len(names\_model\_parameters))

if any(names\_model\_parameters[i] == p for p in

['optimizer', 'learning\_rate', 'weight\_decay', 'dropout\_rate', 'data\_augmentation\_factor'])]

for idx in params\_tl\_idx:

params[idx] = '\*'

versions = '../eo/MI02\_' + '\_'.join(params) + '.out'

files = glob.glob(versions)

if self.main\_metric\_mode == 'min':

best\_perf = np.Inf

else:

best\_perf = -np.Inf

for file in files:

hand = open(file, 'r')

# find best last performance

final\_improvement\_line = None

baseline\_performance\_line = None

for line in hand:

line = line.rstrip()

if re.search('Baseline validation ' + self.main\_metric\_name + ' = ', line):

baseline\_performance\_line = line

if re.search('val\_' + self.main\_metric\_name + ' improved from', line):

final\_improvement\_line = line

hand.close()

if final\_improvement\_line is not None:

perf = float(final\_improvement\_line.split(' ')[7].replace(',', ''))

elif baseline\_performance\_line is not None:

perf = float(baseline\_performance\_line.split(' ')[-1])

else:

continue

# Keep track of the file with the best performance

if self.main\_metric\_mode == 'min':

update = perf < best\_perf

else:

update = perf > best\_perf

if update:

best\_perf = perf

self.path\_load\_weights = \

file.replace('../eo/', self.path\_data).replace('MI02', 'model-weights').replace('.out', '.h5')

if best\_perf not in [-np.Inf, np.Inf]:

print('Transfering the weights from: ' + self.path\_load\_weights + ', with ' + self.main\_metric\_name +

' = ' + str(best\_perf))

return

# Initialize the weights based on models trained on different datasets, ranked by similarity

if self.transfer\_learning == 'datasets':

while True:

# print('Matching models for the following criterias:');

# print(['architecture', 'target'] + list\_parameters\_to\_match)

# start by looking for models trained on the same target, then move to other targets

for target\_to\_load in self.dict\_alternative\_targets\_for\_transfer\_learning[parameters['target']]:

# print('Target used: ' + target\_to\_load)

parameters\_to\_match = parameters.copy()

parameters\_to\_match['target'] = target\_to\_load

# load the ranked performances table to select the best performing model among the similar

# models available

path\_performances\_to\_load = self.path\_data + 'PERFORMANCES\_ranked\_' + \

parameters\_to\_match['target'] + '\_' + 'val' + '.csv'

try:

Performances = pd.read\_csv(path\_performances\_to\_load)

Performances['organ'] = Performances['organ'].astype(str)

except FileNotFoundError:

# print("Could not load the file: " + path\_performances\_to\_load)

break

# iteratively get rid of models that are not similar enough, based on the list

for parameter in ['architecture', 'target'] + self.list\_parameters\_to\_match:

Performances = Performances[Performances[parameter] == parameters\_to\_match[parameter]]

# if at least one model is similar enough, load weights from the best of them

if len(Performances.index) != 0:

self.path\_load\_weights = self.path\_data + 'model-weights\_' + Performances['version'][0] + '.h5'

self.keras\_weights = None

print('transfering the weights from: ' + self.path\_load\_weights)

return

# if no similar model was found, try again after getting rid of the last selection criteria

if len(self.list\_parameters\_to\_match) == 0:

print('No model found for transfer learning.')

break

self.list\_parameters\_to\_match.pop()

# Otherwise use imagenet weights to initialize

print('Using imagenet weights.')

# using string instead of None for path to not ge

self.path\_load\_weights = None

self.keras\_weights = 'imagenet'

def \_compile\_model(self):

# if learning rate was reduced with success according to logger, start with this reduced learning rate

if self.path\_load\_weights is not None:

path\_logger = self.path\_load\_weights.replace('model-weights', 'logger').replace('.h5', '.csv')

else:

path\_logger = self.path\_data + 'logger\_' + self.version + '.csv'

if os.path.exists(path\_logger):

try:

logger = pd.read\_csv(path\_logger)

best\_log = \

logger[logger['val\_' + self.main\_metric\_name] == logger['val\_' + self.main\_metric\_name].max()]

lr = best\_log['learning\_rate'].values[0]

except pd.errors.EmptyDataError:

os.remove(path\_logger)

lr = self.learning\_rate

else:

lr = self.learning\_rate

self.model.compile(optimizer=self.optimizers[self.optimizer](lr=lr, clipnorm=1.0), loss=self.loss\_function,

metrics=self.metrics)

def \_compute\_baseline\_performance(self):

# calculate initial val\_loss value

if self.continue\_training:

idx\_metric\_name = ([self.loss\_name] + self.metrics\_names).index(self.main\_metric\_name)

baseline\_perfs = self.model.evaluate(self.GENERATORS['val'], steps=self.GENERATORS['val'].steps)

self.baseline\_performance = baseline\_perfs[idx\_metric\_name]

elif self.main\_metric\_mode == 'min':

self.baseline\_performance = np.Inf

else:

self.baseline\_performance = -np.Inf

print('Baseline validation ' + self.main\_metric\_name + ' = ' + str(self.baseline\_performance))

def \_define\_callbacks(self):

if self.debug\_mode:

path\_logger = self.path\_data + 'logger-debug.csv'

append = False

else:

path\_logger = self.path\_data + 'logger\_' + self.version + '.csv'

append = self.continue\_training

csv\_logger = MyCSVLogger(path\_logger, separator=',', append=append)

model\_checkpoint\_backup = MyModelCheckpoint(self.path\_save\_weights.replace('model-weights',

'backup-model-weights'),

monitor='val\_' + self.main\_metric.name,

baseline=self.baseline\_performance, verbose=1, save\_best\_only=True,

save\_weights\_only=True, mode=self.main\_metric\_mode,

save\_freq='epoch')

model\_checkpoint = MyModelCheckpoint(self.path\_save\_weights,

monitor='val\_' + self.main\_metric.name, baseline=self.baseline\_performance,

verbose=1, save\_best\_only=True, save\_weights\_only=True,

mode=self.main\_metric\_mode, save\_freq='epoch')

patience\_reduce\_lr = min(7, 3 \* self.GENERATORS['train'].n\_subepochs\_per\_epoch)

reduce\_lr\_on\_plateau = ReduceLROnPlateau(monitor='loss', factor=0.5, patience=patience\_reduce\_lr, verbose=1,

mode='min', min\_delta=0, cooldown=0, min\_lr=0)

early\_stopping = EarlyStopping(monitor='val\_' + self.main\_metric.name, min\_delta=0, patience=15, verbose=0,

mode=self.main\_metric\_mode, baseline=self.baseline\_performance,

restore\_best\_weights=True)

self.callbacks = [csv\_logger, model\_checkpoint\_backup, model\_checkpoint, early\_stopping, reduce\_lr\_on\_plateau]

def build\_model(self):

self.\_weights\_for\_transfer\_learning()

self.\_generate\_architecture()

# Load weights if possible

try:

load\_weights = True if os.path.exists(self.path\_load\_weights) else False

except TypeError:

load\_weights = False

if load\_weights:

self.\_load\_model\_weights()

else:

# save transferred weights as default, in case no better weights are found

self.model.save\_weights(self.path\_save\_weights.replace('model-weights', 'backup-model-weights'))

self.model.save\_weights(self.path\_save\_weights)

self.\_compile\_model()

self.\_compute\_baseline\_performance()

self.\_define\_callbacks()

def train\_model(self):

# garbage collector

\_ = gc.collect()

# use more verbose when debugging

verbose = 1 if self.debug\_mode else 2

# train the model

self.model.fit(self.GENERATORS['train'], steps\_per\_epoch=self.GENERATORS['train'].steps,

validation\_data=self.GENERATORS['val'], validation\_steps=self.GENERATORS['val'].steps,

shuffle=False, use\_multiprocessing=False, workers=self.n\_cpus, epochs=self.n\_epochs\_max,

class\_weight=self.class\_weights, callbacks=self.callbacks, verbose=verbose)

class PredictionsGenerate(DeepLearning):

"""

Generates the predictions for each model.

Unscales the predictions.

"""

def \_\_init\_\_(self, target=None, organ=None, view=None, transformation=None, architecture=None, n\_fc\_layers=None,

n\_fc\_nodes=None, optimizer=None, learning\_rate=None, weight\_decay=None, dropout\_rate=None,

data\_augmentation\_factor=None, outer\_fold=None, debug\_mode=False):

# Initialize parameters

DeepLearning.\_\_init\_\_(self, target, organ, view, transformation, architecture, n\_fc\_layers, n\_fc\_nodes,

optimizer, learning\_rate, weight\_decay, dropout\_rate, data\_augmentation\_factor,

debug\_mode)

self.outer\_fold = outer\_fold

self.mode = 'model\_testing'

# Define dictionaries attributes for data, generators and predictions

self.DATA\_FEATURES\_BATCH = {}

self.DATA\_FEATURES\_LEFTOVERS = {}

self.GENERATORS\_BATCH = None

self.GENERATORS\_LEFTOVERS = None

self.PREDICTIONS = {}

def \_split\_batch\_leftovers(self):

# split the samples into two groups: what can fit into the batch size, and the leftovers.

for fold in self.folds:

n\_leftovers = len(self.DATA\_FEATURES[fold].index) % self.n\_ids\_batch

if n\_leftovers > 0:

self.DATA\_FEATURES\_BATCH[fold] = self.DATA\_FEATURES[fold].iloc[:-n\_leftovers]

self.DATA\_FEATURES\_LEFTOVERS[fold] = self.DATA\_FEATURES[fold].tail(n\_leftovers)

else:

self.DATA\_FEATURES\_BATCH[fold] = self.DATA\_FEATURES[fold] # special case for syntax if no leftovers

if fold in self.DATA\_FEATURES\_LEFTOVERS.keys():

del self.DATA\_FEATURES\_LEFTOVERS[fold]

def \_generate\_outerfold\_predictions(self):

# prepare unscaling

if self.target in self.targets\_regression:

mean\_train = self.DATA\_FEATURES['train'][self.target + '\_raw'].mean()

std\_train = self.DATA\_FEATURES['train'][self.target + '\_raw'].std()

else:

mean\_train, std\_train = None, None

# Generate predictions

for fold in self.folds:

print('Predicting samples from fold ' + fold + '.')

print(str(len(self.DATA\_FEATURES[fold].index)) + ' samples to predict.')

print('Predicting batches: ' + str(len(self.DATA\_FEATURES\_BATCH[fold].index)) + ' samples.')

pred\_batch = self.model.predict(self.GENERATORS\_BATCH[fold], steps=self.GENERATORS\_BATCH[fold].steps,

verbose=1)

if fold in self.GENERATORS\_LEFTOVERS.keys():

print('Predicting leftovers: ' + str(len(self.DATA\_FEATURES\_LEFTOVERS[fold].index)) + ' samples.')

pred\_leftovers = self.model.predict(self.GENERATORS\_LEFTOVERS[fold],

steps=self.GENERATORS\_LEFTOVERS[fold].steps, verbose=1)

pred\_full = np.concatenate((pred\_batch, pred\_leftovers)).squeeze()

else:

pred\_full = pred\_batch.squeeze()

print('Predicted a total of ' + str(len(pred\_full)) + ' samples.')

# take the average between left and right predictions for paired organs

if self.organ + '\_' + self.view in self.left\_right\_organs\_views:

pred\_full = np.mean(pred\_full.reshape(-1, 2), axis=1)

# unscale predictions

if self.target in self.targets\_regression:

pred\_full = pred\_full \* std\_train + mean\_train

# format the dataframe

self.DATA\_FEATURES[fold]['pred'] = pred\_full

self.PREDICTIONS[fold] = self.DATA\_FEATURES[fold]

self.PREDICTIONS[fold]['id'] = [ID.replace('.jpg', '') for ID in self.PREDICTIONS[fold]['id']]

def \_generate\_predictions(self):

self.path\_load\_weights = self.path\_data + 'model-weights\_' + self.version + '\_' + self.outer\_fold + '.h5'

self.\_load\_data\_features()

if self.debug\_mode:

self.\_take\_subset\_to\_debug()

self.\_load\_model\_weights()

self.\_split\_batch\_leftovers()

# generate the generators

self.GENERATORS\_BATCH = self.\_generate\_generators(DATA\_FEATURES=self.DATA\_FEATURES\_BATCH)

if self.DATA\_FEATURES\_LEFTOVERS is not None:

self.GENERATORS\_LEFTOVERS = self.\_generate\_generators(DATA\_FEATURES=self.DATA\_FEATURES\_LEFTOVERS)

self.\_generate\_outerfold\_predictions()

def \_format\_predictions(self):

for fold in self.folds:

perf\_fun = self.dict\_metrics\_sklearn[self.dict\_main\_metrics\_names[self.target]]

perf = perf\_fun(self.PREDICTIONS[fold][self.target + '\_raw'], self.PREDICTIONS[fold]['pred'])

print('The ' + fold + ' performance is: ' + str(perf))

# format the predictions

self.PREDICTIONS[fold].index.name = 'column\_names'

self.PREDICTIONS[fold] = self.PREDICTIONS[fold][['id', 'outer\_fold', 'pred']]

def generate\_predictions(self):

self.\_generate\_architecture()

self.\_generate\_predictions()

self.\_format\_predictions()

def save\_predictions(self):

for fold in self.folds:

self.PREDICTIONS[fold].to\_csv(self.path\_data + 'Predictions\_instances\_' + self.version + '\_' + fold + '\_'

+ self.outer\_fold + '.csv', index=False)

class PredictionsConcatenate(Basics):

"""

Concatenates the predictions coming from the different cross validation folds.

"""

def \_\_init\_\_(self, target=None, organ=None, view=None, transformation=None, architecture=None, n\_fc\_layers=None,

n\_fc\_nodes=None, optimizer=None, learning\_rate=None, weight\_decay=None, dropout\_rate=None,

data\_augmentation\_factor=None):

# Initialize parameters

Basics.\_\_init\_\_(self)

self.version = target + '\_' + organ + '\_' + view + '\_' + transformation + '\_' + architecture + '\_' + \

n\_fc\_layers + '\_' + n\_fc\_nodes + '\_' + optimizer + '\_' + learning\_rate + '\_' + weight\_decay + \

'\_' + dropout\_rate + '\_' + data\_augmentation\_factor

# Define dictionaries attributes for data, generators and predictions

self.PREDICTIONS = {}

def concatenate\_predictions(self):

for fold in self.folds:

for outer\_fold in self.outer\_folds:

Predictions\_fold = pd.read\_csv(self.path\_data + 'Predictions\_instances\_' + self.version + '\_' + fold +

'\_' + outer\_fold + '.csv')

if fold in self.PREDICTIONS.keys():

self.PREDICTIONS[fold] = pd.concat([self.PREDICTIONS[fold], Predictions\_fold])

else:

self.PREDICTIONS[fold] = Predictions\_fold

def save\_predictions(self):

for fold in self.folds:

self.PREDICTIONS[fold].to\_csv(self.path\_data + 'Predictions\_instances\_' + self.version + '\_' + fold +

'.csv', index=False)

class PredictionsMerge(Basics):

"""

Merges the predictions from all models into a unified dataframe.

"""

def \_\_init\_\_(self, target=None, fold=None):

Basics.\_\_init\_\_(self)

# Define dictionaries attributes for data, generators and predictions

self.target = target

self.fold = fold

self.data\_features = None

self.list\_models = None

self.Predictions\_df\_previous = None

self.Predictions\_df = None

def \_load\_data\_features(self):

self.data\_features = pd.read\_csv(self.path\_data + 'data-features\_instances.csv',

usecols=self.id\_vars + self.demographic\_vars)

for var in self.id\_vars:

self.data\_features[var] = self.data\_features[var].astype(str)

self.data\_features.set\_index('id', drop=False, inplace=True)

self.data\_features.index.name = 'column\_names'

def \_preprocess\_data\_features(self):

# For the training set, each sample is predicted n\_CV\_outer\_folds times, so prepare a larger dataframe

if self.fold == 'train':

df\_all\_folds = None

for outer\_fold in self.outer\_folds:

df\_fold = self.data\_features.copy()

df\_all\_folds = df\_fold if outer\_fold == self.outer\_folds[0] else df\_all\_folds.append(df\_fold)

self.data\_features = df\_all\_folds

def \_load\_previous\_merged\_predictions(self):

if os.path.exists(self.path\_data + 'PREDICTIONS\_withoutEnsembles\_instances\_' + self.target + '\_' + self.fold +

'.csv'):

self.Predictions\_df\_previous = pd.read\_csv(self.path\_data + 'PREDICTIONS\_withoutEnsembles\_instances\_' +

self.target + '\_' + self.fold + '.csv')

self.Predictions\_df\_previous.drop(columns=['eid', 'instance'] + self.demographic\_vars, inplace=True)

def \_list\_models(self):

# generate list of predictions that will be integrated in the Predictions dataframe

self.list\_models = glob.glob(self.path\_data + 'Predictions\_instances\_' + self.target + '\_\*\_' + self.fold +

'.csv')

# get rid of ensemble models and models already merged

self.list\_models = [model for model in self.list\_models if ('\*' not in model)]

if self.Predictions\_df\_previous is not None:

self.list\_models = \

[model for model in self.list\_models

if ('pred\_' + '\_'.join(model.split('\_')[2:-1]) not in self.Predictions\_df\_previous.columns)]

self.list\_models.sort()

def preprocessing(self):

self.\_load\_data\_features()

self.\_preprocess\_data\_features()

self.\_load\_previous\_merged\_predictions()

self.\_list\_models()

def merge\_predictions(self):

# merge the predictions

print('There are ' + str(len(self.list\_models)) + ' models to merge.')

i = 0

# define subgroups to accelerate merging process

list\_subgroups = list(set(['\_'.join(model.split('\_')[3:7]) for model in self.list\_models]))

for subgroup in list\_subgroups:

print('Merging models from the subgroup ' + subgroup)

models\_subgroup = [model for model in self.list\_models if subgroup in model]

Predictions\_subgroup = None

# merge the models one by one

for file\_name in models\_subgroup:

i += 1

version = '\_'.join(file\_name.split('\_')[2:-1])

if self.Predictions\_df\_previous is not None and \

'pred\_' + version in self.Predictions\_df\_previous.columns:

print('The model ' + version + ' has already been merged before.')

else:

print('Merging the ' + str(i) + 'th model: ' + version)

# load csv and format the predictions

prediction = pd.read\_csv(self.path\_data + file\_name)

print('raw prediction\'s shape: ' + str(prediction.shape))

for var in ['id', 'outer\_fold']:

prediction[var] = prediction[var].apply(str)

prediction.rename(columns={'pred': 'pred\_' + version}, inplace=True)

# merge data frames

if Predictions\_subgroup is None:

Predictions\_subgroup = prediction

elif self.fold == 'train':

Predictions\_subgroup = Predictions\_subgroup.merge(prediction, how='outer',

on=['id', 'outer\_fold'])

else:

prediction.drop(['outer\_fold'], axis=1, inplace=True)

# not supported for panda version > 0.23.4 for now

Predictions\_subgroup = Predictions\_subgroup.merge(prediction, how='outer', on=['id'])

# merge group predictions data frames

if self.fold != 'train':

Predictions\_subgroup.drop(['outer\_fold'], axis=1, inplace=True)

if Predictions\_subgroup is not None:

if self.Predictions\_df is None:

self.Predictions\_df = Predictions\_subgroup

elif self.fold == 'train':

self.Predictions\_df = self.Predictions\_df.merge(Predictions\_subgroup, how='outer',

on=['id', 'outer\_fold'])

else:

# not supported for panda version > 0.23.4 for now

self.Predictions\_df = self.Predictions\_df.merge(Predictions\_subgroup, how='outer', on=['id'])

print('Predictions\_df\'s shape: ' + str(self.Predictions\_df.shape))

# garbage collector

gc.collect()

# Merge with the previously merged predictions

if (self.Predictions\_df\_previous is not None) & (self.Predictions\_df is not None):

if self.fold == 'train':

self.Predictions\_df = self.Predictions\_df\_previous.merge(self.Predictions\_df, how='outer',

on=['id', 'outer\_fold'])

else:

self.Predictions\_df.drop(columns=['outer\_fold'], inplace=True)

# not supported for panda version > 0.23.4 for now

self.Predictions\_df = self.Predictions\_df\_previous.merge(self.Predictions\_df, how='outer', on=['id'])

self.Predictions\_df\_previous = None

elif self.Predictions\_df is None:

print('No new models to merge. Exiting.')

print('Done.')

sys.exit(0)

# Reorder the columns alphabetically

pred\_versions = [col for col in self.Predictions\_df.columns if 'pred\_' in col]

pred\_versions.sort()

id\_cols = ['id', 'outer\_fold'] if self.fold == 'train' else ['id']

self.Predictions\_df = self.Predictions\_df[id\_cols + pred\_versions]

def postprocessing(self):

# get rid of useless rows in data\_features before merging to keep the memory requirements as low as possible

self.data\_features = self.data\_features[self.data\_features['id'].isin(self.Predictions\_df['id'].values)]

# merge data\_features and predictions

if self.fold == 'train':

print('Starting to merge a massive dataframe')

self.Predictions\_df = self.data\_features.merge(self.Predictions\_df, how='outer', on=['id', 'outer\_fold'])

else:

# not supported for panda version > 0.23.4 for now

self.Predictions\_df = self.data\_features.merge(self.Predictions\_df, how='outer', on=['id'])

print('Merging done')

# remove rows for which no prediction is available (should be none)

subset\_cols = [col for col in self.Predictions\_df.columns if 'pred\_' in col]

self.Predictions\_df.dropna(subset=subset\_cols, how='all', inplace=True)

# Displaying the R2s

versions = [col.replace('pred\_', '') for col in self.Predictions\_df.columns if 'pred\_' in col]

r2s = []

for version in versions:

df = self.Predictions\_df[[self.target, 'pred\_' + version]].dropna()

r2s.append(r2\_score(df[self.target], df['pred\_' + version]))

R2S = pd.DataFrame({'version': versions, 'R2': r2s})

R2S.sort\_values(by='R2', ascending=False, inplace=True)

print('R2 for each model: ')

print(R2S)

def save\_merged\_predictions(self):

print('Writing the merged predictions...')

self.Predictions\_df.to\_csv(self.path\_data + 'PREDICTIONS\_withoutEnsembles\_instances\_' + self.target + '\_' +

self.fold + '.csv', index=False)

class PredictionsEids(Basics):

"""

Computes the average age prediction across samples from different instances for every participant.

(Scaled back to instance 0)

"""

def \_\_init\_\_(self, target=None, fold=None, debug\_mode=None):

Basics.\_\_init\_\_(self)

# Define dictionaries attributes for data, generators and predictions

self.target = target

self.fold = fold

self.debug\_mode = debug\_mode

self.Predictions = None

self.Predictions\_chunk = None

self.pred\_versions = None

self.res\_versions = None

self.target\_0s = None

self.Predictions\_eids = None

self.Predictions\_eids\_previous = None

self.pred\_versions\_previous = None

def preprocessing(self):

# Load predictions

self.Predictions = pd.read\_csv(

self.path\_data + 'PREDICTIONS\_withoutEnsembles\_instances\_' + self.target + '\_' + self.fold + '.csv')

self.Predictions.drop(columns=['id'], inplace=True)

self.Predictions['eid'] = self.Predictions['eid'].astype(str)

self.Predictions.index.name = 'column\_names'

self.pred\_versions = [col for col in self.Predictions.columns.values if 'pred\_' in col]

# Prepare target values on instance 0 as a reference

target\_0s = pd.read\_csv(self.path\_data + 'data-features\_eids.csv', usecols=['eid', self.target])

target\_0s['eid'] = target\_0s['eid'].astype(str)

target\_0s.set\_index('eid', inplace=True)

target\_0s = target\_0s[self.target]

target\_0s.name = 'target\_0'

target\_0s = target\_0s[self.Predictions['eid'].unique()]

self.Predictions = self.Predictions.merge(target\_0s, on='eid')

# Compute biological ages reported to target\_0

for pred in self.pred\_versions:

# Compute the biais of the predictions as a function of age

print('Generating residuals for model ' + pred.replace('pred\_', ''))

df\_model = self.Predictions[['Age', pred]]

df\_model.dropna(inplace=True)

if (len(df\_model.index)) > 0:

age = df\_model.loc[:, ['Age']]

res = df\_model['Age'] - df\_model[pred]

regr = LinearRegression()

regr.fit(age, res)

self.Predictions[pred.replace('pred\_', 'correction\_')] = regr.predict(self.Predictions[['Age']])

# Take the residuals bias into account when "translating" the prediction to instance 0

correction = self.Predictions['target\_0'] - self.Predictions[self.target] + \

regr.predict(self.Predictions[['Age']]) - regr.predict(self.Predictions[['target\_0']])

self.Predictions[pred] = self.Predictions[pred] + correction

self.Predictions[self.target] = self.Predictions['target\_0']

self.Predictions.drop(columns=['target\_0'], inplace=True)

self.Predictions.index.name = 'column\_names'

def processing(self):

if self.fold == 'train':

# Prepare template to which each model will be appended

Predictions = self.Predictions[['eid'] + self.demographic\_vars]

Predictions = Predictions.groupby('eid', as\_index=True).mean()

Predictions.index.name = 'column\_names'

Predictions['eid'] = Predictions.index.values

Predictions['instance'] = '\*'

Predictions['id'] = Predictions['eid'] + '\_\*'

self.Predictions\_eids = Predictions.copy()

self.Predictions\_eids['outer\_fold'] = -1

for i in range(self.n\_CV\_outer\_folds):

Predictions\_i = Predictions.copy()

Predictions\_i['outer\_fold'] = i

self.Predictions\_eids = self.Predictions\_eids.append(Predictions\_i)

# Append each model one by one because the folds are different

print(str(len(self.pred\_versions)) + ' models to compute.')

for pred\_version in self.pred\_versions:

if pred\_version in self.pred\_versions\_previous:

print(pred\_version.replace('pred\_', '') + ' had already been computed.')

else:

print("Computing results for version " + pred\_version.replace('pred\_', ''))

Predictions\_version = self.Predictions[['eid', pred\_version, 'outer\_fold']]

# Use placeholder for NaN in outer\_folds

Predictions\_version['outer\_fold'][Predictions\_version['outer\_fold'].isna()] = -1

Predictions\_version\_eids = Predictions\_version.groupby(['eid', 'outer\_fold'], as\_index=False).mean()

self.Predictions\_eids = self.Predictions\_eids.merge(Predictions\_version\_eids,

on=['eid', 'outer\_fold'], how='outer')

self.Predictions\_eids[of\_version] = self.Predictions\_eids['outer\_fold']

self.Predictions\_eids[of\_version][self.Predictions\_eids[of\_version] == -1] = np.nan

del Predictions\_version

\_ = gc.collect

self.Predictions\_eids.drop(columns=['outer\_fold'], inplace=True)

else:

self.Predictions\_eids = self.Predictions.groupby('eid').mean()

self.Predictions\_eids['eid'] = self.Predictions\_eids.index.values

self.Predictions\_eids['instance'] = '\*'

self.Predictions\_eids['id'] = self.Predictions\_eids['eid'].astype(str) + '\_' + \

self.Predictions\_eids['instance']

# Re-order the columns

self.Predictions\_eids = self.Predictions\_eids[self.id\_vars + self.demographic\_vars + self.pred\_versions]

def postprocessing(self):

# Displaying the R2s

versions = [col.replace('pred\_', '') for col in self.Predictions\_eids.columns if 'pred\_' in col]

r2s = []

for version in versions:

df = self.Predictions\_eids[[self.target, 'pred\_' + version]].dropna()

r2s.append(r2\_score(df[self.target], df['pred\_' + version]))

R2S = pd.DataFrame({'version': versions, 'R2': r2s})

R2S.sort\_values(by='R2', ascending=False, inplace=True)

print('R2 for each model: ')

print(R2S)

def \_generate\_single\_model\_predictions(self):

for pred\_version in self.pred\_versions:

path\_save = \

self.path\_data + 'Predictions\_eids\_' + '\_'.join(pred\_version.split('\_')[1:]) + '\_' + self.fold + '.csv'

# Generate only if does not exist already.

if not os.path.exists(path\_save):

Predictions\_version = self.Predictions\_eids[['id', 'outer\_fold', pred\_version]]

Predictions\_version.rename(columns={pred\_version: 'pred'}, inplace=True)

Predictions\_version.dropna(subset=['pred'], inplace=True)

Predictions\_version.to\_csv(path\_save, index=False)

def save\_predictions(self):

self.Predictions\_eids.to\_csv(self.path\_data + 'PREDICTIONS\_withoutEnsembles\_eids\_' + self.target + '\_' +

self.fold + '.csv', index=False)

# Generate and save files for every single model

self.\_generate\_single\_model\_predictions()

class PerformancesGenerate(Metrics):

"""

Computes the performances for each model.

"""

def \_\_init\_\_(self, target=None, organ=None, view=None, transformation=None, architecture=None, n\_fc\_layers=None,

n\_fc\_nodes=None, optimizer=None, learning\_rate=None, weight\_decay=None, dropout\_rate=None,

data\_augmentation\_factor=None, fold=None, pred\_type=None, debug\_mode=False):

Metrics.\_\_init\_\_(self)

self.target = target

self.organ = organ

self.view = view

self.transformation = transformation

self.architecture = architecture

self.n\_fc\_layers = n\_fc\_layers

self.n\_fc\_nodes = n\_fc\_nodes

self.optimizer = optimizer

self.learning\_rate = learning\_rate

self.weight\_decay = weight\_decay

self.dropout\_rate = dropout\_rate

self.data\_augmentation\_factor = data\_augmentation\_factor

self.fold = fold

self.pred\_type = pred\_type

if debug\_mode:

self.n\_bootstrap\_iterations = 3

else:

self.n\_bootstrap\_iterations = 1000

self.version = target + '\_' + organ + '\_' + view + '\_' + transformation + '\_' + architecture + '\_' + \

n\_fc\_layers + '\_' + n\_fc\_nodes + '\_' + optimizer + '\_' + learning\_rate + '\_' + weight\_decay + \

'\_' + dropout\_rate + '\_' + data\_augmentation\_factor

self.names\_metrics = self.dict\_metrics\_names[self.dict\_prediction\_types[target]]

self.data\_features = None

self.Predictions = None

self.PERFORMANCES = None

def \_preprocess\_data\_features\_predictions\_for\_performances(self):

# load dataset

data\_features = pd.read\_csv(self.path\_data + 'data-features\_' + self.pred\_type + '.csv',

usecols=['id', 'Sex', 'Age'])

# format data\_features to extract y

data\_features.rename(columns={self.target: 'y'}, inplace=True)

data\_features = data\_features[['id', 'y']]

data\_features['id'] = data\_features['id'].astype(str)

data\_features['id'] = data\_features['id']

data\_features.set\_index('id', drop=False, inplace=True)

data\_features.index.name = 'columns\_names'

self.data\_features = data\_features

def \_preprocess\_predictions\_for\_performances(self):

Predictions = pd.read\_csv(self.path\_data + 'Predictions\_' + self.pred\_type + '\_' + self.version + '\_' +

self.fold + '.csv')

Predictions['id'] = Predictions['id'].astype(str)

self.Predictions = Predictions.merge(self.data\_features, how='inner', on=['id'])

# Initialize performances dataframes and compute sample sizes

def \_initiate\_empty\_performances\_df(self):

# Define an empty performances dataframe to store the performances computed

row\_names = ['all'] + self.outer\_folds

col\_names\_sample\_sizes = ['N']

if self.target in self.targets\_binary:

col\_names\_sample\_sizes.extend(['N\_0', 'N\_1'])

col\_names = ['outer\_fold'] + col\_names\_sample\_sizes

col\_names.extend(self.names\_metrics)

performances = np.empty((len(row\_names), len(col\_names),))

performances.fill(np.nan)

performances = pd.DataFrame(performances)

performances.index = row\_names

performances.columns = col\_names

performances['outer\_fold'] = row\_names

# Convert float to int for sample sizes and some metrics.

for col\_name in col\_names\_sample\_sizes:

# need recent version of pandas to use type below. Otherwise nan cannot be int

performances[col\_name] = performances[col\_name].astype('Int64')

# compute sample sizes for the data frame

performances.loc['all', 'N'] = len(self.Predictions.index)

if self.target in self.targets\_binary:

performances.loc['all', 'N\_0'] = len(self.Predictions.loc[self.Predictions['y'] == 0].index)

performances.loc['all', 'N\_1'] = len(self.Predictions.loc[self.Predictions['y'] == 1].index)

for outer\_fold in self.outer\_folds:

performances.loc[outer\_fold, 'N'] = len(

self.Predictions.loc[self.Predictions['outer\_fold'] == int(outer\_fold)].index)

if self.target in self.targets\_binary:

performances.loc[outer\_fold, 'N\_0'] = len(

self.Predictions.loc[

(self.Predictions['outer\_fold'] == int(outer\_fold)) & (self.Predictions['y'] == 0)].index)

performances.loc[outer\_fold, 'N\_1'] = len(

self.Predictions.loc[

(self.Predictions['outer\_fold'] == int(outer\_fold)) & (self.Predictions['y'] == 1)].index)

# initialize the dataframes

self.PERFORMANCES = {}

for mode in self.modes:

self.PERFORMANCES[mode] = performances.copy()

# Convert float to int for sample sizes and some metrics.

for col\_name in self.PERFORMANCES[''].columns.values:

if any(metric in col\_name for metric in self.metrics\_displayed\_in\_int):

# need recent version of pandas to use type below. Otherwise nan cannot be int

self.PERFORMANCES[''][col\_name] = self.PERFORMANCES[''][col\_name].astype('Int64')

def preprocessing(self):

self.\_preprocess\_data\_features\_predictions\_for\_performances()

self.\_preprocess\_predictions\_for\_performances()

self.\_initiate\_empty\_performances\_df()

# Fill the columns for this model, outer\_fold by outer\_fold

def compute\_performances(self):

# fill it outer\_fold by outer\_fold

for outer\_fold in ['all'] + self.outer\_folds:

print('Calculating the performances for the outer fold ' + outer\_fold)

# Generate a subdataframe from the predictions table for each outerfold

if outer\_fold == 'all':

predictions\_fold = self.Predictions.copy()

else:

predictions\_fold = self.Predictions.loc[self.Predictions['outer\_fold'] == int(outer\_fold), :]

# if no samples are available for this fold, fill columns with nans

if len(predictions\_fold.index) == 0:

print('NO SAMPLES AVAILABLE FOR MODEL ' + self.version + ' IN OUTER\_FOLD ' + outer\_fold)

else:

# For binary classification, generate class prediction

if self.target in self.targets\_binary:

predictions\_fold\_class = predictions\_fold.copy()

predictions\_fold\_class['pred'] = predictions\_fold\_class['pred'].round()

else:

predictions\_fold\_class = None

# Fill the Performances dataframe metric by metric

for name\_metric in self.names\_metrics:

# print('Calculating the performance using the metric ' + name\_metric)

if name\_metric in self.metrics\_needing\_classpred:

predictions\_metric = predictions\_fold\_class

else:

predictions\_metric = predictions\_fold

metric\_function = self.dict\_metrics\_sklearn[name\_metric]

self.PERFORMANCES[''].loc[outer\_fold, name\_metric] = metric\_function(predictions\_metric['y'],

predictions\_metric['pred'])

self.PERFORMANCES['\_sd'].loc[outer\_fold, name\_metric] = \

self.\_bootstrap(predictions\_metric, metric\_function)[1]

self.PERFORMANCES['\_str'].loc[outer\_fold, name\_metric] = "{:.3f}".format(

self.PERFORMANCES[''].loc[outer\_fold, name\_metric]) + '+-' + "{:.3f}".format(

self.PERFORMANCES['\_sd'].loc[outer\_fold, name\_metric])

# calculate the fold sd (variance between the metrics values obtained on the different folds)

folds\_sd = self.PERFORMANCES[''].iloc[1:, :].std(axis=0)

for name\_metric in self.names\_metrics:

self.PERFORMANCES['\_str'].loc['all', name\_metric] = "{:.3f}".format(

self.PERFORMANCES[''].loc['all', name\_metric]) + '+-' + "{:.3f}".format(

folds\_sd[name\_metric]) + '+-' + "{:.3f}".format(self.PERFORMANCES['\_sd'].loc['all', name\_metric])

# print the performances

print('Performances for model ' + self.version + ': ')

print(self.PERFORMANCES['\_str'])

def save\_performances(self):

for mode in self.modes:

path\_save = self.path\_data + 'Performances\_' + self.pred\_type + '\_' + self.version + '\_' + self.fold + \

mode + '.csv'

self.PERFORMANCES[mode].to\_csv(path\_save, index=False)

class PerformancesMerge(Metrics):

"""

Merges the performances of the different models into a unified dataframe.

"""

def \_\_init\_\_(self, target=None, fold=None, pred\_type=None, ensemble\_models=None):

# Parameters

Metrics.\_\_init\_\_(self)

self.target = target

self.fold = fold

self.pred\_type = pred\_type

self.ensemble\_models = self.convert\_string\_to\_boolean(ensemble\_models)

self.names\_metrics = self.dict\_metrics\_names[self.dict\_prediction\_types[target]]

self.main\_metric\_name = self.dict\_main\_metrics\_names[target]

# list the models that need to be merged

self.list\_models = glob.glob(self.path\_data + 'Performances\_' + pred\_type + '\_' + target + '\_\*\_' + fold +

'\_str.csv')

# get rid of ensemble models

if self.ensemble\_models:

self.list\_models = [model for model in self.list\_models if '\*' in model]

else:

self.list\_models = [model for model in self.list\_models if '\*' not in model]

self.Performances = None

self.Performances\_alphabetical = None

self.Performances\_ranked = None

def \_initiate\_empty\_performances\_summary\_df(self):

# Define the columns of the Performances dataframe

# columns for sample sizes

names\_sample\_sizes = ['N']

if self.target in self.targets\_binary:

names\_sample\_sizes.extend(['N\_0', 'N\_1'])

# columns for metrics

names\_metrics = self.dict\_metrics\_names[self.dict\_prediction\_types[self.target]]

# for normal folds, keep track of metric and bootstrapped metric's sd

names\_metrics\_with\_sd = []

for name\_metric in names\_metrics:

names\_metrics\_with\_sd.extend([name\_metric, name\_metric + '\_sd', name\_metric + '\_str'])

# for the 'all' fold, also keep track of the 'folds\_sd' (metric's sd calculated using the folds' results)

names\_metrics\_with\_folds\_sd\_and\_sd = []

for name\_metric in names\_metrics:

names\_metrics\_with\_folds\_sd\_and\_sd.extend(

[name\_metric, name\_metric + '\_folds\_sd', name\_metric + '\_sd', name\_metric + '\_str'])

# merge all the columns together. First description of the model, then sample sizes and metrics for each fold

names\_col\_Performances = ['version'] + self.names\_model\_parameters

# special outer fold 'all'

names\_col\_Performances.extend(

['\_'.join([name, 'all']) for name in names\_sample\_sizes + names\_metrics\_with\_folds\_sd\_and\_sd])

# other outer\_folds

for outer\_fold in self.outer\_folds:

names\_col\_Performances.extend(

['\_'.join([name, outer\_fold]) for name in names\_sample\_sizes + names\_metrics\_with\_sd])

# Generate the empty Performance table from the rows and columns.

Performances = np.empty((len(self.list\_models), len(names\_col\_Performances),))

Performances.fill(np.nan)

Performances = pd.DataFrame(Performances)

Performances.columns = names\_col\_Performances

# Format the types of the columns

for colname in Performances.columns.values:

if (colname in self.names\_model\_parameters) | ('\_str' in colname):

col\_type = str

else:

col\_type = float

Performances[colname] = Performances[colname].astype(col\_type)

self.Performances = Performances

def merge\_performances(self):

# define parameters

names\_metrics = self.dict\_metrics\_names[self.dict\_prediction\_types[self.target]]

# initiate dataframe

self.\_initiate\_empty\_performances\_summary\_df()

# Fill the Performance table row by row

for i, model in enumerate(self.list\_models):

# load the performances subdataframe

PERFORMANCES = {}

for mode in self.modes:

PERFORMANCES[mode] = pd.read\_csv(model.replace('\_str', mode))

PERFORMANCES[mode].set\_index('outer\_fold', drop=False, inplace=True)

# Fill the columns corresponding to the model's parameters

version = '\_'.join(model.split('\_')[2:-2])

parameters = self.\_version\_to\_parameters(version)

# fill the columns for model parameters

self.Performances['version'][i] = version

for parameter\_name in self.names\_model\_parameters:

self.Performances[parameter\_name][i] = parameters[parameter\_name]

# Fill the columns for this model, outer\_fold by outer\_fold

for outer\_fold in ['all'] + self.outer\_folds:

# Generate a subdataframe from the predictions table for each outerfold

# Fill sample size columns

self.Performances['N\_' + outer\_fold][i] = PERFORMANCES[''].loc[outer\_fold, 'N']

# For binary classification, calculate sample sizes for each class and generate class prediction

if self.target in self.targets\_binary:

self.Performances['N\_0\_' + outer\_fold][i] = PERFORMANCES[''].loc[outer\_fold, 'N\_0']

self.Performances['N\_1\_' + outer\_fold][i] = PERFORMANCES[''].loc[outer\_fold, 'N\_1']

# Fill the Performances dataframe metric by metric

for name\_metric in names\_metrics:

for mode in self.modes:

self.Performances[name\_metric + mode + '\_' + outer\_fold][i] = PERFORMANCES[mode].loc[

outer\_fold, name\_metric]

# calculate the fold sd (variance between the metrics values obtained on the different folds)

folds\_sd = PERFORMANCES[''].iloc[1:, :].std(axis=0)

for name\_metric in names\_metrics:

self.Performances[name\_metric + '\_folds\_sd\_all'] = folds\_sd[name\_metric]

# Convert float to int for sample sizes and some metrics.

for name\_col in self.Performances.columns.values:

cond1 = name\_col.startswith('N\_')

cond2 = any(metric in name\_col for metric in self.metrics\_displayed\_in\_int)

cond3 = '\_sd' not in name\_col

cond4 = '\_str' not in name\_col

if cond1 | cond2 & cond3 & cond4:

self.Performances[name\_col] = self.Performances[name\_col].astype('Int64')

# need recent version of pandas to use this type. Otherwise nan cannot be int

# For ensemble models, merge the new performances with the previously computed performances

if self.ensemble\_models:

Performances\_withoutEnsembles = pd.read\_csv(self.path\_data + 'PERFORMANCES\_tuned\_alphabetical\_' +

self.pred\_type + '\_' + self.target + '\_' + self.fold + '.csv')

self.Performances = Performances\_withoutEnsembles.append(self.Performances)

# reorder the columns (weird: automatic alphabetical re-ordering happened when append was called for 'val')

self.Performances = self.Performances[Performances\_withoutEnsembles.columns]

# Ranking, printing and saving

self.Performances\_alphabetical = self.Performances.sort\_values(by='version')

cols\_to\_print = ['version', self.main\_metric\_name + '\_str\_all']

print('Performances of the models ranked by models\'names:')

print(self.Performances\_alphabetical[cols\_to\_print])

sort\_by = self.dict\_main\_metrics\_names[self.target] + '\_all'

sort\_ascending = self.main\_metrics\_modes[self.dict\_main\_metrics\_names[self.target]] == 'min'

self.Performances\_ranked = self.Performances.sort\_values(by=sort\_by, ascending=sort\_ascending)

print('Performances of the models ranked by the performance on the main metric on all the samples:')

print(self.Performances\_ranked[cols\_to\_print])

def save\_performances(self):

name\_extension = 'withEnsembles' if self.ensemble\_models else 'withoutEnsembles'

path = self.path\_data + 'PERFORMANCES\_' + name\_extension + '\_alphabetical\_' + self.pred\_type + '\_' + \

self.target + '\_' + self.fold + '.csv'

self.Performances\_alphabetical.to\_csv(path, index=False)

self.Performances\_ranked.to\_csv(path.replace('\_alphabetical\_', '\_ranked\_'), index=False)

class PerformancesTuning(Metrics):

"""

For each model, selects the best hyperparameter combination.

"""

def \_\_init\_\_(self, target=None, pred\_type=None):

Metrics.\_\_init\_\_(self)

self.target = target

self.pred\_type = pred\_type

self.PERFORMANCES = {}

self.PREDICTIONS = {}

self.Performances = None

self.models = None

self.folds = ['val', 'test']

def load\_data(self):

for fold in self.folds:

path = self.path\_data + 'PERFORMANCES\_withoutEnsembles\_ranked\_' + self.pred\_type + '\_' + self.target + \

'\_' + fold + '.csv'

self.PERFORMANCES[fold] = pd.read\_csv(path).set\_index('version', drop=False)

self.PERFORMANCES[fold]['organ'] = self.PERFORMANCES[fold]['organ'].astype(str)

self.PERFORMANCES[fold].index.name = 'columns\_names'

self.PREDICTIONS[fold] = pd.read\_csv(path.replace('PERFORMANCES', 'PREDICTIONS').replace('\_ranked', ''))

def preprocess\_data(self):

# Get list of distinct models without taking into account hyperparameters tuning

self.Performances = self.PERFORMANCES['val']

self.Performances['model'] = self.Performances['organ'] + '\_' + self.Performances['view'] + '\_' + \

self.Performances['transformation'] + '\_' + self.Performances['architecture']

self.models = self.Performances['model'].unique()

def select\_models(self):

main\_metric\_name = self.dict\_main\_metrics\_names[self.target]

main\_metric\_mode = self.main\_metrics\_modes[main\_metric\_name]

Perf\_col\_name = main\_metric\_name + '\_all'

for model in self.models:

Performances\_model = self.Performances[self.Performances['model'] == model]

Performances\_model.sort\_values([Perf\_col\_name, 'n\_fc\_layers', 'n\_fc\_nodes', 'learning\_rate', 'dropout\_rate',

'weight\_decay', 'data\_augmentation\_factor'],

ascending=[main\_metric\_mode == 'min', True, True, False, False, False,

False], inplace=True)

best\_version = Performances\_model['version'][

Performances\_model[Perf\_col\_name] == Performances\_model[Perf\_col\_name].max()].values[0]

versions\_to\_drop = [version for version in Performances\_model['version'].values if

not version == best\_version]

# define columns from predictions to drop

cols\_to\_drop = ['pred\_' + version for version in versions\_to\_drop] + ['outer\_fold\_' + version for version in

versions\_to\_drop]

for fold in self.folds:

self.PERFORMANCES[fold].drop(versions\_to\_drop, inplace=True)

self.PREDICTIONS[fold].drop(cols\_to\_drop, axis=1, inplace=True)

# drop 'model' column

self.Performances.drop(['model'], axis=1, inplace=True)

# Display results

for fold in self.folds:

print('The tuned ' + fold + ' performances are:')

print(self.PERFORMANCES[fold])

def save\_data(self):

# Save the files

for fold in self.folds:

path\_pred = self.path\_data + 'PREDICTIONS\_tuned\_' + self.pred\_type + '\_' + self.target + '\_' + fold + \

'.csv'

path\_perf = self.path\_data + 'PERFORMANCES\_tuned\_ranked\_' + self.pred\_type + '\_' + self.target + '\_' + \

fold + '.csv'

self.PREDICTIONS[fold].to\_csv(path\_pred, index=False)

self.PERFORMANCES[fold].to\_csv(path\_perf, index=False)

Performances\_alphabetical = self.PERFORMANCES[fold].sort\_values(by='version')

Performances\_alphabetical.to\_csv(path\_perf.replace('ranked', 'alphabetical'), index=False)

# This class was coded by Samuel Diai.

class InnerCV:

"""

Helper class to perform an inner cross validation to tune the hyperparameters of models trained on scalar predictors

"""

def \_\_init\_\_(self, models, inner\_splits, n\_iter):

self.inner\_splits = inner\_splits

self.n\_iter = n\_iter

if isinstance(models, str):

models = [models]

self.models = models

@staticmethod

def get\_model(model\_name, params):

if model\_name == 'ElasticNet':

return ElasticNet(max\_iter=2000, \*\*params)

elif model\_name == 'RandomForest':

return RandomForestRegressor(\*\*params)

elif model\_name == 'GradientBoosting':

return GradientBoostingRegressor(\*\*params)

elif model\_name == 'Xgboost':

return XGBRegressor(\*\*params)

elif model\_name == 'LightGbm':

return LGBMRegressor(\*\*params)

elif model\_name == 'NeuralNetwork':

return MLPRegressor(solver='adam',

activation='relu',

hidden\_layer\_sizes=(128, 64, 32),

batch\_size=1000,

early\_stopping=True, \*\*params)

@staticmethod

def get\_hyper\_distribution(model\_name):

if model\_name == 'ElasticNet':

return {

'alpha': hp.loguniform('alpha', low=np.log(0.01), high=np.log(10)),

'l1\_ratio': hp.uniform('l1\_ratio', low=0.01, high=0.99)

}

elif model\_name == 'RandomForest':

return {

'n\_estimators': hp.randint('n\_estimators', upper=300) + 150,

'max\_features': hp.choice('max\_features', ['auto', 0.9, 0.8, 0.7, 0.6, 0.5, 0.4]),

'max\_depth': hp.choice('max\_depth', [None, 10, 8, 6])

}

elif model\_name == 'GradientBoosting':

return {

'n\_estimators': hp.randint('n\_estimators', upper=300) + 150,

'max\_features': hp.choice('max\_features', ['auto', 0.9, 0.8, 0.7, 0.6, 0.5, 0.4, 0.3]),

'learning\_rate': hp.uniform('learning\_rate', low=0.01, high=0.3),

'max\_depth': hp.randint('max\_depth', 10) + 5

}

elif model\_name == 'Xgboost':

return {

'colsample\_bytree': hp.uniform('colsample\_bytree', low=0.2, high=0.7),

'gamma': hp.uniform('gamma', low=0.1, high=0.5),

'learning\_rate': hp.uniform('learning\_rate', low=0.02, high=0.2),

'max\_depth': hp.randint('max\_depth', 10) + 5,

'n\_estimators': hp.randint('n\_estimators', 300) + 150,

'subsample': hp.uniform('subsample', 0.2, 0.8)

}

elif model\_name == 'LightGbm':

return {

'num\_leaves': hp.randint('num\_leaves', 40) + 5,

'min\_child\_samples': hp.randint('min\_child\_samples', 400) + 100,

'min\_child\_weight': hp.choice('min\_child\_weight', [1e-5, 1e-3, 1e-2, 1e-1, 1, 1e1, 1e2, 1e3, 1e4]),

'subsample': hp.uniform('subsample', low=0.2, high=0.8),

'colsample\_bytree': hp.uniform('colsample\_bytree', low=0.4, high=0.6),

'reg\_alpha': hp.choice('reg\_alpha', [0, 1e-1, 1, 2, 5, 7, 10, 50, 100]),

'reg\_lambda': hp.choice('reg\_lambda', [0, 1e-1, 1, 5, 10, 20, 50, 100]),

'n\_estimators': hp.randint('n\_estimators', 300) + 150

}

elif model\_name == 'NeuralNetwork':

return {

'learning\_rate\_init': hp.loguniform('learning\_rate\_init', low=np.log(5e-5), high=np.log(2e-2)),

'alpha': hp.uniform('alpha', low=1e-6, high=1e3)

}

def create\_folds(self, X, y):

"""

X columns : eid + features except target

y columns : eid + target

"""

X\_eid = X.drop\_duplicates('eid')

y\_eid = y.drop\_duplicates('eid')

eids = X\_eid.eid

# Kfold on the eid, then regroup all ids

inner\_cv = KFold(n\_splits=self.inner\_splits, shuffle=False, random\_state=0)

list\_test\_folds = [elem[1] for elem in inner\_cv.split(X\_eid, y\_eid)]

list\_test\_folds\_eid = [eids[elem].values for elem in list\_test\_folds]

list\_test\_folds\_id = [X.index[X.eid.isin(list\_test\_folds\_eid[elem])].values for elem in

range(len(list\_test\_folds\_eid))]

return list\_test\_folds\_id

def optimize\_hyperparameters(self, X, y, scoring):

"""

input X : dataframe with features + eid

input y : dataframe with target + eid

"""

if 'instance' in X.columns:

X = X.drop(columns=['instance'])

if 'instance' in y.columns:

y = y.drop(columns=['instance'])

list\_test\_folds\_id = self.create\_folds(X, y)

X = X.drop(columns=['eid'])

y = y.drop(columns=['eid'])

# Create custom Splits

list\_test\_folds\_id\_index = [np.array([X.index.get\_loc(elem) for elem in list\_test\_folds\_id[fold\_num]])

for fold\_num in range(len(list\_test\_folds\_id))]

test\_folds = np.zeros(len(X), dtype='int')

for fold\_count in range(len(list\_test\_folds\_id)):

test\_folds[list\_test\_folds\_id\_index[fold\_count]] = fold\_count

inner\_cv = PredefinedSplit(test\_fold=test\_folds)

list\_best\_params = {}

list\_best\_score = {}

objective, model\_name = None, None

for model\_name in self.models:

def objective(hyperparameters):

estimator\_ = self.get\_model(model\_name, hyperparameters)

pipeline = Pipeline([('scaler', StandardScaler()), ('estimator', estimator\_)])

scores = cross\_validate(pipeline, X.values, y, scoring=scoring, cv=inner\_cv, n\_jobs=self.inner\_splits)

return {'status': STATUS\_OK, 'loss': -scores['test\_score'].mean(),

'attachments': {'split\_test\_scores\_and\_params': (scores['test\_score'], hyperparameters)}}

space = self.get\_hyper\_distribution(model\_name)

trials = Trials()

best = fmin(objective, space, algo=tpe.suggest, max\_evals=self.n\_iter, trials=trials)

best\_params = space\_eval(space, best)

list\_best\_params[model\_name] = best\_params

list\_best\_score[model\_name] = - min(trials.losses())

# Recover best between all models :

best\_model = max(list\_best\_score.keys(), key=(lambda k: list\_best\_score[k]))

best\_model\_hyp = list\_best\_params[best\_model]

# Recreate best estim :

estim = self.get\_model(best\_model, best\_model\_hyp)

pipeline\_best = Pipeline([('scaler', StandardScaler()), ('estimator', estim)])

pipeline\_best.fit(X.values, y)

return pipeline\_best

"""

Useful for EnsemblesPredictions. This function needs to be global to allow pool to pickle it.

"""

def compute\_ensemble\_folds(ensemble\_inputs):

if len(ensemble\_inputs[1]) < 100:

print('Small sample size:' + str(len(ensemble\_inputs[1])))

n\_inner\_splits = 5

else:

n\_inner\_splits = 10

# Can use different models: models=['ElasticNet', 'LightGBM', 'NeuralNetwork']

cv = InnerCV(models=['ElasticNet'], inner\_splits=n\_inner\_splits, n\_iter=30)

model = cv.optimize\_hyperparameters(ensemble\_inputs[0], ensemble\_inputs[1], scoring='r2')

return model

class EnsemblesPredictions(Metrics):

"""

Hierarchically builds ensemble models from the already existing predictions.

"""

def \_\_init\_\_(self, target=None, pred\_type=None, regenerate\_models=False):

# Parameters

Metrics.\_\_init\_\_(self)

self.target = target

self.pred\_type = pred\_type

self.folds = ['val', 'test']

self.regenerate\_models = regenerate\_models

self.ensembles\_performance\_cutoff\_percent = 0.5

self.parameters = {'target': target, 'organ': '\*', 'view': '\*', 'transformation': '\*', 'architecture': '\*',

'n\_fc\_layers': '\*', 'n\_fc\_nodes': '\*', 'optimizer': '\*', 'learning\_rate': '\*',

'weight\_decay': '\*', 'dropout\_rate': '\*', 'data\_augmentation\_factor': '\*'}

self.version = self.\_parameters\_to\_version(self.parameters)

self.main\_metric\_name = self.dict\_main\_metrics\_names[target]

self.init\_perf = -np.Inf if self.main\_metrics\_modes[self.main\_metric\_name] == 'max' else np.Inf

path\_perf = self.path\_data + 'PERFORMANCES\_tuned\_ranked\_' + pred\_type + '\_' + target + '\_val.csv'

self.Performances = pd.read\_csv(path\_perf).set\_index('version', drop=False)

self.Performances['organ'] = self.Performances['organ'].astype(str)

self.list\_ensemble\_levels = ['transformation', 'view', 'organ']

self.PREDICTIONS = {}

self.weights\_by\_category = None

self.weights\_by\_ensembles = None

self.N\_ensemble\_CV\_split = 10

self.instancesS = {'instances': ['01', '1.5x', '23'], 'eids': ['\*']}

self.instances\_names\_to\_numbers = {'01': ['0', '1'], '1.5x': ['1.5', '1.51', '1.52', '1.53', '1.54'],

'23': ['2', '3'], '\*': ['\*']}

self.INSTANCES\_DATASETS = {

'01': ['Eyes', 'Hearing', 'Lungs', 'Arterial', 'Musculoskeletal', 'Biochemistry', 'ImmuneSystem'],

'1.5x': ['PhysicalActivity'],

'23': ['Brain', 'Arterial', 'Heart', 'Abdomen', 'Musculoskeletal'],

'\*': ['Brain', 'Eyes', 'Hearing', 'Lungs', 'Arterial', 'Heart', 'Abdomen', 'Musculoskeletal',

'PhysicalActivity', 'ImmuneSystem']

}

# Get rid of columns and rows for the versions for which all samples as NANs

@staticmethod

def \_drop\_na\_pred\_versions(PREDS, Performances):

# Select the versions for which only NAs are available

pred\_versions = [col for col in PREDS['val'].columns.values if 'pred\_' in col]

to\_drop = []

for pv in pred\_versions:

for fold in PREDS.keys():

if PREDS[fold][pv].notna().sum() == 0:

to\_drop.append(pv)

break

# Drop the corresponding columns from preds, and rows from performances

index\_to\_drop = [p.replace('pred\_', '') for p in to\_drop if '\*' not in p]

for fold in PREDS.keys():

PREDS[fold].drop(to\_drop, axis=1, inplace=True)

return Performances.drop(index\_to\_drop)

def load\_data(self):

for fold in self.folds:

self.PREDICTIONS[fold] = pd.read\_csv(

self.path\_data + 'PREDICTIONS\_tuned\_' + self.pred\_type + '\_' + self.target + '\_' + fold + '.csv')

def \_build\_single\_ensemble(self, PREDICTIONS, version):

# Drop columns that are exclusively NaNs

all\_nan = PREDICTIONS['val'].isna().all() | PREDICTIONS['test'].isna().all()

non\_nan\_cols = all\_nan[~all\_nan.values].index

for fold in self.folds:

PREDICTIONS[fold] = PREDICTIONS[fold][non\_nan\_cols]

Predictions = PREDICTIONS['val']

# Select the columns for the model

ensemble\_preds\_cols = [col for col in Predictions.columns.values if

bool(re.compile('pred\_' + version).match(col))]

# If only one model in the ensemble, just copy the column. Otherwise build the ensemble model

if len(ensemble\_preds\_cols) == 1:

for fold in self.folds:

PREDICTIONS[fold]['pred\_' + version] = PREDICTIONS[fold][ensemble\_preds\_cols[0]]

else:

# Initiate the dictionaries

PREDICTIONS\_OUTERFOLDS = {}

ENSEMBLE\_INPUTS = {}

for outer\_fold in self.outer\_folds:

# take the subset of the rows that correspond to the outer\_fold

PREDICTIONS\_OUTERFOLDS[outer\_fold] = {}

XS\_outer\_fold = {}

YS\_outer\_fold = {}

dict\_fold\_to\_outer\_folds = {

'val': [float(outer\_fold)],

'test': [(float(outer\_fold) + 1) % self.n\_CV\_outer\_folds],

'train': [float(of) for of in self.outer\_folds

if float(of) not in [float(outer\_fold), (float(outer\_fold) + 1) % self.n\_CV\_outer\_folds]]

}

for fold in self.folds:

PREDICTIONS\_OUTERFOLDS[outer\_fold][fold] = \

PREDICTIONS[fold][PREDICTIONS[fold]['outer\_fold'].isin(dict\_fold\_to\_outer\_folds[fold])]

PREDICTIONS\_OUTERFOLDS[outer\_fold][fold] = PREDICTIONS\_OUTERFOLDS[outer\_fold][fold][

['id', 'eid', 'instance', self.target] + ensemble\_preds\_cols].dropna()

X = PREDICTIONS\_OUTERFOLDS[outer\_fold][fold][['id', 'eid', 'instance'] + ensemble\_preds\_cols]

X.set\_index('id', inplace=True)

XS\_outer\_fold[fold] = X

y = PREDICTIONS\_OUTERFOLDS[outer\_fold][fold][['id', 'eid', self.target]]

y.set\_index('id', inplace=True)

YS\_outer\_fold[fold] = y

ENSEMBLE\_INPUTS[outer\_fold] = [XS\_outer\_fold['val'], YS\_outer\_fold['val']]

# Build ensemble model using ElasticNet and/or LightGBM, Neural Network.

PREDICTIONS\_ENSEMBLE = {}

pool = Pool(self.N\_ensemble\_CV\_split)

MODELS = pool.map(compute\_ensemble\_folds, list(ENSEMBLE\_INPUTS.values()))

pool.close()

pool.join()

# Concatenate all outer folds

for outer\_fold in self.outer\_folds:

for fold in self.folds:

X = PREDICTIONS\_OUTERFOLDS[outer\_fold][fold][ensemble\_preds\_cols]

PREDICTIONS\_OUTERFOLDS[outer\_fold][fold]['pred\_' + version] = MODELS[int(outer\_fold)].predict(X)

PREDICTIONS\_OUTERFOLDS[outer\_fold][fold]['outer\_fold'] = float(outer\_fold)

df\_outer\_fold = PREDICTIONS\_OUTERFOLDS[outer\_fold][fold][['id', 'outer\_fold',

'pred\_' + version]]

# Initiate, or append if some previous outerfolds have already been concatenated

if fold not in PREDICTIONS\_ENSEMBLE.keys():

PREDICTIONS\_ENSEMBLE[fold] = df\_outer\_fold

else:

PREDICTIONS\_ENSEMBLE[fold] = PREDICTIONS\_ENSEMBLE[fold].append(df\_outer\_fold)

# Add the ensemble predictions to the dataframe

for fold in self.folds:

if fold == 'train':

PREDICTIONS[fold] = PREDICTIONS[fold].merge(PREDICTIONS\_ENSEMBLE[fold], how='outer',

on=['id', 'outer\_fold'])

else:

PREDICTIONS\_ENSEMBLE[fold].drop('outer\_fold', axis=1, inplace=True)

PREDICTIONS[fold] = PREDICTIONS[fold].merge(PREDICTIONS\_ENSEMBLE[fold], how='outer', on=['id'])

def \_build\_single\_ensemble\_wrapper(self, version, ensemble\_level):

print('Building the ensemble model ' + version)

pred\_version = 'pred\_' + version

# Evaluate if the ensemble model should be built

# 1 - separately on instance 0-1, 1.5 and 2-3 (for ensemble at the top level, since overlap between models is 0)

# 2 - piece by piece on each outer\_fold

# 1-Compute instances 0-1, 1.5 and 2-3 separately

if ensemble\_level == 'organ':

for fold in self.folds:

self.PREDICTIONS[fold][pred\_version] = np.nan

# Add an ensemble for each instances (01, 1.5x, and 23)

if self.pred\_type == 'instances':

for instances\_names in self.instancesS[self.pred\_type]:

pv = 'pred\_' + version.split('\_')[0] + '\_\*instances' + instances\_names + '\_' + \

'\_'.join(version.split('\_')[2:])

self.PREDICTIONS[fold][pv] = np.nan

for instances\_names in self.instancesS[self.pred\_type]:

print('Building final ensemble model for samples in the instances: ' + instances\_names)

# Take subset of rows and columns

instances = self.instances\_names\_to\_numbers[instances\_names]

instances\_datasets = self.INSTANCES\_DATASETS[instances\_names]

versions = \

[col.replace('pred\_', '') for col in self.PREDICTIONS['val'].columns if 'pred\_' in col]

instances\_versions = [version for version in versions

if any(dataset in version for dataset in instances\_datasets)]

cols\_to\_keep = self.id\_vars + self.demographic\_vars + \

['pred\_' + version for version in instances\_versions]

PREDICTIONS = {}

for fold in self.folds:

PREDICTIONS[fold] = self.PREDICTIONS[fold][self.PREDICTIONS[fold].instance.isin(instances)]

PREDICTIONS[fold] = PREDICTIONS[fold][cols\_to\_keep]

self.\_build\_single\_ensemble(PREDICTIONS, version)

# Print a quick performance estimation for each instance(s)

df\_model = PREDICTIONS['test'][[self.target, 'pred\_' + version]].dropna()

print(instances\_names)

print(self.main\_metric\_name + ' for instance(s) ' + instances\_names + ': ' +

str(r2\_score(df\_model[self.target], df\_model['pred\_' + version])))

print('The sample size is ' + str(len(df\_model.index)) + '.')

# Add the predictions to the dataframe, chunck by chunk, instances by instances

for fold in self.folds:

self.PREDICTIONS[fold][pred\_version][self.PREDICTIONS[fold].instance.isin(instances)] = \

PREDICTIONS[fold][pred\_version].values

# Add an ensemble for the instance(s) only

if self.pred\_type == 'instances':

pv = 'pred\_' + version.split('\_')[0] + '\_\*instances' + instances\_names + '\_' + \

'\_'.join(version.split('\_')[2:])

self.PREDICTIONS[fold][pv][self.PREDICTIONS[fold].instance.isin(instances)] = \

PREDICTIONS[fold][pred\_version].values

# Add three extra ensemble models for eids, to allow larger sample sizes for GWAS purposes

if self.pred\_type == 'eids':

for instances\_names in ['01', '1.5x', '23']:

print('Building final sub-ensemble model for samples in the instances: ' + instances\_names)

# Keep only relevant columns

instances\_datasets = self.INSTANCES\_DATASETS[instances\_names]

versions = \

[col.replace('pred\_', '') for col in self.PREDICTIONS['val'].columns if 'pred\_' in col]

instances\_versions = [version for version in versions

if any(dataset in version for dataset in instances\_datasets)]

cols\_to\_keep = self.id\_vars + self.demographic\_vars + \

['pred\_' + version for version in instances\_versions]

PREDICTIONS = {}

for fold in self.folds:

PREDICTIONS[fold] = self.PREDICTIONS[fold].copy()

PREDICTIONS[fold] = PREDICTIONS[fold][cols\_to\_keep]

self.\_build\_single\_ensemble(PREDICTIONS, version)

# Print a quick performance estimation for each instance(s)

df\_model = PREDICTIONS['test'][[self.target, 'pred\_' + version]].dropna()

print(instances\_names)

print(self.main\_metric\_name + ' for instance(s) ' + instances\_names + ': ' +

str(r2\_score(df\_model[self.target], df\_model['pred\_' + version])))

print('The sample size is ' + str(len(df\_model.index)) + '.')

# Add the predictions to the dataframe

pv = 'pred\_' + version.split('\_')[0] + '\_\*instances' + instances\_names + '\_' + \

'\_'.join(version.split('\_')[2:])

for fold in self.folds:

self.PREDICTIONS[fold][pv] = PREDICTIONS[fold][pred\_version].values

# 2-Compute fold by fold

else:

self.\_build\_single\_ensemble(self.PREDICTIONS, version)

# build and save a dataset for this specific ensemble model

for fold in self.folds:

df\_single\_ensemble = self.PREDICTIONS[fold][['id', 'outer\_fold', pred\_version]]

df\_single\_ensemble.rename(columns={pred\_version: 'pred'}, inplace=True)

df\_single\_ensemble.dropna(inplace=True, subset=['pred'])

df\_single\_ensemble.to\_csv(self.path\_data + 'Predictions\_' + self.pred\_type + '\_' + version + '\_' + fold +

'.csv', index=False)

# Add extra ensembles at organ level

if ensemble\_level == 'organ':

for instances\_names in ['01', '1.5x', '23']:

pv = 'pred\_' + version.split('\_')[0] + '\_\*instances' + instances\_names + '\_' + \

'\_'.join(version.split('\_')[2:])

version\_instances = version.split('\_')[0] + '\_\*instances' + instances\_names + '\_' + \

'\_'.join(version.split('\_')[2:])

df\_single\_ensemble = self.PREDICTIONS[fold][['id', 'outer\_fold', pv]]

df\_single\_ensemble.rename(columns={pv: 'pred'}, inplace=True)

df\_single\_ensemble.dropna(inplace=True, subset=['pred'])

df\_single\_ensemble.to\_csv(self.path\_data + 'Predictions\_' + self.pred\_type + '\_' +

version\_instances + '\_' + fold + '.csv', index=False)

def \_recursive\_ensemble\_builder(self, Performances\_grandparent, parameters\_parent, version\_parent,

list\_ensemble\_levels\_parent):

# Compute the ensemble models for the children first, so that they can be used for the parent model

Performances\_parent = Performances\_grandparent[

Performances\_grandparent['version'].isin(

fnmatch.filter(Performances\_grandparent['version'], version\_parent))]

# if the last ensemble level has not been reached, go down one level and create a branch for each child.

# Otherwise the leaf has been reached

if len(list\_ensemble\_levels\_parent) > 0:

list\_ensemble\_levels\_child = list\_ensemble\_levels\_parent.copy()

ensemble\_level = list\_ensemble\_levels\_child.pop()

list\_children = Performances\_parent[ensemble\_level].unique()

for child in list\_children:

parameters\_child = parameters\_parent.copy()

parameters\_child[ensemble\_level] = child

version\_child = self.\_parameters\_to\_version(parameters\_child)

# recursive call to the function

self.\_recursive\_ensemble\_builder(Performances\_parent, parameters\_child, version\_child,

list\_ensemble\_levels\_child)

else:

ensemble\_level = None

# compute the ensemble model for the parent

# Check if ensemble model has already been computed. If it has, load the predictions. If it has not, compute it.

if not self.regenerate\_models and \

os.path.exists(self.path\_data + 'Predictions\_' + self.pred\_type + '\_' + version\_parent + '\_test.csv'):

print('The model ' + version\_parent + ' has already been computed. Loading it...')

for fold in self.folds:

df\_single\_ensemble = pd.read\_csv(self.path\_data + 'Predictions\_' + self.pred\_type + '\_' +

version\_parent + '\_' + fold + '.csv')

df\_single\_ensemble.rename(columns={'pred': 'pred\_' + version\_parent}, inplace=True)

# Add the ensemble predictions to the dataframe

if fold == 'train':

self.PREDICTIONS[fold] = self.PREDICTIONS[fold].merge(df\_single\_ensemble, how='outer',

on=['id', 'outer\_fold'])

else:

df\_single\_ensemble.drop(columns=['outer\_fold'], inplace=True)

self.PREDICTIONS[fold] = self.PREDICTIONS[fold].merge(df\_single\_ensemble, how='outer', on=['id'])

# Add the extra ensemble models at the 'organ' level

if ensemble\_level == 'organ':

if self.pred\_type == 'instances':

instances = self.instancesS[self.pred\_type]

else:

instances = ['01', '23']

for instances\_names in instances:

pv = 'pred\_' + version\_parent.split('\_')[0] + '\_\*instances' + instances\_names + '\_' + \

'\_'.join(version\_parent.split('\_')[2:])

version\_instances = version\_parent.split('\_')[0] + '\_\*instances' + instances\_names + '\_' + \

'\_'.join(version\_parent.split('\_')[2:])

df\_single\_ensemble = pd.read\_csv(self.path\_data + 'Predictions\_' + self.pred\_type + '\_' +

version\_instances + '\_' + fold + '.csv')

df\_single\_ensemble.rename(columns={'pred': pv}, inplace=True)

if fold == 'train':

self.PREDICTIONS[fold] = self.PREDICTIONS[fold].merge(df\_single\_ensemble, how='outer',

on=['id', 'outer\_fold'])

else:

df\_single\_ensemble.drop(columns=['outer\_fold'], inplace=True)

self.PREDICTIONS[fold] = self.PREDICTIONS[fold].merge(df\_single\_ensemble, how='outer',

on=['id'])

else:

self.\_build\_single\_ensemble\_wrapper(version\_parent, ensemble\_level)

# Print a quick performance estimation

df\_model = self.PREDICTIONS['test'][[self.target, 'pred\_' + version\_parent]].dropna()

print(self.main\_metric\_name + ': ' + str(r2\_score(df\_model[self.target], df\_model['pred\_' + version\_parent])))

print('The sample size is ' + str(len(df\_model.index)) + '.')

def generate\_ensemble\_predictions(self):

self.\_recursive\_ensemble\_builder(self.Performances, self.parameters, self.version, self.list\_ensemble\_levels)

# Reorder the columns alphabetically

for fold in self.folds:

pred\_versions = [col for col in self.PREDICTIONS[fold].columns if 'pred\_' in col]

pred\_versions.sort()

self.PREDICTIONS[fold] = self.PREDICTIONS[fold][self.id\_vars + self.demographic\_vars + pred\_versions]

# Displaying the R2s

for fold in self.folds:

versions = [col.replace('pred\_', '') for col in self.PREDICTIONS[fold].columns if 'pred\_' in col]

r2s = []

for version in versions:

df = self.PREDICTIONS[fold][[self.target, 'pred\_' + version]].dropna()

r2s.append(r2\_score(df[self.target], df['pred\_' + version]))

R2S = pd.DataFrame({'version': versions, 'R2': r2s})

R2S.sort\_values(by='R2', ascending=False, inplace=True)

print(fold + ' R2s for each model: ')

print(R2S)

def save\_predictions(self):

for fold in self.folds:

self.PREDICTIONS[fold].to\_csv(self.path\_data + 'PREDICTIONS\_withEnsembles\_' + self.pred\_type + '\_' +

self.target + '\_' + fold + '.csv', index=False)

class ResidualsGenerate(Basics):

"""

Computes accelerated aging phenotypes (Residuals, corrected for residuals bias with respect to age)

"""

def \_\_init\_\_(self, target=None, fold=None, pred\_type=None, debug\_mode=False):

# Parameters

Basics.\_\_init\_\_(self)

self.target = target

self.fold = fold

self.pred\_type = pred\_type

self.debug\_mode = debug\_mode

self.Residuals = pd.read\_csv(self.path\_data + 'PREDICTIONS\_withEnsembles\_' + pred\_type + '\_' + target + '\_' +

fold + '.csv')

self.list\_models = [col\_name.replace('pred\_', '') for col\_name in self.Residuals.columns.values

if 'pred\_' in col\_name]

def generate\_residuals(self):

list\_models = [col\_name.replace('pred\_', '') for col\_name in self.Residuals.columns.values

if 'pred\_' in col\_name]

for model in list\_models:

print('Generating residuals for model ' + model)

df\_model = self.Residuals[['Age', 'pred\_' + model]]

no\_na\_indices = [not b for b in df\_model['pred\_' + model].isna()]

df\_model.dropna(inplace=True)

if (len(df\_model.index)) > 0:

age = df\_model.loc[:, ['Age']]

res = df\_model['Age'] - df\_model['pred\_' + model]

regr = LinearRegression()

regr.fit(age, res)

res\_correction = regr.predict(age)

res\_corrected = res - res\_correction

self.Residuals.loc[no\_na\_indices, 'pred\_' + model] = res\_corrected

# debug plot

if self.debug\_mode:

print('Bias for the residuals ' + model, regr.coef\_)

plt.scatter(age, res)

plt.scatter(age, res\_corrected)

regr2 = LinearRegression()

regr2.fit(age, res\_corrected)

print('Coefficients after: \n', regr2.coef\_)

self.Residuals.rename(columns=lambda x: x.replace('pred\_', 'res\_'), inplace=True)

def save\_residuals(self):

self.Residuals.to\_csv(self.path\_data + 'RESIDUALS\_' + self.pred\_type + '\_' + self.target + '\_' + self.fold +

'.csv', index=False)

class ResidualsCorrelations(Basics):

"""

Computes the phenotypic correlation between aging dimensions.

"""

def \_\_init\_\_(self, target=None, fold=None, pred\_type=None, debug\_mode=False):

Basics.\_\_init\_\_(self)

self.target = target

self.fold = fold

self.pred\_type = pred\_type

self.debug\_mode = debug\_mode

if debug\_mode:

self.n\_bootstrap\_iterations\_correlations = 10

else:

self.n\_bootstrap\_iterations\_correlations = 1000

self.Residuals = None

self.CORRELATIONS = {}

self.Correlation\_sample\_sizes = None

def preprocessing(self):

# load data

Residuals = pd.read\_csv(self.path\_data + 'RESIDUALS\_' + self.pred\_type + '\_' + self.target + '\_' + self.fold +

'.csv')

# Format the dataframe

Residuals\_only = Residuals[[col\_name for col\_name in Residuals.columns.values if 'res\_' in col\_name]]

Residuals\_only.rename(columns=lambda x: x.replace('res\_' + self.target + '\_', ''), inplace=True)

# Reorder the columns to make the correlation matrix more readable

# Need to temporarily rename '?' because its ranking differs from the '\*' and ',' characters

Residuals\_only.columns = [col\_name.replace('?', ',placeholder') for col\_name in Residuals\_only.columns.values]

Residuals\_only = Residuals\_only.reindex(sorted(Residuals\_only.columns), axis=1)

Residuals\_only.columns = [col\_name.replace(',placeholder', '?') for col\_name in Residuals\_only.columns.values]

self.Residuals = Residuals\_only

def \_bootstrap\_correlations(self):

names = self.Residuals.columns.values

results = []

for i in range(self.n\_bootstrap\_iterations\_correlations):

if (i + 1) % 100 == 0:

print('Bootstrap iteration ' + str(i + 1) + ' out of ' + str(self.n\_bootstrap\_iterations\_correlations))

data\_i = resample(self.Residuals, replace=True, n\_samples=len(self.Residuals.index))

results.append(np.array(data\_i.corr()))

results = np.array(results)

RESULTS = {}

for op in ['mean', 'std']:

results\_op = pd.DataFrame(getattr(np, op)(results, axis=0))

results\_op.index = names

results\_op.columns = names

RESULTS[op] = results\_op

self.CORRELATIONS['\_sd'] = RESULTS['std']

def generate\_correlations(self):

# Generate the correlation matrix

self.CORRELATIONS[''] = self.Residuals.corr()

# Gerate the std by bootstrapping

self.\_bootstrap\_correlations()

# Merge both as a dataframe of strings

self.CORRELATIONS['\_str'] = self.CORRELATIONS[''].round(3).applymap(str) \

+ '+-' + self.CORRELATIONS['\_sd'].round(3).applymap(str)

# Print correlations

print(self.CORRELATIONS[''])

# Generate correlation sample sizes

self.Residuals[~self.Residuals.isna()] = 1

self.Residuals[self.Residuals.isna()] = 0

self.Correlation\_sample\_sizes = self.Residuals.transpose() @ self.Residuals

def save\_correlations(self):

self.Correlation\_sample\_sizes.to\_csv(self.path\_data + 'ResidualsCorrelations\_samplesizes\_' + self.pred\_type +

'\_' + self.target + '\_' + self.fold + '.csv', index=True)

for mode in self.modes:

self.CORRELATIONS[mode].to\_csv(self.path\_data + 'ResidualsCorrelations' + mode + '\_' + self.pred\_type +

'\_' + self.target + '\_' + self.fold + '.csv', index=True)

class PerformancesSurvival(Metrics):

"""

Computes the performances in terms of survival prediction using biological age phenotypes as survival predictors.

"""

def \_\_init\_\_(self, target=None, fold=None, pred\_type=None, debug\_mode=None):

Metrics.\_\_init\_\_(self)

self.target = target

self.fold = fold

self.pred\_type = pred\_type

if debug\_mode:

self.n\_bootstrap\_iterations = 3

else:

self.n\_bootstrap\_iterations = 1000

self.PERFORMANCES = None

self.Survival = None

self.SURV = None

def \_bootstrap\_c\_index(self, data):

results = []

for i in range(self.n\_bootstrap\_iterations):

data\_i = resample(data, replace=True, n\_samples=len(data.index))

if len(data\_i['Death'].unique()) == 2:

results.append(concordance\_index(data\_i['Age'], -data\_i['pred'], data\_i['Death']))

'''

To debug if this part fails again

try:

results.append(concordance\_index(data\_i['Age'], -data\_i['pred'], data\_i['Death']))

except:

print('WEIRD, should not happen! Printing the df')

print(data\_i)

self.data\_i\_debug = data\_i

break

'''

if len(results) > 0:

results\_mean = np.mean(results)

results\_std = np.std(results)

else:

results\_mean = np.nan

results\_std = np.nan

return results\_mean, results\_std

def load\_data(self):

# Load and preprocess PERFORMANCES

self.PERFORMANCES = pd.read\_csv(self.path\_data + 'PERFORMANCES\_withEnsembles\_alphabetical\_' +

self.pred\_type + '\_' + self.target + '\_' + self.fold + '.csv')

self.PERFORMANCES.set\_index('version', drop=False, inplace=True)

self.PERFORMANCES.index.name = 'index'

for inner\_fold in ['all'] + [str(i) for i in range(10)]:

for metric in ['C-Index', 'C-Index-difference']:

for mode in self.modes:

self.PERFORMANCES[metric + mode + '\_' + inner\_fold] = np.nan

Residuals = pd.read\_csv(

self.path\_data + 'RESIDUALS\_' + self.pred\_type + '\_' + self.target + '\_' + self.fold + '.csv')

Survival = pd.read\_csv(self.path\_data + 'data\_survival.csv')

self.Survival = pd.merge(Survival[['id', 'FollowUpTime', 'Death']], Residuals, on='id')

data\_folds = pd.read\_csv(self.path\_data + 'data-features\_eids.csv', usecols=['eid', 'outer\_fold'])

self.SURV = {}

for i in range(10):

self.SURV[i] = \

self.Survival[self.Survival['eid'].isin(data\_folds['eid'][data\_folds['outer\_fold'] == i].values)]

def compute\_c\_index\_and\_save\_data(self):

models = [col.replace('res\_' + self.target, self.target) for col in self.Survival.columns if 'res\_' in col]

for k, model in enumerate(models):

if k % 30 == 0:

print('Computing CI for the ' + str(k) + 'th model out of ' + str(len(models)) + ' models.')

# Load Performances dataframes

PERFS = {}

for mode in self.modes:

PERFS[mode] = pd.read\_csv('../data/Performances\_' + self.pred\_type + '\_' + model + '\_' + self.fold +

mode + '.csv')

PERFS[mode].set\_index('outer\_fold', drop=False, inplace=True)

PERFS[mode]['C-Index'] = np.nan

PERFS[mode]['C-Index-difference'] = np.nan

df\_model = self.Survival[['FollowUpTime', 'Death', 'Age', 'res\_' + model]].dropna()

df\_model.rename(columns={'res\_' + model: 'pred'}, inplace=True)

# Compute CI over all samples

if len(df\_model['Death'].unique()) == 2:

ci\_model = concordance\_index(df\_model['FollowUpTime'], -(df\_model['Age'] - df\_model['pred']),

df\_model['Death'])

ci\_age = concordance\_index(df\_model['FollowUpTime'], -df\_model['Age'], df\_model['Death'])

ci\_diff = ci\_model - ci\_age

PERFS[''].loc['all', 'C-Index'] = ci\_model

PERFS[''].loc['all', 'C-Index-difference'] = ci\_diff

self.PERFORMANCES.loc[model, 'C-Index\_all'] = ci\_model

self.PERFORMANCES.loc[model, 'C-Index-difference\_all'] = ci\_diff

\_, ci\_sd = self.\_bootstrap\_c\_index(df\_model)

PERFS['\_sd'].loc['all', 'C-Index'] = ci\_sd

PERFS['\_sd'].loc['all', 'C-Index-difference'] = ci\_sd

self.PERFORMANCES.loc[model, 'C-Index\_sd\_all'] = ci\_sd

self.PERFORMANCES.loc[model, 'C-Index-difference\_sd\_all'] = ci\_sd

# Compute CI over each fold

for i in range(10):

df\_model\_i = self.SURV[i][['FollowUpTime', 'Death', 'Age', 'res\_' + model]].dropna()

df\_model\_i.rename(columns={'res\_' + model: 'pred'}, inplace=True)

if len(df\_model\_i['Death'].unique()) == 2:

ci\_model\_i = concordance\_index(df\_model\_i['FollowUpTime'],

-(df\_model\_i['Age'] - df\_model\_i['pred']),

df\_model\_i['Death'])

ci\_age\_i = concordance\_index(df\_model\_i['FollowUpTime'], -df\_model\_i['Age'], df\_model\_i['Death'])

ci\_diff\_i = ci\_model\_i - ci\_age\_i

PERFS[''].loc[str(i), 'C-Index'] = ci\_model\_i

PERFS[''].loc[str(i), 'C-Index-difference'] = ci\_diff\_i

self.PERFORMANCES.loc[model, 'C-Index\_' + str(i)] = ci\_model\_i

self.PERFORMANCES.loc[model, 'C-Index-difference\_' + str(i)] = ci\_diff\_i

\_, ci\_i\_sd = self.\_bootstrap\_c\_index(df\_model\_i)

PERFS['\_sd'].loc[str(i), 'C-Index'] = ci\_i\_sd

PERFS['\_sd'].loc[str(i), 'C-Index-difference'] = ci\_i\_sd

self.PERFORMANCES.loc[model, 'C-Index\_sd\_' + str(i)] = ci\_i\_sd

self.PERFORMANCES.loc[model, 'C-Index-difference\_sd\_' + str(i)] = ci\_i\_sd

# Compute sd using all folds

ci\_str = round(PERFS[''][['C-Index', 'C-Index-difference']], 3).astype(str) + '+-' + \

round(PERFS['\_sd'][['C-Index', 'C-Index-difference']], 3).astype(str)

PERFS['\_str'][['C-Index', 'C-Index-difference']] = ci\_str

for col in ['C-Index', 'C-Index-difference']:

cols = [col + '\_str\_' + str(i) for i in range(10)]

# Fill model's performance matrix

ci\_std\_lst = PERFS['\_str'].loc['all', col].split('+-')

ci\_std\_lst.insert(1, str(round(PERFS[''][col].iloc[1:].std(), 3)))

ci\_std\_str = '+-'.join(ci\_std\_lst)

PERFS['\_str'].loc['all', col] = ci\_std\_str

# Fill global performances matrix

self.PERFORMANCES.loc[model, cols] = ci\_str[col].values[1:]

self.PERFORMANCES.loc[model, col + '\_str\_all'] = ci\_std\_str

# Save new performances

for mode in self.modes:

PERFS[mode].to\_csv('../data/Performances\_' + self.pred\_type + '\_withCI\_' + model + '\_' + self.fold +

mode + '.csv')

# Ranking, printing and saving

# Sort by alphabetical order

Performances\_alphabetical = self.PERFORMANCES.sort\_values(by='version')

Performances\_alphabetical.to\_csv(self.path\_data + 'PERFORMANCES\_withEnsembles\_withCI\_alphabetical\_' +

self.pred\_type + '\_' + self.target + '\_' + self.fold + '.csv', index=False)

# Sort by C-Index difference, to print

cols\_to\_print = ['version', 'C-Index-difference\_str\_all']

Performances\_ranked = self.PERFORMANCES.sort\_values(by='C-Index-difference\_all', ascending=False)

print('Performances of the models ranked by C-Index difference with C-Index based on age only,'

' on all the samples:')

print(Performances\_ranked[cols\_to\_print])

# Sort by main metric, to save

sort\_by = self.dict\_main\_metrics\_names[self.target] + '\_all'

sort\_ascending = self.main\_metrics\_modes[self.dict\_main\_metrics\_names[self.target]] == 'min'

Performances\_ranked = self.PERFORMANCES.sort\_values(by=sort\_by, ascending=sort\_ascending)

Performances\_ranked.to\_csv(self.path\_data + 'PERFORMANCES\_withEnsembles\_withCI\_withEnsembles\_ranked\_' +

self.pred\_type + '\_' + self.target + '\_' + self.fold + '.csv', index=False)

# Save with ensembles

models\_nonensembles = [idx for idx in Performances\_alphabetical.index if '\*' not in idx]

path\_save = self.path\_data + 'PERFORMANCES\_withoutEnsembles\_withCI\_alphabetical\_' + self.pred\_type + '\_' + \

self.target + '\_' + self.fold + '.csv'

Performances\_alphabetical.loc[models\_nonensembles, :].to\_csv(path\_save, index=False)

Performances\_ranked.loc[models\_nonensembles, :].to\_csv(path\_save.replace('alphabetical', 'ranked'))

def print\_key\_results(self):

# Helper function

def compute\_p\_value(row):

sd = float(row['C-Index-difference\_str\_all'].split('+-')[1])

z = np.abs(row['C-Index-difference\_all']) / sd

pv = norm.sf(abs(z)) \* 2

return pv

# Preprocess the data

Performances = pd.read\_csv(

self.path\_data + 'PERFORMANCES\_withEnsembles\_withCI\_alphabetical\_' + self.pred\_type + '\_' +

self.target + '\_' + self.fold + '.csv')

Performances.set\_index('version', drop=False, inplace=True)

Perfs\_CI = Performances[['version', 'C-Index\_all', 'C-Index-difference\_all',

'C-Index-difference\_str\_all']].sort\_values(by='C-Index-difference\_all')

Perfs\_CI['C-Index\_CA'] = Perfs\_CI['C-Index\_all'] - Perfs\_CI['C-Index-difference\_all']

Perfs\_CI['p-value'] = Perfs\_CI.apply(compute\_p\_value, axis=1)

# Select only models for which difference between biological age's CI and chronological age's CI is significant

Perfs\_CI\_significant = Perfs\_CI[Perfs\_CI['p-value'] < 0.05]

Perfs\_CI\_significant\_FDR = Perfs\_CI[Perfs\_CI['p-value'] \* len(Perfs\_CI.index) < 0.05]

# Take the subset corresponding to the 11 main dimensions

main\_dims = ['Brain', 'Eyes', 'Hearing', 'Lungs', 'Arterial', 'Heart', 'Abdomen', 'Musculoskeletal',

'PhysicalActivity', 'Biochemistry', 'ImmuneSystem']

main\_rows = ['Age\_' + dim + '\_\*' \* 10 for dim in main\_dims]

Perfs\_CI\_main = Perfs\_CI.loc[main\_rows, :]

Perfs\_CI\_main.sort\_values(by='C-Index-difference\_all', inplace=True)

# Select only models for which difference between biological age's CI and chronological age's CI is significant

Perfs\_CI\_main\_significant = Perfs\_CI\_main[Perfs\_CI\_main['p-value'] < 0.05]

Perfs\_CI\_main\_significant\_FDR = Perfs\_CI\_main[Perfs\_CI\_main['p-value'] \* len(Perfs\_CI\_main.index) < 0.05]

# Compute the statistics to compare biological ages and chronological age on all the dimensions

CI\_diff\_mean = Perfs\_CI['C-Index-difference\_all'].mean()

CI\_diff\_std = Perfs\_CI['C-Index-difference\_all'].std()

\_t\_stat\_all, pv\_all = ttest\_rel(Perfs\_CI['C-Index\_all'], Perfs\_CI['C-Index\_CA'])

# Number of dimensions outperforming and underperforming compared to chronological age

n\_CI\_diff\_positives = (Perfs\_CI['C-Index-difference\_all'] > 0).sum()

n\_CI\_diff\_negatives = (Perfs\_CI['C-Index-difference\_all'] < 0).sum()

n\_CI\_diff\_positives\_significant = (Perfs\_CI\_significant['C-Index-difference\_all'] > 0).sum()

n\_CI\_diff\_negatives\_significant = (Perfs\_CI\_significant['C-Index-difference\_all'] < 0).sum()

n\_CI\_diff\_positives\_significant\_FDR = (Perfs\_CI\_significant\_FDR['C-Index-difference\_all'] > 0).sum()

n\_CI\_diff\_negatives\_significant\_FDR = (Perfs\_CI\_significant\_FDR['C-Index-difference\_all'] < 0).sum()

# print results

print('The mean CI difference over the ' + str(len(Perfs\_CI.index)) + ' biological ages = ' +

str(round(CI\_diff\_mean, 3)) + '; standard deviation = ' + str(round(CI\_diff\_std, 3)) +

'; paired t-test p-value = ' + str(pv\_all))

print('Out of the ' + str(len(Perfs\_CI.index)) + ' dimensions, ' + str(n\_CI\_diff\_positives) +

' dimensions outperform CA as survival predictors, and ' + str(n\_CI\_diff\_negatives) +

' dimensions underperform.')

# Compute the statistics to compare biological ages and chronological age on the 11 main dimensions

CI\_diff\_main\_mean = Perfs\_CI\_main['C-Index-difference\_all'].mean()

CI\_diff\_main\_std = Perfs\_CI\_main['C-Index-difference\_all'].std()

\_t\_stat\_main, pv\_main = ttest\_rel(Perfs\_CI\_main['C-Index\_all'], Perfs\_CI\_main['C-Index\_CA'])

# Number of dimensions outperforming and underperforming compared to chronological age

n\_CI\_diff\_main\_positives = (Perfs\_CI\_main['C-Index-difference\_all'] > 0).sum()

n\_CI\_diff\_main\_negatives = (Perfs\_CI\_main['C-Index-difference\_all'] < 0).sum()

n\_CI\_diff\_main\_positives\_significant = (Perfs\_CI\_main\_significant['C-Index-difference\_all'] > 0).sum()

n\_CI\_diff\_main\_negatives\_significant = (Perfs\_CI\_main\_significant['C-Index-difference\_all'] < 0).sum()

n\_CI\_diff\_main\_positives\_significant\_FDR = (Perfs\_CI\_main\_significant\_FDR['C-Index-difference\_all'] > 0).sum()

n\_CI\_diff\_main\_negatives\_significant\_FDR = (Perfs\_CI\_main\_significant\_FDR['C-Index-difference\_all'] < 0).sum()

# print results

print('The mean CI difference over the ' + str(len(Perfs\_CI\_main.index)) + ' biological ages = ' +

str(round(CI\_diff\_main\_mean, 3)) + '; standard deviation = ' + str(round(CI\_diff\_main\_std, 3)) +

'; paired t-test p-value = ' + str(pv\_main))

print('Out of the ' + str(len(Perfs\_CI\_main.index)) + ' main biological dimensions, ' + str(

n\_CI\_diff\_main\_positives) +

' dimensions outperform CA as survival predictors, and ' + str(n\_CI\_diff\_main\_negatives) +

' dimensions underperform.')

Perfs\_CI\_main[['version', 'C-Index-difference\_all',

'C-Index-difference\_str\_all', 'C-Index\_all', 'C-Index\_CA']].sort\_values(

by='C-Index-difference\_all')

row\_names = ['All', 'significant', 'FDR\_significant']

col\_names = ['All', '+', '-']

n\_models = pd.DataFrame(np.empty((len(row\_names), len(col\_names),)))

n\_models.index = row\_names

n\_models.columns = col\_names

N\_MODELS = {'All\_dims': n\_models.copy(), 'Main\_dims': n\_models.copy()}

best\_models = n\_models.drop(columns=['All'])

BEST\_MODELS = {'All\_dims': best\_models.copy(), 'Main\_dims': best\_models.copy()}

BEST\_CI\_DIFFS = {'All\_dims': best\_models.copy(), 'Main\_dims': best\_models.copy()}

N\_MODELS['All\_dims'].loc[:, '+'] = \

[n\_CI\_diff\_positives, n\_CI\_diff\_positives\_significant, n\_CI\_diff\_positives\_significant\_FDR]

BEST\_MODELS['All\_dims'].loc[:, '+'] = [Perfs\_CI['version'][len(Perfs\_CI.index) - 1],

Perfs\_CI\_significant['version'][len(Perfs\_CI\_significant.index) - 1],

Perfs\_CI\_significant\_FDR['version'][

len(Perfs\_CI\_significant\_FDR.index) - 1]]

BEST\_CI\_DIFFS['All\_dims'].loc[:, '+'] = \

[Perfs\_CI['C-Index-difference\_str\_all'][len(Perfs\_CI.index) - 1],

Perfs\_CI\_significant['C-Index-difference\_str\_all'][len(Perfs\_CI\_significant.index) - 1],

Perfs\_CI\_significant\_FDR['C-Index-difference\_str\_all'][len(Perfs\_CI\_significant\_FDR.index) - 1]]

N\_MODELS['All\_dims'].loc[:, '-'] = \

[n\_CI\_diff\_negatives, n\_CI\_diff\_negatives\_significant, n\_CI\_diff\_negatives\_significant\_FDR]

BEST\_MODELS['All\_dims'].loc[:, '-'] = [Perfs\_CI['version'][0],

Perfs\_CI\_significant['version'][0],

Perfs\_CI\_significant\_FDR['version'][0]]

BEST\_CI\_DIFFS['All\_dims'].loc[:, '-'] = [Perfs\_CI['C-Index-difference\_str\_all'][0],

Perfs\_CI\_significant['C-Index-difference\_str\_all'][0],

Perfs\_CI\_significant\_FDR['C-Index-difference\_str\_all'][0]]

N\_MODELS['All\_dims']['All'] = N\_MODELS['All\_dims']['+'] + N\_MODELS['All\_dims']['-']

N\_MODELS['Main\_dims'].loc[:, '+'] = \

[n\_CI\_diff\_main\_positives, n\_CI\_diff\_main\_positives\_significant, n\_CI\_diff\_main\_positives\_significant\_FDR]

BEST\_MODELS['Main\_dims'].loc[:, '+'] = \

[Perfs\_CI\_main['version'][len(Perfs\_CI\_main.index) - 1],

Perfs\_CI\_main\_significant['version'][len(Perfs\_CI\_main\_significant.index) - 1],

Perfs\_CI\_main\_significant\_FDR['version'][len(Perfs\_CI\_main\_significant\_FDR.index) - 1]]

BEST\_CI\_DIFFS['Main\_dims'].loc[:, '+'] = \

[Perfs\_CI\_main['C-Index-difference\_str\_all'][len(Perfs\_CI\_main.index) - 1],

Perfs\_CI\_main\_significant['C-Index-difference\_str\_all'][len(Perfs\_CI\_main\_significant.index) - 1],

Perfs\_CI\_main\_significant\_FDR['C-Index-difference\_str\_all'][len(Perfs\_CI\_main\_significant\_FDR.index) - 1]]

N\_MODELS['Main\_dims'].loc[:, '-'] = \

[n\_CI\_diff\_main\_negatives, n\_CI\_diff\_main\_negatives\_significant, n\_CI\_diff\_main\_negatives\_significant\_FDR]

BEST\_MODELS['Main\_dims'].loc[:, '-'] = [Perfs\_CI\_main['version'][0],

Perfs\_CI\_main\_significant['version'][0],

Perfs\_CI\_main\_significant\_FDR['version'][0]]

BEST\_CI\_DIFFS['Main\_dims'].loc[:, '-'] = [Perfs\_CI\_main['C-Index-difference\_str\_all'][0],

Perfs\_CI\_main\_significant['C-Index-difference\_str\_all'][0],

Perfs\_CI\_main\_significant\_FDR['C-Index-difference\_str\_all'][0]]

N\_MODELS['Main\_dims']['All'] = N\_MODELS['Main\_dims']['+'] + N\_MODELS['Main\_dims']['-']

# Reformat to take into account that sometimes no model fits the criteria

for dims in ['All\_dims', 'Main\_dims']:

for sign in ['+', '-']:

for models in ['All', 'significant', 'FDR\_significant']:

if N\_MODELS[dims].loc[models, sign] == 0:

BEST\_MODELS[dims].loc[models, sign] = ''

BEST\_CI\_DIFFS[dims].loc[models, sign] = ''

# Print results

# All dims

print('Number of aging dimensions, best models and associated CI differences for All dims: ')

print(N\_MODELS['All\_dims'])

print(BEST\_MODELS['All\_dims'])

print(BEST\_CI\_DIFFS['All\_dims'])

print('Best model between All dims: ')

print(Perfs\_CI\_significant\_FDR[['C-Index-difference\_str\_all', 'C-Index\_all', 'C-Index\_CA']].iloc[-1, :])

# Main dims

print('Number of aging dimensions, best models and associated CI differences for Main dims: ')

print(N\_MODELS['Main\_dims'])

print(BEST\_MODELS['Main\_dims'])

print(BEST\_CI\_DIFFS['Main\_dims'])

print('Best model between Main dims: ')

print(Perfs\_CI\_main\_significant\_FDR[['C-Index-difference\_str\_all', 'C-Index\_all', 'C-Index\_CA']].iloc[-1, :])

class SelectBest(Metrics):

"""

For each aging main dimension and selected subdimensions, select the best performing model.

"""

def \_\_init\_\_(self, target=None, pred\_type=None):

Metrics.\_\_init\_\_(self)

self.target = target

self.pred\_type = pred\_type

self.folds = ['test']

self.organs\_with\_suborgans = {'Brain': ['Cognitive', 'MRI'], 'Eyes': ['All', 'Fundus', 'OCT'],

'Arterial': ['PulseWaveAnalysis', 'Carotids'],

'Heart': ['ECG', 'MRI'], 'Abdomen': ['Liver', 'Pancreas'],

'Musculoskeletal': ['Spine', 'Hips', 'Knees', 'FullBody', 'Scalars'],

'Biochemistry': ['Urine', 'Blood']}

self.organs = []

self.best\_models = []

self.PREDICTIONS = {}

self.RESIDUALS = {}

self.PERFORMANCES = {}

self.CORRELATIONS = {}

self.CORRELATIONS\_SAMPLESIZES = {}

def \_load\_data(self):

for fold in self.folds:

path\_pred = self.path\_data + 'PREDICTIONS\_withEnsembles\_' + self.pred\_type + '\_' + self.target + '\_' + \

fold + '.csv'

path\_res = self.path\_data + 'RESIDUALS\_' + self.pred\_type + '\_' + self.target + '\_' + fold + '.csv'

path\_perf = self.path\_data + 'PERFORMANCES\_withEnsembles\_withCI\_ranked\_' + self.pred\_type + '\_' + \

self.target + '\_' + fold + '.csv'

path\_corr = self.path\_data + 'ResidualsCorrelations\_str\_' + self.pred\_type + '\_' + self.target + '\_' + \

fold + '.csv'

self.PREDICTIONS[fold] = pd.read\_csv(path\_pred)

self.RESIDUALS[fold] = pd.read\_csv(path\_res)

self.PERFORMANCES[fold] = pd.read\_csv(path\_perf)

self.PERFORMANCES[fold].set\_index('version', drop=False, inplace=True)

self.CORRELATIONS\_SAMPLESIZES[fold] = pd.read\_csv(self.path\_data + 'ResidualsCorrelations\_samplesizes\_' +

self.pred\_type + '\_' + self.target + '\_' + fold + '.csv',

index\_col=0)

self.CORRELATIONS[fold] = {}

for mode in self.modes:

self.CORRELATIONS[fold][mode] = pd.read\_csv(path\_corr.replace('\_str', mode), index\_col=0)

def \_select\_versions(self):

# Load val performances

path\_perf = self.path\_data + 'PERFORMANCES\_withEnsembles\_withCI\_ranked\_' + self.pred\_type + '\_' + \

self.target + '\_test.csv'

Performances = pd.read\_csv(path\_perf)

Performances.set\_index('version', drop=False, inplace=True)

list\_organs = Performances['organ'].unique()

list\_organs.sort()

for organ in list\_organs:

print('Selecting best model for ' + organ)

Perf\_organ = Performances[Performances['organ'] == organ]

self.organs.append(organ)

self.best\_models.append(Perf\_organ['version'].values[0])

if organ in self.organs\_with\_suborgans.keys():

for view in self.organs\_with\_suborgans[organ]:

print('Selecting best model for ' + organ + view)

Perf\_organview = Performances[(Performances['organ'] == organ) & (Performances['view'] == view)]

self.organs.append(organ + view)

self.best\_models.append(Perf\_organview['version'].values[0])

def \_take\_subsets(self):

base\_cols = self.id\_vars + self.demographic\_vars

best\_models\_pred = ['pred\_' + model for model in self.best\_models]

best\_models\_res = ['res\_' + model for model in self.best\_models]

best\_models\_corr = ['\_'.join(model.split('\_')[1:]) for model in self.best\_models]

for fold in self.folds:

self.PREDICTIONS[fold] = self.PREDICTIONS[fold].loc[:, base\_cols + best\_models\_pred]

self.PREDICTIONS[fold].columns = base\_cols + self.organs

self.RESIDUALS[fold] = self.RESIDUALS[fold].loc[:, base\_cols + best\_models\_res]

self.RESIDUALS[fold].columns = base\_cols + self.organs

self.PERFORMANCES[fold] = self.PERFORMANCES[fold].loc[self.best\_models, :]

self.PERFORMANCES[fold].index = self.organs

self.CORRELATIONS\_SAMPLESIZES[fold] = \

self.CORRELATIONS\_SAMPLESIZES[fold].loc[best\_models\_corr, best\_models\_corr]

self.CORRELATIONS\_SAMPLESIZES[fold].index = self.organs

self.CORRELATIONS\_SAMPLESIZES[fold].columns = self.organs

for mode in self.modes:

self.CORRELATIONS[fold][mode] = self.CORRELATIONS[fold][mode].loc[best\_models\_corr, best\_models\_corr]

self.CORRELATIONS[fold][mode].index = self.organs

self.CORRELATIONS[fold][mode].columns = self.organs

def select\_models(self):

self.\_load\_data()

self.\_select\_versions()

self.\_take\_subsets()

def save\_data(self):

for fold in self.folds:

path\_pred = self.path\_data + 'PREDICTIONS\_bestmodels\_' + self.pred\_type + '\_' + self.target + '\_' + fold \

+ '.csv'

path\_res = self.path\_data + 'RESIDUALS\_bestmodels\_' + self.pred\_type + '\_' + self.target + '\_' + fold + \

'.csv'

path\_corr = self.path\_data + 'ResidualsCorrelations\_bestmodels\_str\_' + self.pred\_type + '\_' + self.target \

+ '\_' + fold + '.csv'

path\_perf = self.path\_data + 'PERFORMANCES\_bestmodels\_ranked\_' + self.pred\_type + '\_' + self.target + '\_' \

+ fold + '.csv'

self.PREDICTIONS[fold].to\_csv(path\_pred, index=False)

self.RESIDUALS[fold].to\_csv(path\_res, index=False)

self.PERFORMANCES[fold].sort\_values(by=self.dict\_main\_metrics\_names[self.target] + '\_all', ascending=False,

inplace=True)

self.PERFORMANCES[fold].to\_csv(path\_perf, index=False)

Performances\_alphabetical = self.PERFORMANCES[fold].sort\_values(by='version')

Performances\_alphabetical.to\_csv(path\_perf.replace('ranked', 'alphabetical'), index=False)

for mode in self.modes:

self.CORRELATIONS[fold][mode].to\_csv(path\_corr.replace('\_str', mode), index=True)

# Handy draft to print some key results

Perfs = pd.read\_csv('../data/PERFORMANCES\_withEnsembles\_alphabetical\_instances\_Age\_test.csv')

Perfs.set\_index('version', drop=False, inplace=True)

# Take the subset corresponding to the 11 main dimensions

main\_dims = ['Brain', 'Eyes', 'Hearing', 'Lungs', 'Arterial', 'Heart', 'Abdomen', 'Musculoskeletal',

'PhysicalActivity',

'Biochemistry', 'ImmuneSystem']

main\_rows = ['Age\_' + dim + '\_\*' \* 10 for dim in main\_dims]

Perfs\_main = Perfs.loc[main\_rows, :]

print('R-Squared for all dimensions = ' + str(round(Perfs['R-Squared\_all'].mean(), 3)) + '; std = ' +

str(round(Perfs['R-Squared\_all'].std(), 3)) + '; min = ' + str(round(Perfs['R-Squared\_all'].min(), 3)) +

'; max = ' + str(round(Perfs['R-Squared\_all'].max(), 3)))

print('RMSEs for all dimensions = ' + str(round(Perfs['RMSE\_all'].mean(), 3)) + '; std = ' +

str(round(Perfs['RMSE\_all'].std(), 3)) + '; min = ' + str(

round(Perfs['RMSE\_all'].min(), 3)) + '; max = ' +

str(round(Perfs['RMSE\_all'].max(), 3)))

print('R-Squared for main dimensions = ' + str(round(Perfs\_main['R-Squared\_all'].mean(), 3)) + '; std = ' +

str(round(Perfs\_main['R-Squared\_all'].std(), 3)) + '; min = ' + str(

round(Perfs\_main['R-Squared\_all'].min(), 3)) +

'; max = ' + str(round(Perfs\_main['R-Squared\_all'].max(), 3)))

print('RMSEs for main dimensions = ' + str(round(Perfs\_main['RMSE\_all'].mean(), 3)) + '; std = ' +

str(round(Perfs\_main['RMSE\_all'].std(), 3)) + '; min = ' + str(round(Perfs\_main['RMSE\_all'].min(), 3)) +

'; max = ' + str(round(Perfs\_main['RMSE\_all'].max(), 3)))

class SelectCorrelationsNAs(Basics):

"""

Build a summary correlation matrix: when a correlation cannot be computed in terms of samples ("instances") because

the intersection has a small sample size, fill the NA with the correlation computed at the participant's level

("eids").

"""

def \_\_init\_\_(self, target=None):

Basics.\_\_init\_\_(self)

self.target = target

self.folds = ['test']

self.CORRELATIONS = {'\*': {'': {}, '\_sd': {}, '\_str': {}}}

def load\_data(self):

for models\_type in self.models\_types:

self.CORRELATIONS[models\_type] = {}

for pred\_type in ['instances', 'eids', '\*']:

self.CORRELATIONS[models\_type][pred\_type] = {}

for mode in self.modes:

self.CORRELATIONS[models\_type][pred\_type][mode] = {}

for fold in self.folds:

if pred\_type == '\*':

self.CORRELATIONS[models\_type][pred\_type][mode][fold] = \

pd.read\_csv(self.path\_data + 'ResidualsCorrelations' + models\_type + mode +

'\_instances\_' + self.target + '\_' + fold + '.csv', index\_col=0)

else:

self.CORRELATIONS[models\_type][pred\_type][mode][fold] = \

pd.read\_csv(self.path\_data + 'ResidualsCorrelations' + models\_type + mode + '\_' +

pred\_type + '\_' + self.target + '\_' + fold + '.csv', index\_col=0)

def fill\_na(self):

# Dectect NAs in the instances correlation matrix

for models\_type in self.models\_types:

NAs\_mask = self.CORRELATIONS[models\_type]['instances']['']['test'].isna()

for mode in self.modes:

for fold in self.folds:

self.CORRELATIONS[models\_type]['\*'][mode][fold] = \

self.CORRELATIONS[models\_type]['instances'][mode][fold].copy()

self.CORRELATIONS[models\_type]['\*'][mode][fold][NAs\_mask] = \

self.CORRELATIONS[models\_type]['eids'][mode][fold][NAs\_mask]

def save\_correlations(self):

for models\_type in self.models\_types:

for mode in self.modes:

for fold in self.folds:

self.CORRELATIONS[models\_type]['\*'][mode][fold].to\_csv(self.path\_data + 'ResidualsCorrelations' +

models\_type + mode + '\_\*\_' + self.target +

'\_' + fold + '.csv', index=True)

class CorrelationsAverages:

"""

Computes average correlation at different levels, to summarize the results.

"""

def \_\_init\_\_(self):

self.Performances = pd.read\_csv("../data/PERFORMANCES\_withEnsembles\_ranked\_eids\_Age\_test.csv")

self.Correlations = pd.read\_csv("../data/ResidualsCorrelations\_eids\_Age\_test.csv", index\_col=0)

def \_melt\_correlation\_matrix(self, models):

models = ['\_'.join(c.split('\_')[1:]) for c in models]

Corrs = self.Correlations.loc[models, models]

Corrs = Corrs.where(np.triu(np.ones(Corrs.shape), 1).astype(np.bool))

Corrs = Corrs.stack().reset\_index()

Corrs.columns = ['Row', 'Column', 'Correlation']

return Corrs

@staticmethod

def \_split\_version(row):

names = ['organ', 'view', 'transformation', 'architecture', 'n\_fc\_layers', 'n\_fc\_nodes', 'optimizer',

'learning\_rate', 'weight\_decay', 'dropout\_rate', 'data\_augmentation\_factor']

row\_names = ['row\_' + name for name in names]

col\_names = ['col\_' + name for name in names]

row\_params = row['Row'].split('\_')

col\_params = row['Column'].split('\_')

new\_row = pd.Series(row\_params + col\_params + [row['Correlation']])

new\_row.index = row\_names + col\_names + ['Correlation']

return new\_row

@staticmethod

def \_compute\_stats(data, title):

m = data['Correlation'].mean()

s = data['Correlation'].std()

n = len(data.index)

print('Correlation between ' + title + ': ' + str(round(m, 3)) + '+-' + str(round(s, 3)) + ', n\_pairs=' +

str(n))

@staticmethod

def \_generate\_pairs(ls):

pairs = []

for i in range(len(ls)):

for j in range((i + 1), len(ls)):

pairs.append((ls[i], ls[j]))

return pairs

@staticmethod

def \_extract\_pair(Corrs, pair, level):

extracted = Corrs[((Corrs['row\_' + level] == pair[0]) & (Corrs['col\_' + level] == pair[1])) |

((Corrs['row\_' + level] == pair[1]) & (Corrs['col\_' + level] == pair[0]))]

return extracted

def \_extract\_pairs(self, Corrs, pairs, level):

extracted = None

for pair in pairs:

extracted\_pair = self.\_extract\_pair(Corrs, pair, level)

if extracted is None:

extracted = extracted\_pair

else:

extracted = extracted.append(extracted\_pair)

return extracted

def correlations\_all(self):

Corrs = self.\_melt\_correlation\_matrix(self.Performances['version'].values)

self.\_compute\_stats(Corrs, 'All models')

def correlations\_dimensions(self):

Perf = self.Performances[(self.Performances['view'] == '\*') &

~(self.Performances['organ'].isin(['\*', '\*instances01', '\*instances1.5x',

'\*instances23']))]

Corrs = self.\_melt\_correlation\_matrix(Perf['version'].values)

self.\_compute\_stats(Corrs, 'Main Dimensions')

def correlations\_subdimensions(self):

# Subdimensions

dict\_dims\_to\_subdims = {

'Brain': ['Cognitive', 'MRI'],

'Eyes': ['OCT', 'Fundus', 'IntraocularPressure', 'Acuity', 'Autorefraction'],

'Arterial': ['PulseWaveAnalysis', 'Carotids'],

'Heart': ['ECG', 'MRI'],

'Abdomen': ['Liver', 'Pancreas'],

'Musculoskeletal': ['Spine', 'Hips', 'Knees', 'FullBody', 'Scalars'],

'PhysicalActivity': ['FullWeek', 'Walking'],

'Biochemistry': ['Urine', 'Blood']

}

Corrs\_subdim = None

for dim in dict\_dims\_to\_subdims.keys():

models = ['Age\_' + dim + '\_' + subdim + '\_\*' \* 9 for subdim in dict\_dims\_to\_subdims[dim]]

Corrs\_dim = self.\_melt\_correlation\_matrix(models)

self.\_compute\_stats(Corrs\_dim, dim + ' subdimensions')

if Corrs\_subdim is None:

Corrs\_subdim = Corrs\_dim

else:

Corrs\_subdim = Corrs\_subdim.append(Corrs\_dim)

# Compute the average over the subdimensions

self.\_compute\_stats(Corrs\_subdim, 'Subdimensions')

def correlations\_subsubdimensions(self):

# Only select the ensemble models at the architecture level,

Perf\_ss = self.Performances[self.Performances['architecture'] == '\*']

# Brain - Cognitive

Perf = Perf\_ss[(Perf\_ss['organ'] == 'Brain') & (Perf\_ss['view'] == 'Cognitive')]

# Remove ensemble model and all scalars

Perf = Perf[~Perf['transformation'].isin(['\*', 'AllScalars'])]

Corrs\_bc = self.\_melt\_correlation\_matrix(Perf['version'].values)

self.\_compute\_stats(Corrs\_bc, 'Brain cognitive sub-subdimensions')

# Musculoskeletal - Scalars

Perf = Perf\_ss[(Perf\_ss['organ'] == 'Musculoskeletal') & (Perf\_ss['view'] == 'Scalars')]

Perf = Perf[~Perf['transformation'].isin(['\*', 'AllScalars'])]

Corrs\_ms = self.\_melt\_correlation\_matrix(Perf['version'].values)

self.\_compute\_stats(Corrs\_ms, 'Musculoskeletal - Scalars sub-subdimensions')

# Average over subsubdimensions

Corrs\_subsubdimensions = Corrs\_bc.append(Corrs\_ms)

self.\_compute\_stats(Corrs\_subsubdimensions, 'Sub-subdimensions')

def correlations\_views(self):

# Variables

dict\_dim\_to\_view = {

'Brain\_MRI': [['SagittalReference', 'CoronalReference', 'TransverseReference', 'dMRIWeightedMeans',

'SubcorticalVolumes', 'GreyMatterVolumes'],

['SagittalRaw', 'CoronalRaw', 'TransverseRaw']],

'Arterial\_PulseWaveAnalysis': [['Scalars', 'TimeSeries']],

'Arterial\_Carotids': [['Scalars', 'LongAxis', 'CIMT120', 'CIMT150', 'ShortAxis']],

'Heart\_ECG': [['Scalars', 'TimeSeries']],

'Heart\_MRI': [['2chambersRaw', '3chambersRaw', '4chambersRaw'],

['2chambersContrast', '3chambersContrast', '4chambersContrast']],

'Musculoskeletal\_Spine': [['Sagittal', 'Coronal']],

'Musculoskeletal\_FullBody': [['Figure', 'Flesh']],

'PhysicalActivity\_FullWeek': [

['Scalars', 'Acceleration', 'TimeSeriesFeatures', 'GramianAngularField1minDifference',

'GramianAngularField1minSummation', 'MarkovTransitionField1min',

'RecurrencePlots1min']]

}

Corrs\_views = None

for dim in dict\_dim\_to\_view.keys():

Corrs\_dims = None

for i, views in enumerate(dict\_dim\_to\_view[dim]):

models = ['Age\_' + dim + '\_' + view + '\_\*' \* 8 for view in dict\_dim\_to\_view[dim][i]]

Corrs\_dim = self.\_melt\_correlation\_matrix(models)

if Corrs\_dims is None:

Corrs\_dims = Corrs\_dim

else:

Corrs\_dims = Corrs\_dims.append(Corrs\_dim)

self.\_compute\_stats(Corrs\_dims, dim + ' views')

if Corrs\_views is None:

Corrs\_views = Corrs\_dims

else:

Corrs\_views = Corrs\_views.append(Corrs\_dims)

# Compute the average over the views

self.\_compute\_stats(Corrs\_views, 'Views')

def correlations\_transformations(self):

# Raw vs. Contrast (Heart MRI, Abdomen Liver, Abdomen Pancreas), Raw vs. Reference (Brain MRI),

# Figure vs Skeleton (Musculoskeltal FullBody)

# Filter out the models that are ensembles at the architecture level

models\_to\_keep = [model for model in self.Correlations.index.values if model.split('\_')[3] != '\*']

# Select only the models that are Heart MRI, Abdomen, or Brain MRI

models\_to\_keep = [model for model in models\_to\_keep if

((model.split('\_')[0] == 'Abdomen') & (model.split('\_')[1] in ['Liver', 'Pancreas'])) |

((model.split('\_')[0] == 'Brain') & (model.split('\_')[1] == 'MRI')) |

((model.split('\_')[0] == 'Heart') & (model.split('\_')[1] == 'MRI')) |

((model.split('\_')[0] == 'Musculoskeletal') & (model.split('\_')[1] == 'FullBody') &

(model.split('\_')[2] in ['Figure', 'Skeleton']))]

# Select only the models that have the relevant preprocessing/transformations

models\_to\_keep = [model for model in models\_to\_keep if model.split('\_')[2] in

['Raw', 'Contrast', '2chambersRaw', '2chambersContrast', '3chambersRaw', '3chambersContrast',

'4chambersRaw', '4chambersContrast', 'SagittalRaw', 'SagittalReference', 'CoronalRaw',

'CoronalReference', 'TransverseRaw', 'TransverseReference', 'Figure', 'Skeleton']]

# Select the corresponding rows and columns

Corrs = self.Correlations.loc[models\_to\_keep, models\_to\_keep]

# Melt correlation matrix to dataframe

Corrs = Corrs.where(np.triu(np.ones(Corrs.shape), 1).astype(np.bool))

Corrs = Corrs.stack().reset\_index()

Corrs.columns = ['Row', 'Column', 'Correlation']

Corrs = Corrs.apply(self.\_split\_version, axis=1)

# Only keep the models that have the same organ, view and architecture

Corrs = Corrs[(Corrs['row\_organ'] == Corrs['col\_organ']) & (Corrs['row\_view'] == Corrs['col\_view']) &

(Corrs['row\_architecture'] == Corrs['col\_architecture'])]

# Define preprocessing pairs

dict\_preprocessing = {

'Raw-Reference': [('SagittalRaw', 'SagittalReference'), ('CoronalRaw', 'CoronalReference'),

('TransverseRaw', 'TransverseReference')],

'Raw-Contrast': [('Raw', 'Contrast'), ('2chambersRaw', '2chambersContrast'),

('3chambersRaw', '3chambersContrast'), ('4chambersRaw', '4chambersContrast')],

'Figure-Skeleton': [('Figure', 'Skeleton')]

}

# Compute average correlation between each pair of transformations

Corrs\_transformations = None

for comparison in dict\_preprocessing.keys():

Corrs\_comp = self.\_extract\_pairs(Corrs, dict\_preprocessing[comparison], 'transformation')

print(comparison)

print(Corrs\_comp)

self.\_compute\_stats(Corrs\_comp, comparison)

if Corrs\_transformations is None:

Corrs\_transformations = Corrs\_comp

else:

Corrs\_transformations = Corrs\_transformations.append(Corrs\_comp)

# Compute average correlation between transformations

self.\_compute\_stats(Corrs\_transformations, 'Transformations')

def correlations\_algorithms(self):

# Variables

algorithms\_scalars = ['ElasticNet', 'LightGBM', 'NeuralNetwork']

algorithms\_images = ['InceptionV3', 'InceptionResNetV2']

# Filter out the ensemble models (at the level of the algorithm)

models\_to\_keep = [model for model in self.Correlations.index.values if model.split('\_')[3] != '\*']

Corrs = self.Correlations.loc[models\_to\_keep, models\_to\_keep]

# Melt correlation matrix to dataframe

Corrs = Corrs.where(np.triu(np.ones(Corrs.shape), 1).astype(np.bool))

Corrs = Corrs.stack().reset\_index()

Corrs.columns = ['Row', 'Column', 'Correlation']

Corrs = Corrs.apply(self.\_split\_version, axis=1)

# Select the rows for which everything is identical aside from the dataset

for name in ['organ', 'view', 'transformation']:

Corrs = Corrs[Corrs['row\_' + name] == Corrs['col\_' + name]]

# Compute average correlation between algorithms

self.\_compute\_stats(Corrs, 'Algorithms')

algorithms\_pairs = self.\_generate\_pairs(algorithms\_scalars) + self.\_generate\_pairs(algorithms\_images)

# Compute average correlation between each algorithm pair

for pair in algorithms\_pairs:

Corrs\_pair = self.\_extract\_pair(Corrs, pair, 'architecture')

self.\_compute\_stats(Corrs\_pair, pair[0] + ' and ' + pair[1])

class AttentionMaps(DeepLearning):

"""

Computes the attention maps (saliency maps and Grad\_RAM maps) for all images

"""

def \_\_init\_\_(self, target=None, organ=None, view=None, transformation=None, debug\_mode=False):

# Partial initialization with placeholders to get access to parameters and functions

DeepLearning.\_\_init\_\_(self, 'Age', 'Abdomen', 'Liver', 'Raw', 'InceptionResNetV2', '1', '1024', 'Adam',

'0.0001', '0.1', '0.5', '1.0', False)

# Parameters

self.target = target

self.organ = organ

self.view = view

self.transformation = transformation

self.version = None

self.leftright = True if self.organ + '\_' + self.view in self.left\_right\_organs\_views else False

self.parameters = None

self.image\_width = None

self.image\_height = None

self.batch\_size = None

self.N\_samples\_attentionmaps = 10 # needs to be > 1 for the script to work

if debug\_mode:

self.N\_samples\_attentionmaps = 2

self.dir\_images = '../images/' + organ + '/' + view + '/' + transformation + '/'

self.prediction\_type = self.dict\_prediction\_types[target]

self.Residuals = None

self.df\_to\_plot = None

self.df\_outer\_fold = None

self.class\_mode = None

self.image = None

self.generator = None

self.dict\_architecture\_to\_last\_conv\_layer\_name = \

{'VGG16': 'block5\_conv3', 'VGG19': 'block5\_conv4', 'MobileNet': 'conv\_pw\_13\_relu',

'MobileNetV2': 'out\_relu', 'DenseNet121': 'relu', 'DenseNet169': 'relu', 'DenseNet201': 'relu',

'NASNetMobile': 'activation\_1136', 'NASNetLarge': 'activation\_1396', 'Xception': 'block14\_sepconv2\_act',

'InceptionV3': 'mixed10', 'InceptionResNetV2': 'conv\_7b\_ac', 'EfficientNetB7': 'top\_activation'}

self.last\_conv\_layer = None

self.organs\_views\_transformations\_images = \

['Brain\_MRI\_SagittalRaw', 'Brain\_MRI\_SagittalReference', 'Brain\_MRI\_CoronalRaw',

'Brain\_MRI\_CoronalReference', 'Brain\_MRI\_TransverseRaw', 'Brain\_MRI\_TransverseReference',

'Eyes\_Fundus\_Raw', 'Eyes\_OCT\_Raw', 'Arterial\_Carotids\_Mixed', 'Arterial\_Carotids\_LongAxis',

'Arterial\_Carotids\_CIMT120', 'Arterial\_Carotids\_CIMT150', 'Arterial\_Carotids\_ShortAxis',

'Heart\_MRI\_2chambersRaw', 'Heart\_MRI\_2chambersContrast', 'Heart\_MRI\_3chambersRaw',

'Heart\_MRI\_3chambersContrast', 'Heart\_MRI\_4chambersRaw', 'Heart\_MRI\_4chambersContrast',

'Abdomen\_Liver\_Raw', 'Abdomen\_Liver\_Contrast', 'Abdomen\_Pancreas\_Raw', 'Abdomen\_Pancreas\_Contrast',

'Musculoskeletal\_Spine\_Sagittal', 'Musculoskeletal\_Spine\_Coronal', 'Musculoskeletal\_Hips\_MRI',

'Musculoskeletal\_Knees\_MRI', 'Musculoskeletal\_FullBody\_Mixed', 'Musculoskeletal\_FullBody\_Figure',

'Musculoskeletal\_FullBody\_Skeleton', 'Musculoskeletal\_FullBody\_Flesh',

'PhysicalActivity\_FullWeek\_GramianAngularField1minDifference',

'PhysicalActivity\_FullWeek\_GramianAngularField1minSummation',

'PhysicalActivity\_FullWeek\_MarkovTransitionField1min', 'PhysicalActivity\_FullWeek\_RecurrencePlots1min']

def \_select\_best\_model(self):

# Pick the best model based on the performances

path\_perf = self.path\_data + 'PERFORMANCES\_withoutEnsembles\_ranked\_instances\_' + self.target + '\_test.csv'

Performances = pd.read\_csv(path\_perf).set\_index('version', drop=False)

Performances = Performances[(Performances['organ'] == self.organ)

& (Performances['view'] == self.view)

& (Performances['transformation'] == self.transformation)]

self.version = Performances['version'].values[0]

del Performances

# other parameters

self.parameters = self.\_version\_to\_parameters(self.version)

if self.organ + '\_' + self.view + '\_' + self.transformation in self.organs\_views\_transformations\_images:

DeepLearning.\_\_init\_\_(self, self.parameters['target'], self.parameters['organ'], self.parameters['view'],

self.parameters['transformation'], self.parameters['architecture'],

self.parameters['n\_fc\_layers'], self.parameters['n\_fc\_nodes'],

self.parameters['optimizer'], self.parameters['learning\_rate'],

self.parameters['weight\_decay'], self.parameters['dropout\_rate'],

self.parameters['data\_augmentation\_factor'], False)

def \_format\_residuals(self):

# Format the residuals

Residuals\_full = pd.read\_csv(self.path\_data + 'RESIDUALS\_instances\_' + self.target + '\_test.csv')

Residuals = Residuals\_full[['id', 'outer\_fold'] + self.demographic\_vars + ['res\_' + self.version]]

del Residuals\_full

Residuals.dropna(inplace=True)

Residuals.rename(columns={'res\_' + self.version: 'res'}, inplace=True)

Residuals.set\_index('id', drop=False, inplace=True)

Residuals['outer\_fold'] = Residuals['outer\_fold'].astype(int).astype(str)

Residuals['res\_abs'] = Residuals['res'].abs()

self.Residuals = Residuals

def \_select\_representative\_samples(self):

# Select with samples to plot

print('Selecting representative samples...')

df\_to\_plot = None

# Sex

dict\_sexes\_to\_values = {'Male': 1, 'Female': 0}

for sex in ['Male', 'Female']:

print('Sex: ' + sex)

Residuals\_sex = self.Residuals[self.Residuals['Sex'] == dict\_sexes\_to\_values[sex]]

Residuals\_sex['sex'] = sex

# Age category

for age\_category in ['young', 'middle', 'old']:

print('Age category: ' + age\_category)

if age\_category == 'young':

Residuals\_age = Residuals\_sex[Residuals\_sex['Age'] <= Residuals\_sex['Age'].min() + 10]

elif age\_category == 'middle':

Residuals\_age = Residuals\_sex[(Residuals\_sex['Age'] - Residuals\_sex['Age'].median()).abs() < 5]

else:

Residuals\_age = Residuals\_sex[Residuals\_sex['Age'] >= Residuals\_sex['Age'].max() - 10]

Residuals\_age['age\_category'] = age\_category

# Aging rate

for aging\_rate in ['accelerated', 'normal', 'decelerated']:

print('Aging rate: ' + aging\_rate)

Residuals\_ar = Residuals\_age

if aging\_rate == 'accelerated':

Residuals\_ar.sort\_values(by='res', ascending=True, inplace=True)

elif aging\_rate == 'decelerated':

Residuals\_ar.sort\_values(by='res', ascending=False, inplace=True)

else:

Residuals\_ar.sort\_values(by='res\_abs', ascending=True, inplace=True)

Residuals\_ar['aging\_rate'] = aging\_rate

Residuals\_ar = Residuals\_ar.iloc[:self.N\_samples\_attentionmaps, ]

Residuals\_ar['sample'] = range(len(Residuals\_ar.index))

if df\_to\_plot is None:

df\_to\_plot = Residuals\_ar

else:

df\_to\_plot = df\_to\_plot.append(Residuals\_ar)

# Postprocessing

df\_to\_plot['Biological\_Age'] = df\_to\_plot['Age'] - df\_to\_plot['res']

activations\_path = '../figures/Attention\_Maps/' + self.target + '/' + self.organ + '/' + self.view + '/' + \

self.transformation + '/' + df\_to\_plot['sex'] + '/' + df\_to\_plot['age\_category'] + '/' + \

df\_to\_plot['aging\_rate']

file\_names = '/imagetypeplaceholder\_' + self.target + '\_' + self.organ + '\_' + self.view + '\_' + \

self.transformation + '\_' + df\_to\_plot['sex'] + '\_' + df\_to\_plot['age\_category'] + '\_' + \

df\_to\_plot['aging\_rate'] + '\_' + df\_to\_plot['sample'].astype(str)

if self.leftright:

activations\_path += '/sideplaceholder'

file\_names += '\_sideplaceholder'

df\_to\_plot['save\_title'] = activations\_path + file\_names

path\_save = self.path\_data + 'AttentionMaps-samples\_' + self.target + '\_' + self.organ + '\_' + self.view + \

'\_' + self.transformation + '.csv'

df\_to\_plot.to\_csv(path\_save, index=False)

self.df\_to\_plot = df\_to\_plot

def preprocessing(self):

self.\_select\_best\_model()

self.\_format\_residuals()

self.\_select\_representative\_samples()

def \_preprocess\_for\_outer\_fold(self, outer\_fold):

self.df\_outer\_fold = self.df\_to\_plot[self.df\_to\_plot['outer\_fold'] == outer\_fold]

self.n\_images = len(self.df\_outer\_fold.index)

if self.leftright:

self.n\_images \*= 2

# Generate the data generator(s)

self.n\_images\_batch = self.n\_images // self.batch\_size \* self.batch\_size

self.n\_samples\_batch = self.n\_images\_batch // 2 if self.leftright else self.n\_images\_batch

self.df\_batch = self.df\_outer\_fold.iloc[:self.n\_samples\_batch, :]

if self.n\_images\_batch > 0:

self.generator\_batch = \

MyImageDataGenerator(target=self.target, organ=self.organ, view=self.view,

data\_features=self.df\_batch, n\_samples\_per\_subepoch=None,

batch\_size=self.batch\_size, training\_mode=False,

side\_predictors=self.side\_predictors, dir\_images=self.dir\_images,

images\_width=self.image\_width, images\_height=self.image\_height,

data\_augmentation=False, data\_augmentation\_factor=None, seed=self.seed)

else:

self.generator\_batch = None

self.n\_samples\_leftovers = self.n\_images % self.batch\_size

self.df\_leftovers = self.df\_outer\_fold.iloc[self.n\_samples\_batch:, :]

if self.n\_samples\_leftovers > 0:

self.generator\_leftovers = \

MyImageDataGenerator(target=self.target, organ=self.organ, view=self.view,

data\_features=self.df\_leftovers, n\_samples\_per\_subepoch=None,

batch\_size=self.n\_samples\_leftovers, training\_mode=False,

side\_predictors=self.side\_predictors, dir\_images=self.dir\_images,

images\_width=self.image\_width, images\_height=self.image\_height,

data\_augmentation=False, data\_augmentation\_factor=None, seed=self.seed)

else:

self.generator\_leftovers = None

# load the weights for the fold (for test images in fold i, load the corresponding model: (i-1)%N\_CV\_folds

outer\_fold\_model = str((int(outer\_fold) - 1) % self.n\_CV\_outer\_folds)

self.model.load\_weights(self.path\_data + 'model-weights\_' + self.version + '\_' + outer\_fold\_model + '.h5')

@staticmethod

def \_process\_saliency(saliency):

saliency \*= 255 / np.max(np.abs(saliency))

saliency = saliency.astype(int)

r\_ch = saliency.copy()

r\_ch[r\_ch < 0] = 0

b\_ch = -saliency.copy()

b\_ch[b\_ch < 0] = 0

g\_ch = saliency.copy() \* 0

a\_ch = np.maximum(b\_ch, r\_ch)

saliency = np.dstack((r\_ch, g\_ch, b\_ch, a\_ch))

return saliency

@staticmethod

def \_process\_gradcam(gradcam):

# rescale to 0-255

gradcam = np.maximum(gradcam, 0) / np.max(gradcam)

gradcam = np.uint8(255 \* gradcam)

# Convert to rgb

jet = cm.get\_cmap("jet")

jet\_colors = jet(np.arange(256))[:, :3]

jet\_gradcam = jet\_colors[gradcam]

jet\_gradcam = array\_to\_img(jet\_gradcam)

jet\_gradcam = jet\_gradcam.resize((gradcam.shape[1], gradcam.shape[0]))

jet\_gradcam = img\_to\_array(jet\_gradcam)

return jet\_gradcam

def \_generate\_maps\_for\_one\_batch(self, df, Xs, y):

# Generate saliency

saliencies = get\_gradients\_of\_activations(self.model, Xs, y, layer\_name='input\_1')['input\_1'].sum(axis=3)

# Generate gradam

weights = get\_gradients\_of\_activations(self.model, Xs, y, layer\_name=self.last\_conv\_layer,

)[self.last\_conv\_layer]

weights = weights.mean(axis=(1, 2))

weights /= np.abs(weights.max()) + 1e-7 # for numerical stability

activations = get\_activations(self.model, Xs, layer\_name=self.last\_conv\_layer)[self.last\_conv\_layer]

# We must take the absolute value because for Grad-RAM, unlike for Grad-Cam, we care both about + and - effects

gradcams = np.abs(np.einsum('il,ijkl->ijk', weights, activations))

zoom\_factor = [1] + list(np.array(Xs[0].shape[1:3]) / np.array(gradcams.shape[1:]))

gradcams = zoom(gradcams, zoom\_factor)

# Save single images and filters

for j in range(len(y)):

# select sample

if self.leftright:

idx = j // 2

side = 'right' if j % 2 == 0 else 'left'

else:

idx = j

side = None

path = df['save\_title'].values[idx]

ID = df['id'].values[idx]

# create directory tree if necessary

if self.leftright:

path = path.replace('sideplaceholder', side)

path\_dir = '/'.join(path.split('/')[:-1])

if not os.path.exists(path\_dir):

os.makedirs(path\_dir)

# Save raw image

# Compute path to test if images existed in first place

path\_image = '../images/' + self.organ + '/' + self.view + '/' + self.transformation + '/'

if self.leftright:

path\_image += side + '/'

path\_image += ID + '.jpg'

if not os.path.exists(path\_image):

print('No image found at ' + path\_image + ', skipping.')

continue

img = load\_img(path\_image, target\_size=(saliencies.shape[1], saliencies.shape[2]))

img.save(path.replace('imagetypeplaceholder', 'RawImage') + '.jpg')

# Save saliency

saliency = saliencies[j, :, :]

saliency = self.\_process\_saliency(saliency)

np.save(path.replace('imagetypeplaceholder', 'Saliency') + '.npy', saliency)

# Save gradcam

gradcam = gradcams[j, :, :]

gradcam = self.\_process\_gradcam(gradcam)

np.save(path.replace('imagetypeplaceholder', 'Gradcam') + '.npy', gradcam)

def generate\_filters(self):

if self.organ + '\_' + self.view + '\_' + self.transformation in self.organs\_views\_transformations\_images:

self.\_generate\_architecture()

self.model.compile(optimizer=self.optimizers[self.optimizer](lr=self.learning\_rate, clipnorm=1.0),

loss=self.loss\_function, metrics=self.metrics)

self.last\_conv\_layer = self.dict\_architecture\_to\_last\_conv\_layer\_name[self.parameters['architecture']]

for outer\_fold in self.outer\_folds:

print('Generate attention maps for outer\_fold ' + outer\_fold)

gc.collect()

self.\_preprocess\_for\_outer\_fold(outer\_fold)

n\_samples\_per\_batch = self.batch\_size // 2 if self.leftright else self.batch\_size

for i in range(self.n\_images // self.batch\_size):

print('Generating maps for batch ' + str(i))

Xs, y = self.generator\_batch.\_\_getitem\_\_(i)

df = self.df\_batch.iloc[n\_samples\_per\_batch \* i: n\_samples\_per\_batch \* (i + 1), :]

self.\_generate\_maps\_for\_one\_batch(df, Xs, y)

if self.n\_samples\_leftovers > 0:

print('Generating maps for leftovers')

Xs, y = self.generator\_leftovers.\_\_getitem\_\_(0)

self.\_generate\_maps\_for\_one\_batch(self.df\_leftovers, Xs, y)

class GWASPreprocessing(Basics):

"""

Preprocesses the data for the GWASs.

"""

def \_\_init\_\_(self, target=None):

Basics.\_\_init\_\_(self)

self.target = target

self.fam = None

self.Residuals = None

self.covars = None

self.data = None

self.list\_organs = None

self.IIDs\_organs = {}

self.IIDs\_organ\_pairs = {}

def \_generate\_fam\_file(self):

fam = pd.read\_csv('/n/groups/patel/uk\_biobank/project\_52887\_genetics/ukb52887\_cal\_chr1\_v2\_s488264.fam',

header=None, sep=' ')

fam.columns = ['FID', 'IID', 'father', 'mother', 'Sex', 'phenotype']

fam['phenotype'] = 1

fam.to\_csv(self.path\_data + 'GWAS.fam', index=False, header=False, sep=' ')

fam.to\_csv(self.path\_data + 'GWAS\_exhaustive\_placeholder.tab', index=False, sep='\t')

self.fam = fam

def \_preprocess\_residuals(self):

# Load residuals

Residuals = pd.read\_csv(self.path\_data + 'RESIDUALS\_bestmodels\_eids\_' + self.target + '\_test.csv')

Residuals['id'] = Residuals['eid']

Residuals.rename(columns={'id': 'FID', 'eid': 'IID'}, inplace=True)

Residuals = Residuals[Residuals['Ethnicity.White'] == 1]

cols\_to\_drop = ['instance', 'outer\_fold', 'Sex'] + \

[col for col in Residuals.columns.values if 'Ethnicity.' in col]

Residuals.drop(columns=cols\_to\_drop, inplace=True)

self.Residuals = Residuals

self.list\_organs = [col for col in self.Residuals.columns.values if col not in ['FID', 'IID', 'Age']]

def \_preprocess\_covars(self):

# Load covars

covar\_cols = ['eid', '22001-0.0', '21000-0.0', '54-0.0', '22000-0.0'] + ['22009-0.' + str(i) for i in

range(1, 41)]

covars = pd.read\_csv('/n/groups/patel/uk\_biobank/project\_52887\_41230/ukb41230.csv', usecols=covar\_cols)

dict\_rename = {'eid': 'IID', '22001-0.0': 'Sex', '21000-0.0': 'Ethnicity', '54-0.0': 'Assessment\_center',

'22000-0.0': 'Genotyping\_batch'}

for i in range(1, 41):

dict\_rename.update(dict.fromkeys(['22009-0.' + str(i)], 'PC' + str(i)))

covars.rename(columns=dict\_rename, inplace=True)

covars.dropna(inplace=True)

covars['Sex'][covars['Sex'] == 0] = 2

covars['Sex'] = covars['Sex'].astype(int)

# remove non whites samples as suggested in BOLT-LMM\_v2.3.4\_manual.pdf p18

covars = covars[covars['Ethnicity'].isin([1, 1001, 1002, 1003])]

self.covars = covars

def \_merge\_main\_data(self):

# Merge both dataframes

self.data = self.covars.merge(self.Residuals, on=['IID'])

reordered\_cols = ['FID', 'IID', 'Assessment\_center', 'Genotyping\_batch', 'Age', 'Sex', 'Ethnicity'] + \

['PC' + str(i) for i in range(1, 41)] + self.list\_organs

self.data = self.data[reordered\_cols]

print('Preparing data for heritabilities')

for organ in self.list\_organs:

print('Preparing data for ' + organ)

data\_organ = self.data.copy()

cols\_to\_drop = [organ2 for organ2 in self.list\_organs if organ2 != organ]

data\_organ.drop(columns=cols\_to\_drop, inplace=True)

data\_organ.dropna(inplace=True)

data\_organ.to\_csv(self.path\_data + 'GWAS\_data\_' + self.target + '\_' + organ + '.tab', index=False,

sep='\t')

self.IIDs\_organs[organ] = data\_organ['IID'].values

def \_preprocessing\_genetic\_correlations(self):

print('Preparing data for genetic correlations')

organs\_pairs = pd.DataFrame(columns=['organ1', 'organ2'])

for counter, organ1 in enumerate(self.list\_organs):

for organ2 in self.list\_organs[(counter + 1):]:

print('Preparing data for the organ pair ' + organ1 + ' and ' + organ2)

# Generate GWAS dataframe

organs\_pairs = organs\_pairs.append({'organ1': organ1, 'organ2': organ2}, ignore\_index=True)

data\_organ\_pair = self.data.copy()

cols\_to\_drop = [organ3 for organ3 in self.list\_organs if organ3 not in [organ1, organ2]]

data\_organ\_pair.drop(columns=cols\_to\_drop, inplace=True)

data\_organ\_pair.dropna(inplace=True)

data\_organ\_pair.to\_csv(self.path\_data + 'GWAS\_data\_' + self.target + '\_' + organ1 + '\_' + organ2 +

'.tab', index=False, sep='\t')

self.IIDs\_organ\_pairs[organ1 + '\_' + organ2] = data\_organ\_pair['IID'].values

organs\_pairs.to\_csv(self.path\_data + 'GWAS\_genetic\_correlations\_pairs\_' + self.target + '.csv', header=False,

index=False)

def \_list\_removed(self):

# samples to remove for each organ

print('Listing samples to remove for each organ')

for organ in self.list\_organs:

print('Preparing samples to remove for organ ' + organ)

remove\_organ = self.fam[['FID', 'IID']].copy()

remove\_organ = remove\_organ[-remove\_organ['IID'].isin(self.IIDs\_organs[organ])]

remove\_organ.to\_csv(self.path\_data + 'GWAS\_remove\_' + self.target + '\_' + organ + '.tab', index=False,

header=False, sep=' ')

# samples to remove for each organ pair

print('Listing samples to remove for each organ pair')

for counter, organ1 in enumerate(self.list\_organs):

for organ2 in self.list\_organs[(counter + 1):]:

print('Preparing samples to remove for organ pair ' + organ1 + ' and ' + organ2)

remove\_organ\_pair = self.fam[['FID', 'IID']].copy()

remove\_organ\_pair = \

remove\_organ\_pair[-remove\_organ\_pair['IID'].isin(self.IIDs\_organ\_pairs[organ1 + '\_' + organ2])]

remove\_organ\_pair.to\_csv(self.path\_data + 'GWAS\_remove\_' + self.target + '\_' + organ1 + '\_' + organ2 +

'.tab', index=False, header=False, sep=' ')

def compute\_gwas\_inputs(self):

self.\_generate\_fam\_file()

self.\_preprocess\_residuals()

self.\_preprocess\_covars()

self.\_merge\_main\_data()

self.\_preprocessing\_genetic\_correlations()

self.\_list\_removed()

class GWASPostprocessing(Basics):

"""

Postprocesses the GWAS results and stores the results in summary files.

"""

def \_\_init\_\_(self, target=None):

Basics.\_\_init\_\_(self)

self.target = target

self.organ = None

self.GWAS = None

self.FDR\_correction = 5e-8

def \_processing(self):

self.GWAS = pd.read\_csv(self.path\_data + 'GWAS\_' + self.target + '\_' + self.organ + '\_X.stats', sep='\t')

GWAS\_autosome = pd.read\_csv(self.path\_data + 'GWAS\_' + self.target + '\_' + self.organ + '\_autosome.stats',

sep='\t')

self.GWAS[self.GWAS['CHR'] != 23] = GWAS\_autosome

self.GWAS\_hits = self.GWAS[self.GWAS['P\_BOLT\_LMM\_INF'] < self.FDR\_correction]

def \_save\_data(self):

self.GWAS.to\_csv(self.path\_data + 'GWAS\_' + self.target + '\_' + self.organ + '.csv', index=False)

self.GWAS\_hits.to\_csv(self.path\_data + 'GWAS\_hits\_' + self.target + '\_' + self.organ + '.csv', index=False)

def \_merge\_all\_hits(self):

print('Merging all the GWAS results into a model called All...')

# Summarize all the significant SNPs

files = [file for file in glob.glob(self.path\_data + 'GWAS\_hits\*')

if ('All' not in file) & ('\_withGenes' not in file)]

All\_hits = None

print(files)

for file in files:

print(file)

hits\_organ = pd.read\_csv(file)[

['SNP', 'CHR', 'BP', 'GENPOS', 'ALLELE1', 'ALLELE0', 'A1FREQ', 'F\_MISS', 'CHISQ\_LINREG',

'P\_LINREG', 'BETA', 'SE', 'CHISQ\_BOLT\_LMM\_INF', 'P\_BOLT\_LMM\_INF']]

hits\_organ['organ'] = '.'.join(file.split('\_')[-1].split('.')[:-1])

if All\_hits is None:

All\_hits = hits\_organ

else:

All\_hits = pd.concat([All\_hits, hits\_organ])

All\_hits.sort\_values(by=['CHR', 'BP'], inplace=True)

All\_hits.to\_csv(self.path\_data + 'GWAS\_hits\_' + self.target + '\_All.csv', index=False)

def processing\_all\_organs(self):

if not os.path.exists('../figures/GWAS/'):

os.makedirs('../figures/GWAS/')

for organ in self.organs\_XWAS:

if os.path.exists(self.path\_data + 'GWAS\_' + self.target + '\_' + organ + '\_X.stats') & \

os.path.exists(self.path\_data + 'GWAS\_' + self.target + '\_' + organ + '\_autosome.stats'):

print('Processing data for organ ' + organ)

self.organ = organ

self.\_processing()

self.\_save\_data()

self.\_merge\_all\_hits()

@staticmethod

def \_grep(pattern, path):

for line in open(path, 'r'):

if line.find(pattern) > -1:

return True

return False

@staticmethod

def \_melt\_correlation\_matrix(Correlations, models):

Corrs = Correlations.loc[models, models]

Corrs = Corrs.where(np.triu(np.ones(Corrs.shape), 1).astype(np.bool))

Corrs = Corrs.stack().reset\_index()

Corrs.columns = ['Row', 'Column', 'Correlation']

return Corrs

@staticmethod

def \_compute\_stats(data, title):

m = data['Correlation'].mean()

s = data['Correlation'].std()

n = len(data.index)

print('Correlation between ' + title + ': ' + str(round(m, 3)) + '+-' + str(round(s, 3)) +

', n\_pairs=' + str(n))

def parse\_heritability\_scores(self):

# Generate empty dataframe

Heritabilities = np.empty((len(self.organs\_XWAS), 3,))

Heritabilities.fill(np.nan)

Heritabilities = pd.DataFrame(Heritabilities)

Heritabilities.index = self.organs\_XWAS

Heritabilities.columns = ['Organ', 'h2', 'h2\_sd']

# Fill the dataframe

for organ in self.organs\_XWAS:

path = '../eo/MI09C\_reml\_' + self.target + '\_' + organ + '\_X.out'

if os.path.exists(path) and self.\_grep("h2g", path):

for line in open('../eo/MI09C\_reml\_' + self.target + '\_' + organ + '\_X.out', 'r'):

if line.find('h2g (1,1): ') > -1:

h2 = float(line.split()[2])

h2\_sd = float(line.split()[-1][1:-2])

Heritabilities.loc[organ, :] = [organ, h2, h2\_sd]

# Print and save results

print('Heritabilities:')

print(Heritabilities)

Heritabilities.to\_csv(self.path\_data + 'GWAS\_heritabilities\_' + self.target + '.csv', index=False)

def parse\_genetic\_correlations(self):

# Generate empty dataframe

Genetic\_correlations = np.empty((len(self.organs\_XWAS), len(self.organs\_XWAS),))

Genetic\_correlations.fill(np.nan)

Genetic\_correlations = pd.DataFrame(Genetic\_correlations)

Genetic\_correlations.index = self.organs\_XWAS

Genetic\_correlations.columns = self.organs\_XWAS

Genetic\_correlations\_sd = Genetic\_correlations.copy()

Genetic\_correlations\_str = Genetic\_correlations.copy()

# Fill the dataframe

for counter, organ1 in enumerate(self.organs\_XWAS):

for organ2 in self.organs\_XWAS[(counter + 1):]:

if os.path.exists('../eo/MI09D\_' + self.target + '\_' + organ1 + '\_' + organ2 + '.out'):

for line in open('../eo/MI09D\_' + self.target + '\_' + organ1 + '\_' + organ2 + '.out', 'r'):

if line.find('gen corr (1,2):') > -1:

corr = float(line.split()[3])

corr\_sd = float(line.split()[-1][1:-2])

corr\_str = "{:.3f}".format(corr) + '+-' + "{:.3f}".format(corr\_sd)

Genetic\_correlations.loc[organ1, organ2] = corr

Genetic\_correlations.loc[organ2, organ1] = corr

Genetic\_correlations\_sd.loc[organ1, organ2] = corr\_sd

Genetic\_correlations\_sd.loc[organ2, organ1] = corr\_sd

Genetic\_correlations\_str.loc[organ1, organ2] = corr\_str

Genetic\_correlations\_str.loc[organ2, organ1] = corr\_str

# Print and save the results

print('Genetic correlations:')

print(Genetic\_correlations)

Genetic\_correlations.to\_csv(self.path\_data + 'GWAS\_correlations\_' + self.target + '.csv')

Genetic\_correlations\_sd.to\_csv(self.path\_data + 'GWAS\_correlations\_sd\_' + self.target + '.csv')

Genetic\_correlations\_str.to\_csv(self.path\_data + 'GWAS\_correlations\_str\_' + self.target + '.csv')

# Save sample size for the GWAS correlations

Correlations\_sample\_sizes = Genetic\_correlations.copy()

Correlations\_sample\_sizes = Correlations\_sample\_sizes \* np.NaN

dimensions = Correlations\_sample\_sizes.columns.values

for i1, dim1 in enumerate(dimensions):

for i2, dim2 in enumerate(dimensions[i1:]):

# Find the sample size

path = '../data/GWAS\_data\_Age\_' + dim1 + '\_' + dim2 + '.tab'

if os.path.exists(path):

ss = len(pd.read\_csv(path, sep='\t').index)

Correlations\_sample\_sizes.loc[dim1, dim2] = ss

Correlations\_sample\_sizes.loc[dim2, dim1] = ss

Correlations\_sample\_sizes.to\_csv(self.path\_data + 'GWAS\_correlations\_sample\_sizes\_' + self.target + '.csv')

# Print correlations between main dimensions

main\_dims = ['Abdomen', 'Musculoskeletal', 'Lungs', 'Eyes', 'Heart', 'Arterial', 'Brain', 'Biochemistry',

'Hearing', 'ImmuneSystem', 'PhysicalActivity']

Corrs\_main = self.\_melt\_correlation\_matrix(Correlations, main\_dims)

Corrs\_main\_sd = self.\_melt\_correlation\_matrix(Correlations\_sd, main\_dims)

Corrs\_main['Correlation\_sd'] = Corrs\_main\_sd['Correlation']

Corrs\_main['Correlation\_str'] = Corrs\_main['Correlation'] + '+-' + Corrs\_main['Correlation\_sd']

# Fill the table with sample sizes

sample\_sizes = []

to\_remove\_ss = []

for i, row in Corrs\_main.iterrows():

# Fill the sample size

sample\_size = Correlations\_sample\_sizes.loc[row['Row'], row['Column']]

if sample\_size <= 15000:

to\_remove\_ss.append(i)

sample\_sizes.append(sample\_size)

Corrs\_main['Sample\_size'] = sample\_sizes

self.\_compute\_stats(Corrs\_main, 'all pairs')

self.\_compute\_stats(Corrs\_main.drop(index=to\_remove\_ss), 'after filtering sample sizes <= 15000')

# Print correlations between subdimensions

pairs\_all = \

[['BrainMRI', 'BrainCognitive'], ['EyesOCT', 'EyesFundus'], ['HeartECG', 'HeartMRI'],

['AbdomenLiver', 'AbdomenPancreas'], ['BiochemistryBlood', 'BiochemistryUrine'],

['MusculoskeletalScalars', 'MusculoskeletalFullBody'], ['MusculoskeletalScalars', 'MusculoskeletalSpine'],

['MusculoskeletalScalars', 'MusculoskeletalHips'], ['MusculoskeletalScalars', 'MusculoskeletalKnees'],

['MusculoskeletalFullBody', 'MusculoskeletalSpine'], ['MusculoskeletalFullBody', 'MusculoskeletalHips'],

['MusculoskeletalFullBody', 'MusculoskeletalKnees'], ['MusculoskeletalSpine', 'MusculoskeletalHips'],

['MusculoskeletalSpine', 'MusculoskeletalKnees'], ['MusculoskeletalHips', 'MusculoskeletalKnees']]

pairs\_musculo = \

[['MusculoskeletalScalars', 'MusculoskeletalFullBody'], ['MusculoskeletalScalars', 'MusculoskeletalSpine'],

['MusculoskeletalScalars', 'MusculoskeletalHips'], ['MusculoskeletalScalars', 'MusculoskeletalKnees']]

pairs\_musculo\_images = \

[['MusculoskeletalFullBody', 'MusculoskeletalSpine'], ['MusculoskeletalFullBody', 'MusculoskeletalHips'],

['MusculoskeletalFullBody', 'MusculoskeletalKnees'], ['MusculoskeletalSpine', 'MusculoskeletalHips'],

['MusculoskeletalSpine', 'MusculoskeletalKnees'], ['MusculoskeletalHips', 'MusculoskeletalKnees']]

PAIRS = {'all subdimensions': pairs\_all, 'musculo scalars vs others': pairs\_musculo,

'musculo-no scalars': pairs\_musculo\_images}

for \_, (key, pairs) in enumerate(PAIRS.items()):

print(key)

cors\_pairs = []

for pair in pairs:

cor = Correlations.loc[pair[0], pair[1]]

cor\_sd = Correlations\_sd.loc[pair[0], pair[1]]

ss = Correlations\_sample\_sizes.loc[pair[0], pair[1]]

cors\_pairs.append(cor)

print('Correlation between ' + pair[0] + ' and ' + pair[1] + ' = ' + str(round(cor, 3)) + '+-' +

str(round(cor\_sd, 3)) + '; sample size = ' + str(ss))

print('Mean correlation for ' + key + ' = ' + str(round(np.mean(cors\_pairs), 3)) + '+-' +

str(round(np.std(cors\_pairs), 3)) + ', number of pairs = ' + str(len(pairs)))

@staticmethod

def compare\_phenotypic\_correlation\_with\_genetic\_correlations():

Phenotypic\_correlations = pd.read\_csv('../data/ResidualsCorrelations\_bestmodels\_eids\_Age\_test.csv', index\_col=0)

Phenotypic\_correlations\_sd = pd.read\_csv('../data/ResidualsCorrelations\_bestmodels\_sd\_eids\_Age\_test.csv',

index\_col=0)

Genetic\_correlations = pd.read\_csv('../data/GWAS\_correlations\_Age.csv', index\_col=0)

Genetic\_correlations\_sd = pd.read\_csv('../data/GWAS\_correlations\_sd\_Age.csv', index\_col=0)

Correlations\_sample\_sizes = pd.read\_csv('../data/GWAS\_correlations\_sample\_sizes\_Age.csv', index\_col=0)

Genetic\_correlations\_filtered = Genetic\_correlations.where(Correlations\_sample\_sizes > 15000)

Phenotypic\_correlations\_filtered = Phenotypic\_correlations.where(~Genetic\_correlations\_filtered.isna())

dict\_dims\_layers = {

'all\_dims': Phenotypic\_correlations\_filtered.columns.values,

'main\_dims': ['Brain', 'Eyes', 'Hearing', 'Lungs', 'Arterial', 'Heart', 'Abdomen', 'Musculoskeletal',

'PhysicalActivity', 'Biochemistry', 'ImmuneSystem'],

'sub\_dims': ['BrainCognitive', 'BrainMRI', 'EyesFundus', 'EyesOCT', 'ArterialPulseWaveAnalysis',

'ArterialCarotids',

'HeartECG', 'HeartMRI', 'AbdomenLiver', 'AbdomenPancreas', 'MusculoskeletalSpine',

'MusculoskeletalHips', 'MusculoskeletalKnees', 'MusculoskeletalFullBody',

'MusculoskeletalScalars',

'BiochemistryUrine', 'BiochemistryBlood']

}

def \_print\_comparisons\_between\_pheno\_and\_geno(dimensions):

Pheno\_dims = Phenotypic\_correlations\_filtered.loc[dimensions, dimensions]

Geno\_dims = Genetic\_correlations\_filtered.loc[dimensions, dimensions]

Pheno\_dims = Pheno\_dims.where(np.triu(np.ones(Pheno\_dims.shape), 1).astype(np.bool))

Pheno\_dims = Pheno\_dims.stack().reset\_index()

Geno\_dims = Geno\_dims.where(np.triu(np.ones(Geno\_dims.shape), 1).astype(np.bool))

Geno\_dims = Geno\_dims.stack().reset\_index()

Pheno\_dims.columns = ['Row', 'Column', 'Correlation']

Geno\_dims.columns = ['Row', 'Column', 'Correlation']

Correlations\_dims = Pheno\_dims.copy()

Correlations\_dims = Correlations\_dims.rename(columns={'Correlation': 'Phenotypic'})

Correlations\_dims['Genetic'] = Geno\_dims['Correlation']

final\_cor = str(round(Correlations\_dims[['Phenotypic', 'Genetic']].corr().iloc[0, 1], 3))

# Find min and max difference:

Correlations\_dims['Difference'] = Correlations\_dims['Phenotypic'] - Correlations\_dims['Genetic']

# min

min\_diff = Correlations\_dims.iloc[Correlations\_dims['Difference'].idxmin(), :]

min\_pheno\_sd = Phenotypic\_correlations\_sd.loc[min\_diff['Row'], min\_diff['Column']]

min\_genetic\_sd = Genetic\_correlations\_sd.loc[min\_diff['Row'], min\_diff['Column']]

min\_diff\_sd = np.sqrt(min\_pheno\_sd \*\* 2 + min\_genetic\_sd \*\* 2)

# max

max\_diff = Correlations\_dims.iloc[Correlations\_dims['Difference'].idxmax(), :]

max\_pheno\_sd = Phenotypic\_correlations\_sd.loc[max\_diff['Row'], max\_diff['Column']]

max\_genetic\_sd = Genetic\_correlations\_sd.loc[max\_diff['Row'], max\_diff['Column']]

max\_diff\_sd = np.sqrt(max\_pheno\_sd \*\* 2 + max\_genetic\_sd \*\* 2)

# print key results

print('The correlation between phenotypic and genetic correlations is: ' + final\_cor)

print('The min difference between phenotypic and genetic correlations is between ' + min\_diff[

'Row'] + ' and ' +

min\_diff['Column'] + '. Difference = ' + str(round(min\_diff['Difference'], 3)) + '+-' +

str(round(min\_diff\_sd, 3)) + '; Phenotypic correlation = ' + str(

round(min\_diff['Phenotypic'], 3)) + '+-' +

str(round(min\_pheno\_sd, 3)) + '; Genetic correlation = ' + str(round(min\_diff['Genetic'], 3)) + '+-' +

str(round(min\_genetic\_sd, 3)))

print('The max difference between phenotypic and genetic correlations is between ' + max\_diff[

'Row'] + ' and ' +

max\_diff['Column'] + '. Difference = ' + str(round(max\_diff['Difference'], 3)) + '+-' +

str(round(max\_diff\_sd, 3)) + '; Phenotypic correlation = ' + str(

round(max\_diff['Phenotypic'], 3)) + '+-' +

str(round(max\_pheno\_sd, 3)) + '; Genetic correlation = ' + str(round(max\_diff['Genetic'], 3)) + '+-' +

str(round(max\_genetic\_sd, 3)))

for \_, (dims\_name, dims) in enumerate(dict\_dims\_layers.items()):

print('Printing the comparison between the phenotypic and genetic correlations at the following level: ' +

dims\_name)

\_print\_comparisons\_between\_pheno\_and\_geno(dims)

class GWASAnnotate(Basics):

"""

/!\ This class corresponds to a step in the pipeline that should be performed on local machine, since it must be

complemented with researches on the internet for different steps. /!\

Annotates the hits from the GWAS: names of the genes and gene types.

"""

def \_\_init\_\_(self, target=None):

Basics.\_\_init\_\_(self)

self.target = target

self.All\_hits = None

self.All\_hits\_missing = None

def download\_data(self):

os.chdir('/Users/Alan/Desktop/Aging/Medical\_Images/bash\_local/')

os.system('scp al311@transfer.rc.hms.harvard.edu:/n/groups/patel/Alan/Aging/Medical\_Images/data/' +

self.path\_data + 'GWAS\_hits\_' + self.target + '\_All.csv' + ' ../data/')

self.All\_hits = pd.read\_csv(self.path\_data + 'GWAS\_hits\_' + self.target + '\_All.csv')

@staticmethod

def \_find\_nearest\_gene(row, key):

if row['Overlapped Gene'] != 'None':

gene = row['Overlapped Gene']

gene\_type = row['Type']

elif row['Distance to Nearest Downstream Gene'] <= row['Distance to Nearest Upstream Gene']:

gene = row['Nearest Downstream Gene']

gene\_type = row['Type of Nearest Downstream Gene']

else:

gene = row['Nearest Upstream Gene']

gene\_type = row['Type of Nearest Upstream Gene']

to\_return = pd.Series([row[key], gene, gene\_type])

to\_return.index = [key, 'Gene', 'Gene\_type']

return to\_return

@staticmethod

def \_concatenate\_genes(group, key):

row = group.drop\_duplicates(subset=[key])

unique\_genes\_rows = group.drop\_duplicates(subset=[key, 'Gene'])

row['Gene'] = ', '.join(list(unique\_genes\_rows['Gene']))

row['Gene\_type'] = ', '.join(list(unique\_genes\_rows['Gene\_type']))

return row

def preprocessing\_rs(self):

# Generate the list of SNPs to annotate in two formats to input into https://www.snp-nexus.org/v4/

# Format 1: based on rs#

snps\_rs = pd.Series(self.All\_hits['SNP'].unique())

snps\_rs.index = ['dbsnp'] \* len(snps\_rs.index)

snps\_rs.to\_csv(self.path\_data + 'snps\_rs.txt', sep='\t', header=False)

def postprocessing\_rs(self):

# Load the output fromsnp-nexus and fill the available rows

genes\_rs = pd.read\_csv(self.path\_data + 'GWAS\_genes\_rs.txt', sep='\t')

# Find the nearest gene

genes\_rs = genes\_rs.apply(self.\_find\_nearest\_gene, args=(['Variation ID']), axis=1)

# Concatenate the findinds when several genes matched

genes\_rs = genes\_rs.groupby(by='Variation ID').apply(self.\_concatenate\_genes, 'Variation ID')

# Fill the rows from the main dataframe when possible

genes\_rs.set\_index('Variation ID', inplace=True)

self.All\_hits['Gene'] = np.NaN

self.All\_hits['Gene\_type'] = np.NaN

self.All\_hits.set\_index('SNP', drop=False, inplace=True)

self.All\_hits.loc[genes\_rs.index, ['Gene', 'Gene\_type']] = genes\_rs

def preprocessing\_chrbp(self):

# Format 2: based on CHR and BP

snps\_chrbp = self.All\_hits.loc[self.All\_hits['Gene'].isna(), ['CHR', 'BP', 'ALLELE0', 'ALLELE1']]

snps\_chrbp['strand'] = 1

snps\_chrbp.index = ['chromosome'] \* len(snps\_chrbp.index)

snps\_chrbp.drop\_duplicates(inplace=True)

snps\_chrbp.to\_csv(self.path\_data + 'snps\_chrbp.txt', sep='\t', header=False)

def postprocessing\_chrbp(self):

# Load the output from snp-nexus and fill the available rows

genes\_chrbp = pd.read\_csv(self.path\_data + 'GWAS\_genes\_chrbp.txt', sep='\t')

genes\_chrbp['chrbp'] = genes\_chrbp['Chromosome'].astype(str) + ':' + genes\_chrbp['Position'].astype(str)

# Find the nearest gene

genes\_chrbp = genes\_chrbp.apply(self.\_find\_nearest\_gene, args=(['chrbp']), axis=1)

# Concatenate the findinds when several genes matched

genes\_chrbp = genes\_chrbp.groupby(by='chrbp').apply(self.\_concatenate\_genes, 'chrbp')

# Fill the rows from the main dataframe when possible

genes\_chrbp.set\_index('chrbp', inplace=True)

self.All\_hits['chrbp'] = 'chr' + self.All\_hits['CHR'].astype(str) + ':' + self.All\_hits['BP'].astype(str)

self.All\_hits.set\_index('chrbp', drop=False, inplace=True)

# Only keep subset of genes that actually are hits (somehow extra SNPs are returned too

genes\_chrbp = genes\_chrbp[genes\_chrbp.index.isin(self.All\_hits.index.values)]

self.All\_hits.loc[genes\_chrbp.index, ['Gene', 'Gene\_type']] = genes\_chrbp

def preprocessing\_missing(self):

# Identify which SNPs were not matched so far, and use zoom locus to fill the gaps

self.All\_hits\_missing = self.All\_hits[self.All\_hits['Gene'].isna()]

print(str(len(self.All\_hits\_missing.drop\_duplicates(subset=['SNP']).index)) + ' missing SNPs out of ' +

str(len(self.All\_hits.drop\_duplicates(subset=['SNP']).index)) + '.')

self.All\_hits\_missing.to\_csv(self.path\_data + 'All\_hits\_missing.csv', index=False, sep='\t')

def postprocessing\_missing(self):

# The gene\_type column was filled using https://www.genecards.org/

self.All\_hits.loc['chr1:3691997', ['Gene', 'Gene\_type']] = ['SMIM1', 'protein\_coding']

self.All\_hits.loc['chr2:24194313', ['Gene', 'Gene\_type']] = ['COL4A4', 'protein\_coding']

self.All\_hits.loc['chr2:227896885', ['Gene', 'Gene\_type']] = ['UBXN2A', 'protein\_coding']

self.All\_hits.loc['chr2:27656822', ['Gene', 'Gene\_type']] = ['NRBP1', 'protein\_coding']

self.All\_hits.loc['chr2:42396721', ['Gene', 'Gene\_type']] = ['AC083949.1, EML4', 'rna\_gene, protein\_coding']

self.All\_hits.loc['chr2:71661855', ['Gene', 'Gene\_type']] = ['ZNF638', 'protein\_coding']

self.All\_hits.loc['chr3:141081497', ['Gene', 'Gene\_type']] = ['PXYLP1, AC117383.1, ZBTB38',

'protein\_coding, rna\_gene, protein\_coding']

self.All\_hits.loc['chr4:106317506', ['Gene', 'Gene\_type']] = ['PPA2', 'protein\_coding']

self.All\_hits.loc['chr5:156966773', ['Gene', 'Gene\_type']] = ['ADAM19', 'protein\_coding']

self.All\_hits.loc['chr6:29797695', ['Gene', 'Gene\_type']] = ['HLA-G', 'protein\_coding']

self.All\_hits.loc['chr6:31106501', ['Gene', 'Gene\_type']] = ['PSORS1C1, PSORS1C2',

'protein\_coding, protein\_coding']

self.All\_hits.loc['chr6:31322216', ['Gene', 'Gene\_type']] = ['HLA-B', 'protein\_coding']

self.All\_hits.loc['chr6:32552146', ['Gene', 'Gene\_type']] = ['HLA-DRB1', 'protein\_coding']

self.All\_hits.loc['chr6:33377481', ['Gene', 'Gene\_type']] = ['KIFC1', 'protein\_coding']

self.All\_hits.loc['chr8:9683437', ['Gene', 'Gene\_type']] = ['snoU13', 'small\_nucleolar\_rna\_gene']

self.All\_hits.loc['chr8:19822809', ['Gene', 'Gene\_type']] = ['LPL', 'protein\_coding']

self.All\_hits.loc['chr8:75679126', ['Gene', 'Gene\_type']] = ['MIR2052HG', 'rna\_gene']

self.All\_hits.loc['chr10:18138488', ['Gene', 'Gene\_type']] = ['MRC1', 'protein\_coding']

self.All\_hits.loc['chr10:96084372', ['Gene', 'Gene\_type']] = ['PLCE1', 'protein\_coding']

self.All\_hits.loc['chr11:293001', ['Gene', 'Gene\_type']] = ['PGGHG', 'protein\_coding']

self.All\_hits.loc['chr15:74282833', ['Gene', 'Gene\_type']] = ['STOML1', 'protein\_coding']

self.All\_hits.loc['chr15:89859932', ['Gene', 'Gene\_type']] = ['FANCI, POLG', 'protein\_coding, protein\_coding']

self.All\_hits.loc['chr17:44341869', ['Gene', 'Gene\_type']] = ['AC005829.1', 'pseudo\_gene']

self.All\_hits.loc['chr17:79911164', ['Gene', 'Gene\_type']] = ['NOTUM', 'protein\_coding']

self.All\_hits.loc['chr20:57829821', ['Gene', 'Gene\_type']] = ['ZNF831', 'protein\_coding']

self.All\_hits.loc['chr22:29130347', ['Gene', 'Gene\_type']] = ['CHEK2', 'protein\_coding']

# The following genes were not found by locuszoom, so I used https://www.rcsb.org/pdb/chromosome.do :

self.All\_hits.loc['chr6:31084935', ['Gene', 'Gene\_type']] = ['CDSN', 'protein\_coding']

self.All\_hits.loc['chr6:31105857', ['Gene', 'Gene\_type']] = ['PSORS1C2', 'protein\_coding']

# The following genes was named "0" in locuszoom, so I used https://www.rcsb.org/pdb/chromosome.do :

self.All\_hits.loc['chr23:13771531', ['Gene', 'Gene\_type']] = ['OFD1', 'protein\_coding']

# The following gene did not have a match

self.All\_hits.loc['chr23:56640134', ['Gene', 'Gene\_type']] = ['UNKNOWN', 'UNKNOWN']

# Ensuring that all SNPs have been annotated

print('Ensuring that all SNPs have been annotated:')

assert self.All\_hits['Gene'].isna().sum() == 0

print('Passed.')

# Counter number of unique genes involved, and generating the list

unique\_genes = list(set((', '.join(self.All\_hits['Gene'].unique())).split(', ')))

print('A total of ' + str(len(unique\_genes)) + ' unique genes are associated with accelerated aging.')

np.save(self.path\_data + 'GWAS\_unique\_genes.npy', np.array(unique\_genes))

def postprocessing\_hits(self):

self.All\_hits.drop(columns=['chrbp'], inplace=True)

self.All\_hits.to\_csv(self.path\_data + 'GWAS\_hits\_' + self.target + '\_All\_withGenes.csv', index=False)

for organ in self.organs\_XWAS:

Hits\_organ = self.All\_hits[self.All\_hits['organ'] == organ].drop(columns=['organ'])

Hits\_organ.to\_csv(self.path\_data + 'GWAS\_hits\_' + self.target + '\_' + organ + '\_withGenes.csv', index=False)

def summarize\_results(self):

# Generate empty dataframe

organs = ['\*', '\*instances01', '\*instances1.5x', '\*instances23', 'Abdomen', 'AbdomenLiver', 'AbdomenPancreas',

'Arterial', 'ArterialCarotids', 'ArterialPulseWaveAnalysis', 'Biochemistry', 'BiochemistryBlood',

'BiochemistryUrine', 'Brain', 'BrainCognitive', 'BrainMRI', 'Eyes', 'EyesFundus', 'EyesOCT',

'Hearing', 'Heart', 'HeartECG', 'HeartMRI', 'ImmuneSystem', 'Lungs', 'Musculoskeletal',

'MusculoskeletalFullBody', 'MusculoskeletalHips', 'MusculoskeletalKnees', 'MusculoskeletalScalars',

'MusculoskeletalSpine', 'PhysicalActivity']

cols = ['Organ', 'Sample size', 'SNPs', 'Genes', 'Heritability', 'CA\_prediction\_R2']

GWAS\_summary = np.empty((len(organs), len(cols),))

GWAS\_summary.fill(np.nan)

GWAS\_summary = pd.DataFrame(GWAS\_summary)

GWAS\_summary.index = organs

GWAS\_summary.columns = cols

GWAS\_summary['Organ'] = organs

# Fill dataframe

All\_hits = pd.read\_csv(self.path\_data + 'GWAS\_hits\_' + self.target + '\_All\_withGenes.csv')

Heritabilities = pd.read\_csv(self.path\_data + 'GWAS\_heritabilities\_' + self.target + '.csv', index\_col=0)

Performances = pd.read\_csv(self.path\_data + 'PERFORMANCES\_bestmodels\_alphabetical\_instances\_Age\_test.csv')

dict\_organ\_to\_organ\_view = {

'AbdomenLiver': ['Abdomen', 'Liver'],

'AbdomenPancreas': ['Abdomen', 'Pancreas'],

'ArterialCarotids': ['Arterial', 'Carotids'],

'ArterialPulseWaveAnalysis': ['Arterial', 'PulseWaveAnalysis'],

'BiochemistryBlood': ['Biochemistry', 'Blood'],

'BiochemistryUrine': ['Biochemistry', 'Urine'],

'BrainCognitive': ['Brain', 'Cognitive'],

'BrainMRI': ['Brain', 'MRI'],

'EyesFundus': ['Eyes', 'Fundus'],

'EyesOCT': ['Eyes', 'OCT'],

'HeartECG': ['Heart', 'ECG'],

'HeartMRI': ['Heart', 'MRI'],

'MusculoskeletalFullBody': ['Musculoskeletal', 'FullBody'],

'MusculoskeletalHips': ['Musculoskeletal', 'Hips'],

'MusculoskeletalKnees': ['Musculoskeletal', 'Knees'],

'MusculoskeletalScalars': ['Musculoskeletal', 'Scalars'],

'MusculoskeletalSpine': ['Musculoskeletal', 'Spine']

}

for organ in organs:

organ\_hits = All\_hits[All\_hits['organ'] == organ]

data\_organ = pd.read\_csv(self.path\_data + 'GWAS\_data\_Age\_' + organ + '.tab')

GWAS\_summary.loc[organ, 'Sample size'] = len(data\_organ.index)

GWAS\_summary.loc[organ, 'SNPs'] = len(organ\_hits['SNP'].unique())

GWAS\_summary.loc[organ, 'Genes'] = len(organ\_hits['Gene'].unique())

if organ in Heritabilities.index:

GWAS\_summary.loc[organ, 'Heritability'] = str(round(Heritabilities.loc[organ, 'h2'] \* 100, 1)) + '+-' \

+ str(round(Heritabilities.loc[organ, 'h2\_sd'] \* 100, 1))

if organ in Performances['organ'].values:

GWAS\_summary.loc[organ, 'CA\_prediction\_R2'] = \

str(round(Performances[Performances['organ'] == organ]['R-Squared\_all'].values[0] \* 100,

1)) + '+-' + \

str(round(Performances[Performances['organ'] == organ]['R-Squared\_sd\_all'].values[0] \* 100, 1))

else:

org, view = dict\_organ\_to\_organ\_view[organ]

GWAS\_summary.loc[organ, 'CA\_prediction\_R2'] = \

str(round(Performances[(Performances['organ'] == org) &

(Performances['view'] == view)]['R-Squared\_all'].values[0] \* 100,

1)) + '+-' + \

str(round(Performances[(Performances['organ'] == org) &

(Performances['view'] == view)]['R-Squared\_sd\_all'].values[0] \* 100, 1))

# Format dataframe

for col in ['Sample size', 'SNPs', 'Genes']:

GWAS\_summary[col] = GWAS\_summary[col].astype(int)

# Save data

print('GWAS summary:')

print(GWAS\_summary)

GWAS\_summary.to\_csv(self.path\_data + 'GWAS\_summary' + self.target + '.csv', index=False)

# Report the correlation between R2s and h2\_gs:

GWAS\_summary.dropna(subset=['Heritability', 'CA\_prediction\_R2'], inplace=True)

# Local helper function

def extract\_h2r2(row):

h2 = float(row['Heritability'].split('+-')[0])

r2 = float(row['CA\_prediction\_R2'].split('+-')[0])

return pd.Series([h2, r2])

h2r2s = GWAS\_summary.apply(extract\_h2r2, axis=1)

Corr\_h2s\_r2s = h2r2s.corr().loc[0, 1]

print('Correlation between R2s when predicting chronological age and h2\_g when performing the GWAS = ' +

str(round(Corr\_h2s\_r2s, 3)))

def upload\_data(self):

files = ['snps\_rs.txt', 'GWAS\_genes\_rs.txt', 'snps\_chrbp.txt', 'GWAS\_genes\_chrbp.txt', 'All\_hits\_missing.csv',

'GWAS\_unique\_genes.npy']

for organ in ['All'] + self.organs\_XWAS:

files.append('GWAS\_hits\_' + self.target + '\_' + organ + '\_withGenes.csv')

files = [self.path\_data + file for file in files]

for file in files:

os.system('scp ' + file +

' al311@transfer.rc.hms.harvard.edu:/n/groups/patel/Alan/Aging/Medical\_Images/data/')

class GWASPlots(Basics):

"""Generates Manhattan and QQ plots to summarize the GWASs results."""

def \_\_init\_\_(self, target=None):

Basics.\_\_init\_\_(self)

self.target = target

self.organs = \

[file.split('\_')[2].replace('.csv', '') for file in glob.glob(self.path\_data + 'GWAS\_' + target + '\_\*.csv')]

self.organs.sort()

self.FDR\_correction = 5e-8

# 23 colors for plot, to maximize two by two contrast (mostly important for the volcano plot)

self.dict\_chr\_to\_colors = {'1': '#b9b8b5', '2': '#222222', '3': '#f3c300', '4': '#875692', '5': '#f38400',

'6': '#a1caf1', '7': '#be0032', '8': '#c2b280', '9': '#848482', '10': '#008856',

'11': '#555555', '12': '#0067a5', '13': '#f99379', '14': '#604e97', '15': '#f6a600',

'16': '#b3446c', '17': '#dcd300', '18': '#882d17', '19': '#8db600', '20': '#654522',

'21': '#e25822', '22': '#232f00', '23': '#e68fac'}

self.dict\_colors = {'light\_gray': '#b9b8b5', 'black': '#222222', 'vivid\_yellow': '#f3c300',

'strong\_purple': '#875692', 'vivid\_orange': '#f38400', 'very\_light\_blue': '#a1caf1',

'vivid\_red': '#be0032', 'grayish\_yellow': '#c2b280', 'medium\_gray': '#848482',

'vivid\_green': '#008856', 'dark\_gray': '#555555', 'strong\_blue': '#0067a5',

'strong\_yellowish pink': '#f99379', 'violet': '#604e97', 'vivid\_orange\_yellow': '#f6a600',

'strong\_purplish\_red': '#b3446c', 'vivid\_greenish\_yellow': '#dcd300',

'strong\_reddish\_brown': '#882d17', 'vivid\_yellow\_green': '#8db600',

'vivid\_yellowish\_brown': '#654522', 'vivid\_reddish\_orange': '#e25822',

'deep\_olive\_green': '#232f00', 'strong\_purplish\_pink': '#e68fac'}

def generate\_manhattan\_and\_qq\_plots(self):

GWAS\_All = None

for organ in self.organs + ['All']:

print('Generating Manhattan plot and QQ plot for ' + organ)

# Preprocessing

if organ == 'All':

GWAS = GWAS\_All

else:

GWAS = pd.read\_csv('../data/GWAS\_Age\_' + organ + '.csv', usecols=['SNP', 'CHR', 'BP', 'P\_BOLT\_LMM\_INF'])

GWAS.set\_index('SNP', drop=False, inplace=True)

Genes = pd.read\_csv('../data/GWAS\_hits\_Age\_' + organ + '\_withGenes.csv', index\_col='SNP',

usecols=['SNP', 'Gene'])

GWAS['Gene'] = Genes['Gene']

GWAS.loc[GWAS['Gene'].isna(), 'Gene'] = 'not significant'

# replace 0 with numerical limit for p-values so that log can be safely taken by mhat

GWAS['P\_BOLT\_LMM\_INF'] = GWAS['P\_BOLT\_LMM\_INF'].replace([0], [10 \*\* (-323)])

# Create a column to label SNPs without ambiguity for 'All' organs (can have several hits for same SNP)

GWAS['SNP\_P'] = GWAS['SNP'] + '\_' + GWAS['P\_BOLT\_LMM\_INF'].astype(str)

# Generate dictionary of annotations for Genes

GWAS.sort\_values(by='P\_BOLT\_LMM\_INF', inplace=True)

Genes\_annotations = {}

for chro in GWAS[GWAS['Gene'] != 'not significant']['CHR'].unique():

df\_chr = GWAS[GWAS['CHR'] == chro]

Genes\_annotations.update({df\_chr['SNP\_P'][0]: df\_chr['Gene'][0]})

GWAS.sort\_values(by=['CHR', 'BP'], inplace=True)

# Generate Manhattan plot

color = [self.dict\_chr\_to\_colors[str(ch)] for ch in GWAS['CHR'].unique()]

kwargs = {'df': GWAS, 'chr': 'CHR', 'pv': 'P\_BOLT\_LMM\_INF', 'gwas\_sign\_line': True,

'gwasp': self.FDR\_correction, 'color': color, 'gfont': 4, 'gstyle': 1, 'r': 600, 'dim': (9, 4),

'axlabelfontsize': 11, 'markernames': Genes\_annotations, 'markeridcol': 'SNP\_P'}

if GWAS['P\_BOLT\_LMM\_INF'].min() < 10 \*\* (-23):

kwargs.update({'ylm': (0, -np.log10(GWAS['P\_BOLT\_LMM\_INF'].min()),

-np.log10(GWAS['P\_BOLT\_LMM\_INF'].min()) / 10)})

visuz.marker.mhat(\*\*kwargs)

os.rename('manhatten.png', '../figures/GWAS/GWAS\_ManhattanPlot\_' + self.target + '\_' + organ + '.png')

# Generate QQ plot

# Prepare data

observed = GWAS['P\_BOLT\_LMM\_INF'].sort\_values()

lobs = -np.log10(observed)

expected = np.array(range(len(lobs))) + 1

lexp = -(np.log10(expected / (len(expected) + 1)))

# Plot figure

plt.clf()

plt.scatter(lexp, lobs, marker='\*', color='b', s=3)

plt.plot(lexp, lexp, color='r')

plt.xlabel('Expected -log10(p)')

plt.ylabel('Observed -log10(p)')

plt.savefig('../figures/GWAS/GWAS\_QQPlot\_' + self.target + '\_' + organ + '.png', dpi=600)

# Append the dataframe to the union of all dataframes for the 'All' plot

if organ != 'All':

if GWAS\_All is None:

GWAS\_All = GWAS

else:

GWAS\_All = GWAS\_All.append(GWAS)