READ ME: CODE TO RUN THE MODELS

This document provides an overview of the name, contents, function, and location of each file. To run main.py it is required to provide the path where the data is stored (under final_imputed_dataset.csv, see “READ ME_data_and_MICE”). I’ve attached a requirements.txt file which lists all required packages for your python interpreter to run this analysis.

Name	Function (callable)	Main functionalities
main.py	__init__(Path = "V:/UserData/079915/ Thesis -- Data", Tune = 0, Final = 3, nr_folds = 4, Normalize = False, currentrun = ‘all’, tweedie = False, tweedie_rho = 1.6)	This file first calls ‘DataClass.py’ to create the necessary folds. Then, based on the value of ‘currentrun’, ‘tune’ and ‘final’, this file initiates tuning of hyperparameters using train and validation folds, and predicting on test folds.
DataClass.py	__init__(Path = None, normalization=False, folds=5, seed=0)	This file is used to create folds. Specifically, we first delete left-over aberrant observations based on age. We then modify the data set to only include one observation per individual and create a stratified 70/30 train/test split. We then create an extra test set containing 10,000 last observations of individuals, using that 70% train set. Lastly, we create nr_folds folds for cross validation using the 70% train set minus the 10,000 observations for the extra test set. This results in one test set containing only new individuals, one test set containing only observations for existing individuals, and nr_folds sets of train/validation folds.
	load_data(path = None)	This function loads the dataset from a pre-specified path as y, X_gaus, X_bin.
	normalize(X, normalization)	This function performs normalization on the continuous variables in the regressors.
utils.py	__init__(model)	This file is initiated only when ‘final’ in ‘main.py’ is equal to three, and is made with the intention of retrieving quick stats on the models.
	writecsv(path, suffix=’’)	This function writes the predictions of the model to a csv file on your device, along with the actual haemoglobin values. Suffix can be used to create a suffix to the file name if desired.

	perc_correctlyspec(threshold = 47)	This function calculates the percentage correctly specified based on a certain threshold, i.e., how many predicted values are below (above) the threshold when the true values are also below (above) the threshold relative to the total number of predicted observations.
	perc_deviation(threshold = 5)	This function calculates the percentage of predictions which are within a [true – threshold; true + threshold] interval of the true value.

merfadjusted.py / merfadjusted-ANN.py[1]	__init__(fixed_effects_model=None, gll_early_ stop_threshold=None, max_iterations=20,)	This file is an adjusted version of open source MERF code, which allows for much quicker calculations and requires less memory[2]. This is the core class to instantiate, train, and predict using a mixed effects random forest model. It roughly adheres to the sklearn estimator API. Note that the user must pass in an already instantiated fixed-effects model that adheres to the sklearn regression estimator API, i.e. must have a fit() and predict() method defined.
	predict(X = np.ndarray, Z: np.ndarray, clusters: pd.Series)	This function makes predictions using trained mixed-effects models. For known clusters the trained random effect correction is applied. For unknown clusters the pure fixed effect estimate is used.
	fit(X: np.ndarray, Z: np.ndarray, clusters: pd.Series, y: np.ndarray, X_val: np.ndarray = None, Z_val: np.ndarray = None, clusters_val: pd.Series = None, y_val: np.ndarray = None,)	This function fits the ME model using Expectation-Maximization algorithm.
	get_bhat_history_df	This function retrieves a list of dataframes for the b_hat_history into a multi-indexed dataframe.

ANN.py	__init__(dataset, tune=False,	If tune is true, a Keras neural network is built and its hyperparameters are tuned using the

[1] The only difference between 'merfadjusted.py' and 'merfadjustedANN.py' is the way the model is loaded in due to issues with Keras. Their functionality remains the same.

[2] User warning: my modified version (as it stands now) takes Z as input, but regardless of what input is given, it always uses a vector of ones as random effects features.

	normalization='none', params={})	mean RMSE over nr_folds folds. All negative predictions are casted to be equal to zero. If tune is false, a Keras neural network is built with either the parameters from the tuned model, or user-specified parameters (depending on the input in 'main.py'), which in turn is used to make predictions on both test sets.
MeANN.py	__init__(dataset, tune=False, normalization='none', params={})	If tune is true, a Keras neural network is built and is used as input for the fixed-effects estimation in the mixed-effects model. Its hyperparameters are tuned using the RMSE over one fold (the first to be specific). All negative predictions are casted to be equal to zero. Since training this model requires more memory than provided by the computing facilities at EMC, this file is programmed to write its outcomes to excel in real-time, such that the best model outcome is chosen after the program terminates due to memory error. If tune is false, a Keras neural network is built with either the parameters from the tuned model, or user-specified parameters (depending on the input in 'main.py'), which in turn is used in a mixed-effects model to make predictions on both test sets.
XtremeGBoost.py	__init__(dataset, tune=False, normalization='none', params={}, tweedie=False, p=None)	If tune is true, a XGBoost model is built and its hyperparameters are tuned using either the mean RMSE of mean tweedie loss over nr_folds folds depening on whether tweedie = False or True. All negative predictions are casted to be equal to zero. If tune is false, a XGBoost model is built with either the parameters from the tuned model, or user-specified parameters (depending on the input in 'main.py'), which in turn is used to make predictions on both test sets.
MeXtremeGBoost.py	__init__(dataset, tune=False, normalization='none', params={})	If tune is true, a XGBoost model is built and is used as input for the fixed-effects estimation in the mixed-effects model. Its hyperparameters are tuned using the RMSE over 1 fold (the first to be specific). All negative predictions are casted to be equal to zero. If tune is false, a XGBoost model is built with either the parameters from the tuned model, or user-specified parameters (depending on the input in 'main.py'), which in turn is used in a mixed-effects model to make predictions on both test sets.