Simulation Study: Hyperparameter Tuning with the DoubleML Package

Georg Zhelev

November 23, 2021
Working Paper

Abstract

This project creates a programming framework in python to train multiple high-dimensional data sets using multiple parameter tuning strategies and learners while parallelizing the estimation on high performance computing server. The DoubleML package for causal machine learning is used, which provides implementation of the double / debiased machine learning framework of Chernozhukov et al. (2018)[1]. The parametric partially linear regression (PLR) model is applied and performance measures such as bias, coverage, mean squared error and standard deviation are calculated for the different training settings. Learners such as lasso, random forest, gradient boosting for both regression and classification are applied to estimating the nuisance functions of the outcome and treatment variables. A combination of the best learners for each nuisance component is chosen. Results are presented on per strategy basis and show that the most effective strategy is tuning on folds instead of on the entire sample.

1 Introduction

DoubleML is an R and Python package that provides implementation of the double / debiased machine learning framework of Chernozhukov et al. (2018)[1]. The package provides functionalities to estimate parameters in causal models based on machine learning. Details of its functionalities, theoretical backing as well as case studies are provided by Bach et al. (2021) [2].

1.1 Strategies for Parameter Tuning

The DoubleML package is used to estimate the effect of a low-dimensional treatment parameter in the presence of high-dimensional potential covariates. The package implements an algorithm, which removes confounding bias, common in non-randomized observational data and removes regularization bias, resulting from the implementation of machine learning methods. Further by splitting the sample and averaging the two resulting estimators removes the inherent overfitting bias when using flexible methods such as random forest. Since machine learning methods are used for the estimation in the DoubleML package, their parameters need to be tuned so they optimally perform. Using the package it is not clear, what tuning strategies should be applied to make the learners perform best. Since sample splitting is both used for the tuning of

hyper-parameters, but also for the removal of overfitting bias in the double /debiased approach, it is not clear how they interact with one another.

The splits for parameter tuning and for model fitting are not necessarily identical. For example, the entire sample could be split into two, perform the tuning on the first sample and then pass the obtained parameters to fit a model on the second sample. The tuning can also be done fold-specific, a setting called `tune_on_folds = TRUE` pictured in Figure 1. The tuning can also be done on the entire sample. Thus, it is interesting to see if tuning and cross-fitting on the same folds performs on average better than tuning and cross-fitting on the entire sample. Tuning on the entire sample and then passing the hyperparameters to the DoubleML can lead to overfitting.

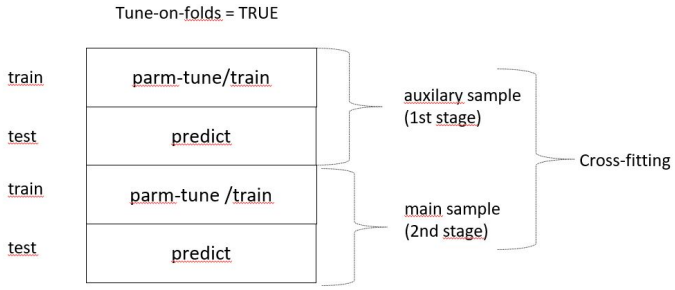


Figure 1: Sample-Splitting

1.2 Data

The data for the simulation study comes from the Atlantic Causal Inference Conference 2019 Data Challenge [4]. In this challenge, a great number of data sets have been generated in a way that they mimic distributional relationships that are found in many economic real data applications. Although the data and the variables behind it or not defined (specifically have a meaning), they are well-suited for demonstration purposes. The object of the 2019 Data Challenge is the ATE parameter. There are low and high dimensional tracks. Covariates are simulated or drawn from 7 source datasets: healthcare, business, social. Covariates come from real-world and simulated sources. Outcome and treatment variables are simulated so they allow for easy main terms models, have poor overlap, a complex functional form and treatment effect heterogeneity and IVs are also present.

Total there are 6400 datasets (32 DGPs per track). Teams analyze data and submit results files. Covariate Data Sources for the High-Dimensional Track come from the UCI Machine Learning Repository [] https://archive.ics.uci.edu/ml/index.php, Columbia University [?] http://www.stat.columbia.edu/ gelman/arm/examples/ and from simulations [] ACIC 2019 Presentation. From the 3,200 high-dimensional data sets over 32 DGPs, 1,600 data sets over 16 DGPs have been pre-processed for the project. These are the data sets which have a continuous outcome variable and a binary treatment. Each DGP has 100 data sets generated from it. Each data set has 1000 observations over 202 variables, of which, one is the outcome variable Y, and one the treatment variable D and a set of 200 potential covariates X. Since the data was simulated, it is also provided which data set belongs to which DGP and the true average treatment effect ATE is also provided.

1.3 Model

The inference problem is to determine the causal effect of the binary variable D on the outcome variable Y, while controlling for potential covariates X. There are two causal models available in the `DoubleMLData` object that can be used to estimate the ATE. First, the partially linear regression model (PLR) assumes that the treatment effect is additive and linear which implies that it does not vary across individuals. Hence the estimated coefficient is the average treatment effect ATE. Second, the interactive regression model (IRM) is a more flexible model, because it does not impose functional form restrictions on the underlying regression relationships, for example, linearity or additivity as in the PLR. This means that the model hosts heterogeneous treatment effects, i.e., account for variation in the effect across observations. In this project the PLR model is used in the estimation. An implementation with a non-parametric IRM is also planned in the future. Both models similarly require two learners for estimating the two nuisance parameters: the effect of the covariates on the treatment and the effect of the covariates on the outcome. First, the effect of the covariates on the treatment is estimated by $m_0(x_i)$ as

$$D = m_0(x_i) + U, \;\; E[U|X, D] = 0.$$

This equation keeps track of confounding. Second, the effect of the covariates on the outcome is estimated by $g_0(x_i)$ as

$$Y = D\theta + g_0(x_i) + V, \;\; E[V|X] = 0,$$

where Y is the outcome variable and D is the policy treatment variable of interest. Vector $X = (X_1, ..., X_p)$ consists of a high-dimensional vector of controls also referred to as potential covariates. The estimation is a two-stage process. First the effect of X is partialed out from D on an auxiliary sample and the an an estimate of Y is obtained in the second stage on the main sample. Therefore there are two nuisance function to estimate $m_0(x_i)$ and $g_0(x_i)$, also referred to ml-m and ml-g in later parts of the paper.

1.4 Method

Different learners are tried for the estimation of the two nuisance functions. Since there are two nuisance functions to estimate $m_0(x_i)$ and $g_0(x_i)$, the second one is always estimated with a regression learner, because the estimated outcome variable is continuous. The first one is estimated with a varying regression or classification learner, because the estimated treatment variable is binary. The learners lasso with cross validation, random forest and extreme gradient boosting are implemented. Later a best learner, which can result from any combination of the nuisance learners is chosen based on the lowest tuning residual. The performance of the tuning strategies as well as learners is evaluated by looking at tuning residuals, bias, confidence interval coverage and mean squared error. The goal is to run a large sample of data sets to determine, which tuning strategy and learner performs best on the 16 data generating processes of the high-dimensional features. Because of the large number of data sets the simulation is run on a high-performance-computing (HPC) server. Since data is simulated and the true ATE is known, this enables a comparing performance across tuning strategies and learners by looking at metrics such as bias, mean squared error (MSE), coverage

and standard deviation of the MSE. The next section describes the structure of the estimation procedure, which includes data loading, implementation of tuning strategies, learners used for estimation, tuning-grids with hyperparameters, training, choice of best learner and performance metrics. The third section covers results per tuning strategy and per learner. The conclusion section mentions areas of further work and improvements to the current python-code.

2 Estimation Procedure

The structure of the estimation procedure and how it proceeds to estimate ATE coefficients is described next. The procedure takes data that has already been simulated, therefore simulates no new data itself. Its components are

- data extraction and loading,
- set up of tuning grids,
- implementation of six tuning strategies,
- saving of results and tuned-parameters,
- compilation of results in a single data-file,
- choice of best learner, and
- calculation of performance statistics and visualizations.

The code can be applied to one or multiple data sets depending on computing capacity. The current version allows for the processing of any of the 1,600 high-dimensional data sets of the 2019 ACIC Challenge.

2.1 Data Loading

The data loading procedure allows for the programmatic extraction of data sets based on: DGP-id, manual user-selection of data sets or a random selection of n data sets. The procedure assumes a downloaded copy of the zip file with the high-dimensional data sets. Due to the large size of the data sets it is preferable to save data sets in a different repository as the code-repository in order to allow for version control separated from data-saving. A meta file has been prepared which covers all 1,600 high-dimensional data sets with meta statistics such as: *data-set-id*, *DGpid* and *trueATE*. This file is used to determine which data sets are extracted and loaded and to refer to the true ATE to evaluate performance.

2.2 Training

The six tuning strategies listed in Table 1 consider: tuning on folds, tuning on the whole sample, no tuning and tuning on a hold-out sample with 70%, 50%, and 30% of the data being used exclusively for parameter tuning.

Table 1: Tuning Strategies

Strategy	Details
Tuning on folds	tune parameters and cross-fit on same folds
Global	tune on entire data sample
none	no tuning, empty grids
Hold-out 30	tune on 70% of data sample
Hold-out 50	tune on 50% of data sample
Hold-out 70	tune on 30% of data sample

First, tuning on folds, will split the sample into the folds that are also used for cross-fitting. This procedure is computationally more intense than the others. Second, tuning on the entire sample, splits the data into K-folds, which are then used for the parameter tuning. The cross-fitting is performed independently from the tuning. Third, no tuning is used, which will probably perform the least, because there is no one-size-fits-all parametrization. No tuning acts as a helpful benchmark to illustrate the need to tune parameters. The fourth strategy is cross-fitting on a hold-out sample by splitting the data set into two parts - first used for the cross-fitting part and the second for parameter search. Different sample splits 30%, 50% and 70% are used for parameter tuning. Each tuning strategy sets up a new `doubleml.DoubleMLData` object where the outcome features and treatment variables are defined. A PLR model is set up using two learners for each nuisance component and the following settings:

Table 2: DoubleML-Settings

Number of folds	5
Repetition	1
Score function	partialling out
dml algorithm	dml2
significance level	95%

Only by the hold-out tuning strategy the data sample is split into two, out of which two `doubleml.DoubleMLData` result, one of which is used for parameter tuning and the other for model fitting. There are two nuisance components to estimate $m_0(x_i)$ and $g_0(x_i)$ in the PLR model: the effect of the covariates on the treatment and the effect of the covariates on the outcome. Because the treatment nuisance component is binary either a regression or classification can be used. The nuisance component of the outcome in turn is only estimated with a regression, because its value is continuous. This allows for a combination of regression only and a mixture of regression and classification learners. Since lasso, random forest and extreme gradient boosting are the learners of choice, considering their combinations increases the number of learners to six as shown in Table 3.

Table 3: Learners

Learner	ml-m	ml-g
lasso-reg	lasso regressor	lasso regressor
lasso-class	lasso regressor	logistic regression w/ Lasso penalty
rf-reg	rf regressor	rf regressor
rf-class	rf regressor	rf classifier
xgb-reg	xgb regressor	xgb regressor
xgb-class	xgb regressor	xgb classifier

2.3 Tuning-Grids

Lasso's tuning grid, where regression is used to estimate both nuisance components, has a cross-validation (CV) setting of five folds and 2000 iterations. The optimal alpha parameter is chosen automatically through CV. The scoring method used to evaluate the predictions is set to `negative mean squared error`, which due to the scikit-learn's randomized search documentation is the appropriate scoring method to evaluate a regression-learner. The absolute value of this scoring method is taken. The tuning grid for the lasso regressor is otherwise empty. In the case of the lasso tuning strategy where no tuning is implemented, `sklearn.linear-model.Lasso` is used instead of `sklearn.linear-model.LassoCV`. This is done to avoid any tuning whatsoever. Lasso's tuning grid, where classification is used to estimate the nuisance component of the treatment is based on logistic regression with five folds of cross-validation and 2000 iterations. The tuning grid allows for a *liblinear*, *lbfs*, *newton-cg*, *sag* and *saga* solvers with a penalty type ranging between l1 and l2. `Sklearn`'s `linear-model.LogisticRegression` uses the inverse of regularization strength as the regularization parameter C. The grid starts from the value $C = 0.0001$ and moves to the value $C = 1$ in 10 spaced evenly steps on the log scale. The scoring method used to evaluate the predictions, based on tuning parameters is set to `accuracy`. The tuning parameters are shown in Table 4

The random forest regressors and classifiers use a more sophisticated grid, which varies the depth, maximum numbers of features, minimum samples per leaf, minimum samples per split as well as the number of estimators. By max features *"auto"* are the max features set to the square root of the length of the features vector. The scoring method used to evaluate the predictions is set to `negative mean squared error` or `accuracy` respectively. For example the parameters of the ml-g learner are show in Table 5. Extreme gradient boosting uses the `xgb-regressor` and `xgb-classifier` provided by the `xgboost`. The regressor uses "squared error" while the classifier "binary: logistic" as objective with an evaluation metric "log-loss". The tuning grid varies the size of estimators, the learning rate, eta, gamma, max depth of the tree, as well as subsample and colsample by level as seen in Table 6. The scoring method is varied between `negative mean squared error` in the case of regression and `accuracy` in the case of classification.

Table 4: Lasso-Grid ml-g

Solver	liblinear
Penalty	l1, l2
Cs	log(0.0001 - 1)
Solver	lbfgs, newton-cg, sag
Penalty	l2
Cs	log(0.0001 - 1)
Solver	saga
Penalty	l1, l2
Cs	log(0.0001 - 1)

Table 5: Random Forest Grid Table

max-depth	[8, 10, 12, 15]
max-features	['auto', 'sqrt']
min-samples-leaf	[1, 2, 4, 5]
min-samples-split	[4, 5, 6, 10]
n-estimators	[100, 200, 300,]

Table 6: XGBoost Grid Table

n-estimators	range between 10-80 step by 10
learning-rate	0.001, 0.010, 0.100, 0.500
eta	0.01, 0.1, 0.2, 0,5
gamma	0, 0.5, 2, 10
max-depth	range between 5-16 step by 2
subsample	0.1 to 1 step by 0.01
colsample-bylevel	0.1 to 1 step by 0.01

Table 7: Parameter Search Settings

Type of Search	randomized search
cross-validation folds	3
search iterations	10

To tune the learners based on the grids, DoubleML's tuning method `.tune` is applied, which in the case of the python package is based on `scikit-learn`. Parameter search is done using randomized search. Parameter search on the same folds as in the cross-fitting is applied in the *tune-on-folds* strategy. For all strategies the number of cross-validation folds is set to 3, number of search iterations is set to 10. The learners are tuned using the tuning grids described above. In the case of the baseline strategy *no-tuning* empty grids for all learners are used. In order to avoid an optimum alpha value for the lasso regression in the *no-tuning* option, Lasso with cross-validation is used. The resulting tuning parameters are saved so a model can later be fit, without having to repeat parameter search. Tuning residuals, estimated coefficients, confidence intervals and standard errors are recorded into a results table indicated with the respective learner-strategy combination.

2.4 Best Learner

After data-sets walk through the training process using the different tuning strategies and learners, the collected results are compiled. In the **post-compile.py** module the results are combined in a single results table and the **best-learner.py** module is executed. Since there are six learners trained per tuning strategy and 6 tuning strategies, the results of a single data-set are 36 estimations of the average treatment effect. From the 36 the best learner-strategy combination is chosen, one for each data set. Similar to the training process, the **best-learner.py** module creates a new `DoubleMLData` and a `doubleml.DoubleMLPLR` object. The two best learners for the nuisance components given the lowest tuning-residual are chosen. An issue arises when learners with the minimal tuning-residuals come from different tuning strategies. Since the ml-g tuning-residual is always bigger than the ml-m tuning-residual, a range of 1.3 to 1.9 compared to 0.2 to 0.6, its reduction will cause the most improvement in the estimation. Therefore the ml-g learner with the lowest tuning-residual is chosen first. Because this low tuning-residual has been achieved as part of a specific tuning-strategy, the best learner of the second nuisance component will be the minimum tuning-

residual for that same tuning strategy. Therefore choose the strategy, which minimizes the larger tuning-residual and adopt the second learner given that strategy. The learner combination is chosen to be the best learner. It could be for example a combination of lasso regression for the outcome component and random forest classification for the binary nuisance component. Then a new PLR object is fit. This time instead of conducting parameter search, the saved parameters of the chosen learner are called. The model is then fitted and results are appended adding an additional 37th best-result to the data table for the data set in question. This number is made up of 6 learners times 6 tuning strategies plus one best learner. In Figure 2 is a flow-chart that visualizes the training process.

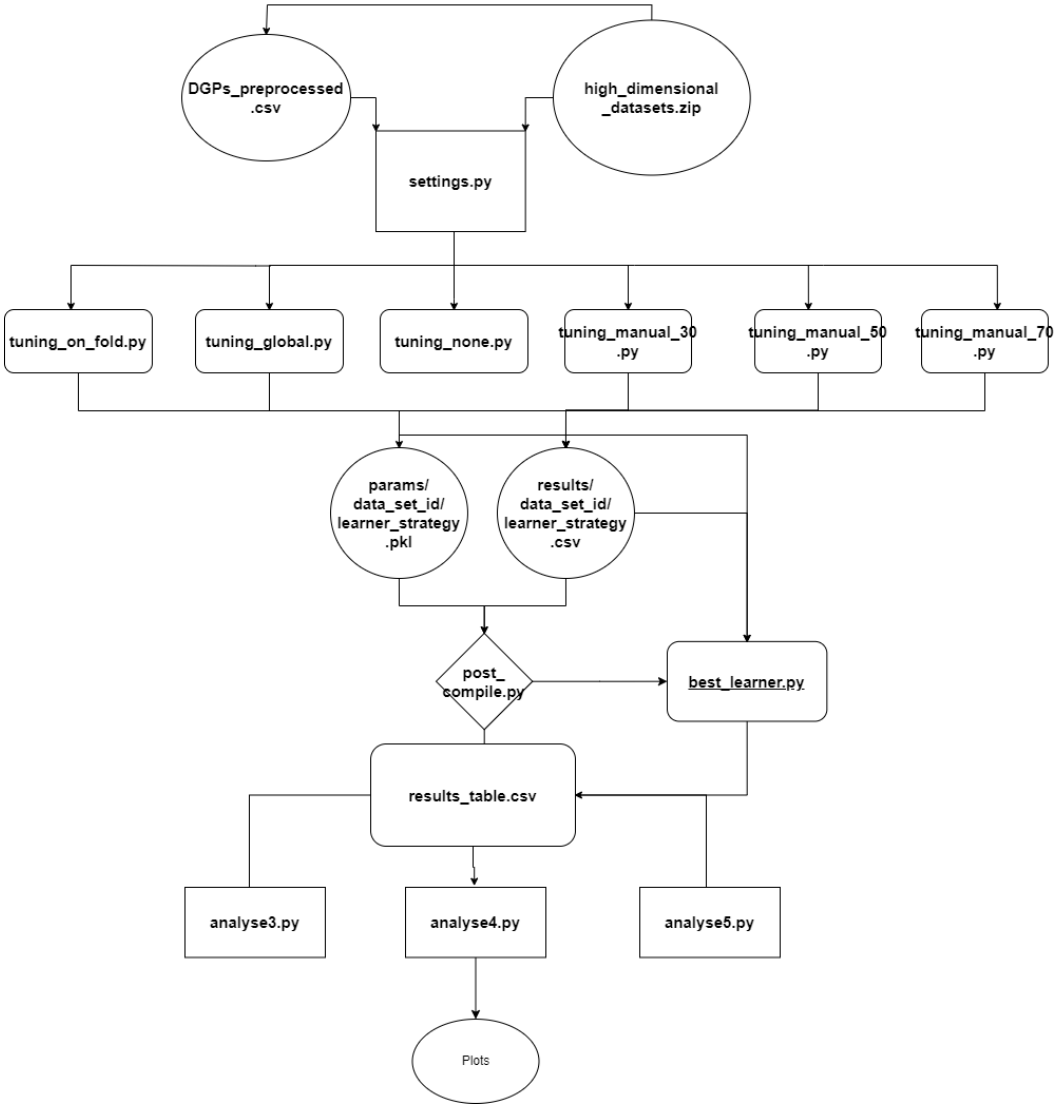


Figure 2: Files Flow-Chart

2.5 Parallelization

By default the PLR model fits all six learners consecutively. This leads to long computational times for parameter search. Parallelization of the model tuning and training at the strategy level using job arrays on the Hummer Cluster at the University of Hamburg has been implemented to solve this problem.

2.6 HPC File Directory

2.7 Compilation of Results

In addition to choosing the best learner the **post-compile.py** module compiles the results into a single csv file. The module is manually executed by the user after the data sets have been trained. Since data sets in the tens and hundreds are trained it is smart to execute this module on the server. Individual fitted results per learner-strategy as well as tuned-parameters need to be available in the folder structure in order to be included in the final **results-table.csv**. The csv file indicates the used learner, estimated coefficient, confidence interval, standard error of estimate, true coefficient, coverage of true coefficient, squared error from true coefficient, bias, data set name and DGP-id, tuning strategy, type of score, dml-algoritm, number of observations and features, tuning residuals, and nuisance component scores. A sample of results for the lasso-classification learner for three tuning-strategies is shown in Table 8.

Table 8: CSV File Results-Format

Measure	Result	Result	Result
coverage	True	True	True
learner	lasso_class	lasso_class	lasso_class
coef	0.872	0.816	0.784
ci_lower	0.736	0.582	0.597
ci_upper	1.004	1.048	0.971
std_err	0.068	0.119	0.095
true_ATE	0.8	0.8	0.8
Bias	0.070	0.016	0.016
strategy	_global	_manual_30	_manual_50
id	high924	high924	high924
n	1000	1000	1000
p	200	200	200
n_folds	5	5	5
n_rep	1	1	1
score	partialling out	partialling out	partialling out
dml_procedure	dml2	dml2	dml2
ml_g	1.3103	1.3683	1.4671
ml_m	0.5990	0.5687	0.5360
ml_total	1.909254	1.937	2.003037
msr_ml_g	mean_sqrd_err	mean_sqrd_err	mean_sqrd_err
msr_ml_m	accuracy	accuracy	accuracy
sqr_err	0.0049	0.0002	0.000253
dgp_id	2	2	2

The resulting *results-table.csv* can be loaded and evaluated separately from the training and compilation of results.

2.8 Performance Metrics

There are different ways of evaluating the results. Below is an attempt to imitate the performance evaluation conducted by the ACIC 2019 Challenge organizers. The **analyse.py** module calculates rMSE, bias, coverage and standard deviation of the rMSE as per the ACIC 2019 Challenge presentation [4] on page 7. For example rMSE is calculated according to the ACIC Data Challenge 2019 presentation as

$$rMSE = \frac{1}{16} \sum_{n=1}^{16} \frac{MSE_{m,i}}{rMSE_{oracle,i}}. \tag{2.1}$$

Relative mean squared error (rMSE) is calculated for each method (learner) m and dgp-id i. This in turn is divided by an $rMSE_{oracle}$ for each method and for each dgp-id. The reasoning is that the relative rMSE-oracle should be zero, since the true ATE is known in the simulation. Because however one can not divide by 0 the $rMSE_{oracle}$ should be a little larger than zero. Since the oracle values are not available on the challenge website nor after contacting the organizers, one can not score their submission to the challenge. Therefore certain values for oracle rMSE, bias and sd were assumed in order to allow for an evaluation of the results. These values are listed in table 9. The assumed oracle values tend to be lower/better than the results values, which creates a ratio bigger than 1. For example in the presentation of the ACIC 2019 Challenge [4] on page 8 the best achievable mean relative rMSE is 1. Any value bigger than one indicates that the value of the numerator is bigger than the value in the denominator. This makes sense, because the estimation will never be as good as the oracle estimate. Therefore the values of the calculated rMSEs are above 1 in this project. The closer the value to 1, the better the estimation. This also applies to bias. Only coverage and sd are not divided by an oracle metric, but are averaged.

Table 9: Assumed Oracle Metric

Metric	Assumed
rMSE	0.6
Bias	0.1
SD	averaged over GDP-id
Coverage	averaged over GDP-id

The squared error of each learner strategy combination is calculated according to the formula $SE = (Y_{true} - \hat{Y}_i)^2$,. Then the mean squared error is calculated over dgp id within the same learners and strategies $MSE = \frac{1}{16} \sum_{i=1}^{16} SE$. Then the rMSE is calculated by dividing by the oracle MSE for each learner/strategy combination. Coverage is a boolean value which results from the condition if the true ATE is between the lower and upper confidence interval value $\hat{Y_{lower}}_i > Y_{true} < \hat{Y_{upper}}_i$. Bias is the absolute difference between the true and estimated ATE $|Y_{true} - \hat{Y}_i|$. Standard deviation of the mean squared error is $\sigma = \sqrt[2]{\frac{\sum (SE_i - \bar{SE})^2}{n-1}}$, with $\bar{SE}$ representing the mean SE and i the learner strategy combination. Because there are six learners times six strategies, 36 results are obtained for each of the four metrics. The best learner is evaluated in a separate analysis. The 36 results are then populated into a dictionary of results. The scores are built according to the formula above or in the case of coverage and sd averages are taken.

2.9 Composite rank scores

The goal of this project is to determine differences between tuning strategies. Therefore the results are first sliced by strategy. Each strategy has its own results with the six learners. Subsequently the results of each strategy are sliced by learner and by dgp-id and the performance metrics rMSE, coverage, bias and standard deviation of the MSE are calculated. After division by the oracle metric, averages are taken over the 16 dgp-ids such as in equation 2.1. Composite rank scores are built to rank strategies. The range of the rank scores are between -16 and 16. One point is received for 1st place, 1/2 point for 2nd place, -1 point for last place and

-1/2 for next-to-last place. This allows for ranking between strategies for specific learners or metrics. For example in table 10 left, the averaged over DGP-id metrics from the lasso classification learner are shown for each strategy. The scores in the right table reflect these results. Only the rMSE and Bias have been scored, according to the scheme mentioned. The confidence interval and SD are the same as in the right table. To determine the SD of the rMSE, the SE of each dgp-id is used in the sample, therefore only a final rank is possible. The confidence interval is difficult to rank within gdp-id and then to sum up such as per 2.1, because there are ties in the ranking, which prevent the scoring. Therefore averaging was used. The rMSE and Bias allow for scoring at on the dgp-id level and for summing up over all dgp-ids. For example in 10 the global strategy has 7 points, which means that its rMSE was 7 times the lowest rMSE from all strategies.

Table 10: Composite Scores Example

(a) Lasso Classification Metrics

	rMSE	95% CI	Bias	SD
_global	2.43	0.71	4.09	1.51
_none	15.83	0.41	11.78	8.13
_manual_50	3.61	0.75	5.12	2.37
_manual_30	11.61	0.73	8.64	7.26
_on_folds	2.55	0.71	4.26	1.55
_manual_70	4.37	0.69	5.55	2.59

(b) Lasso Classification Scores and Averages

	rMSE	95% CI	Bias	SD
_global	7.0	0.71	8.0	1.51
_none	-8.0	0.41	-7.5	8.13
_manual_50	-0.5	0.75	-0.5	2.37
_manual_30	-7.0	0.73	-8.0	7.26
_on_folds	8.5	0.71	8.5	1.55
_manual_70	0.0	0.69	-0.5	2.59

3 Results

The results show in the results section are for n=207 processed data-sets from all 16 DGPs. Due to luck of the drawn some DGPs have more data sets selected than others. As the sample increases however this variation disappears.

- Data Sets evaluated: 207
- run-time: 4.5 hours
- Strategies evaluated: 6
- Learners evaluated: 6
- DGPs evaluated: 16 (binary treatment/continuous outcome)
- separate Best-Learner Analysis

3.1 Average Metrics over 16 DGP-ids

The top right of Table 11 shows the averaged over DGP-id metrics for the lasso regression learner. One can observe that the global and on-folds strategies have the lowest rMSE, bias and standard deviation of the rMSE. The hold out strategy 50 and 30 have the highest confidence interval coverage 84%. The reason for the same performance by the lasso regression learner for global and on folds tuning strategies is due to the lack of dml-induced-tuning of the lasso learner. This learner conducts its own CV tuning internally to find the best penalty parameter. Due to the implemented random seed for replication, the training yields the

same results for both strategies. This phenomenon is only present by lasso regression. All other methods have tuning grids, which due to the dml-internal-tuning, result in different metrics per strategy. The high granularity of the data in the tables makes in difficult to picture, which strategies perform well, therefore in the next section the scores and averages are graphed.

Table 11: Average Metrics over 16 DGP-ids

(a) Lasso Reg

	rMSE	95% CI	Bias	SD
_global	1.97	0.78	3.64	1.24
_none	12.14	0.40	9.72	6.51
_manual_50	3.03	0.84	4.53	2.02
_manual_30	5.49	0.84	6.11	3.36
_on_folds	1.97	0.78	3.64	1.24
_manual_70	2.44	0.83	4.02	1.58

(b) Lasso Class

	rMSE	95% CI	Bias	SD
_global	2.43	0.71	4.09	1.51
_none	15.83	0.41	11.78	8.13
_manual_50	3.61	0.75	5.12	2.37
_manual_30	11.61	0.73	8.64	7.26
_on_folds	2.55	0.71	4.26	1.55
_manual_70	4.37	0.69	5.55	2.59

(c) Rf Reg

	rMSE	95% CI	Bias	SD
_global	4.96	0.63	7.26	2.00
_none	5.18	0.62	7.45	2.20
_manual_50	7.67	0.68	8.71	3.25
_manual_30	9.48	0.73	9.86	3.94
_on_folds	4.95	0.63	7.32	1.97
_manual_70	5.75	0.67	7.62	2.46

(d) RF Class

	rMSE	95% CI	Bias	SD
_global	4.09	0.60	6.93	1.55
_none	4.23	0.63	6.99	1.68
_manual_50	6.93	0.67	8.35	2.97
_manual_30	10.18	0.71	10.22	4.19
_on_folds	4.13	0.60	6.92	1.59
_manual_70	5.10	0.68	7.22	2.15

(e) Xgb Reg

	rMSE	95% CI	Bias	SD
_global	5.07	0.49	7.78	2.01
_none	11.87	0.44	10.11	5.97
_manual_50	5.51	0.69	7.48	2.79
_manual_30	12.90	0.72	11.01	6.00
_on_folds	5.72	0.52	8.12	2.18
_manual_70	9.45	0.56	10.37	3.47

(f) Xgb Class

	rMSE	95% CI	Bias	SD
_global	7.57	0.42	9.35	3.07
_none	21.15	0.37	13.84	10.30
_manual_50	9.29	0.57	9.01	6.11
_manual_30	12.71	0.60	11.32	5.98
_on_folds	6.65	0.45	8.87	2.88
_manual_70	15.21	0.41	13.03	7.97

3.2 Graphics of Scores and Averages

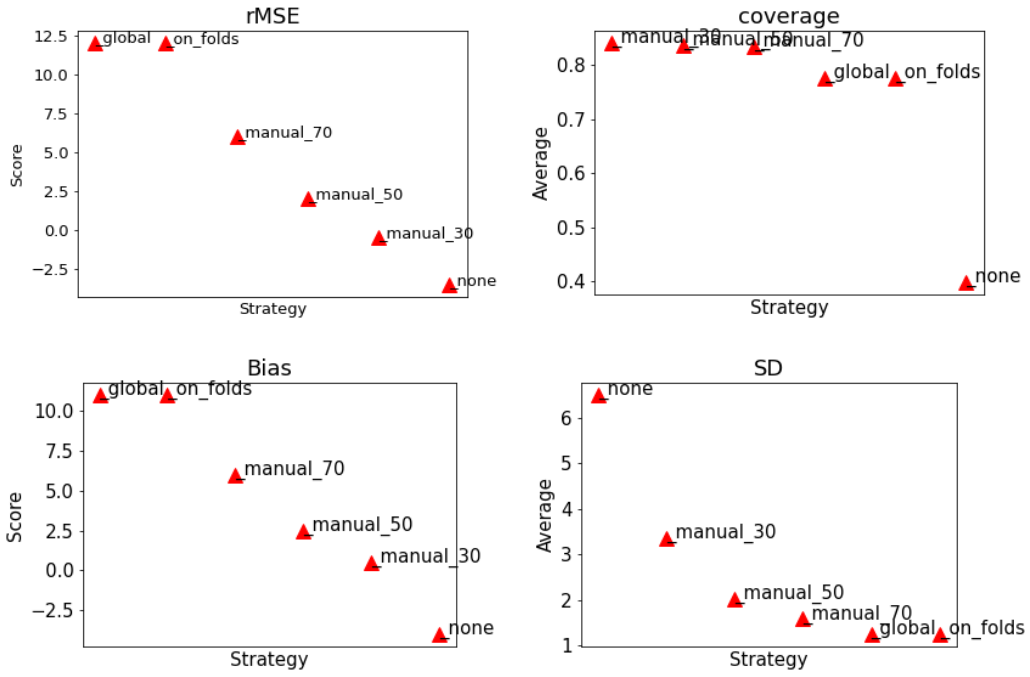
The scoring of the learners hinges on the idea that the best learners earn the most points. Bias and rMSE are easy to score, however coverage is difficult to rank into 6 clear positions, because of ties in the ranking. Therefore coverage averages instead of scores are pictured in the graphics below. Similarly the standard deviation of the rMSE is averaged over the 16 DGP-ids, because there due to sampling there is no score per dgp-id. Therefore only a single SD score results. Because of averaging coverage and SD leads to a single vector it makes more sense to simply show the averages in order to show the relative distance between the strategies. Therefore coverage and SD are in terms of averages instead of scores.

3.2.1 Lasso

In figure 3 the composite scores and averages for the lasso regression method are expressed. One can see that the global and on folds strategies have the best rMSE scores over all data generating processes. Strategies such as manual-70, manual-50 and manual-30 perform less well. Because the model is tuned on 70% of the

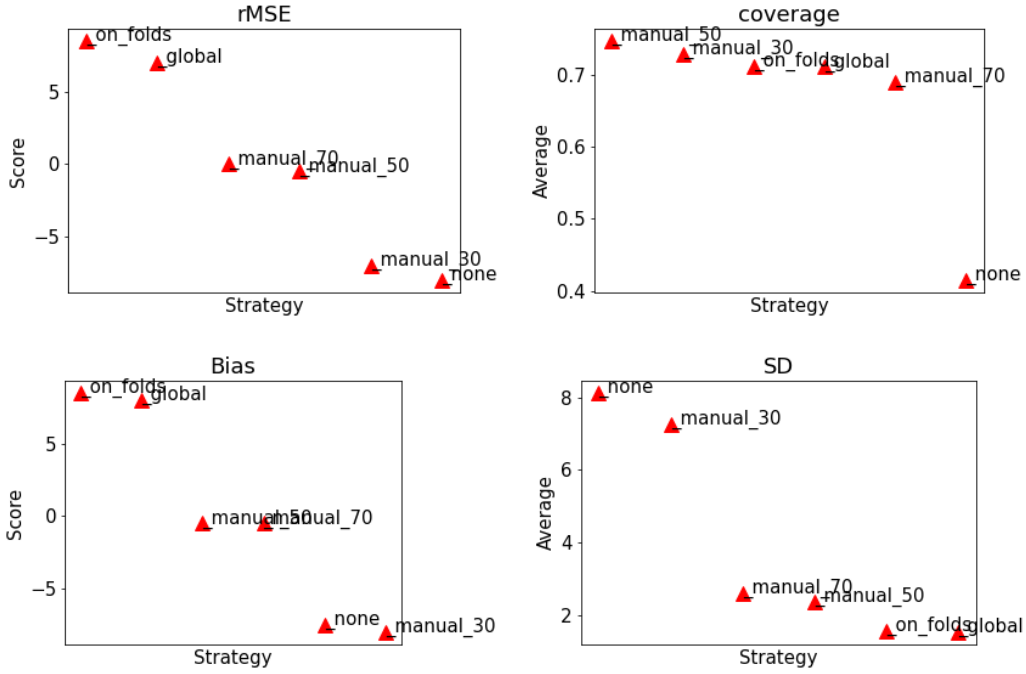
data by manual-70, this strategy performs better than strategies where the model is fit on 50 and 30%. The no-tuning strategy performs as expected the worst. By coverage the hold out strategies show better coverage than the global and on folds tuning. Especially the manual-30 strategy, which implies tuning on 70% of the sample seems to have the best coverage over all DGPs. The manual-30 strategy which only tunes on 70% of the sample performs better than the global and on folds tuning strategies. The SD graph is made up of averages therefore the lowest is the best.

Figure 3: Lasso Regressor Treatment



In figure 4 are the results for the lasso classification learner. One can observe that on folds and global strategies have the best rMSE, bias and lowest SD. Coverage is consistent with lasso regression results above higher for two hold out strategies. Because the hold out strategies use less data to fit a model than on folds and global strategies, they also tend to perform worse.

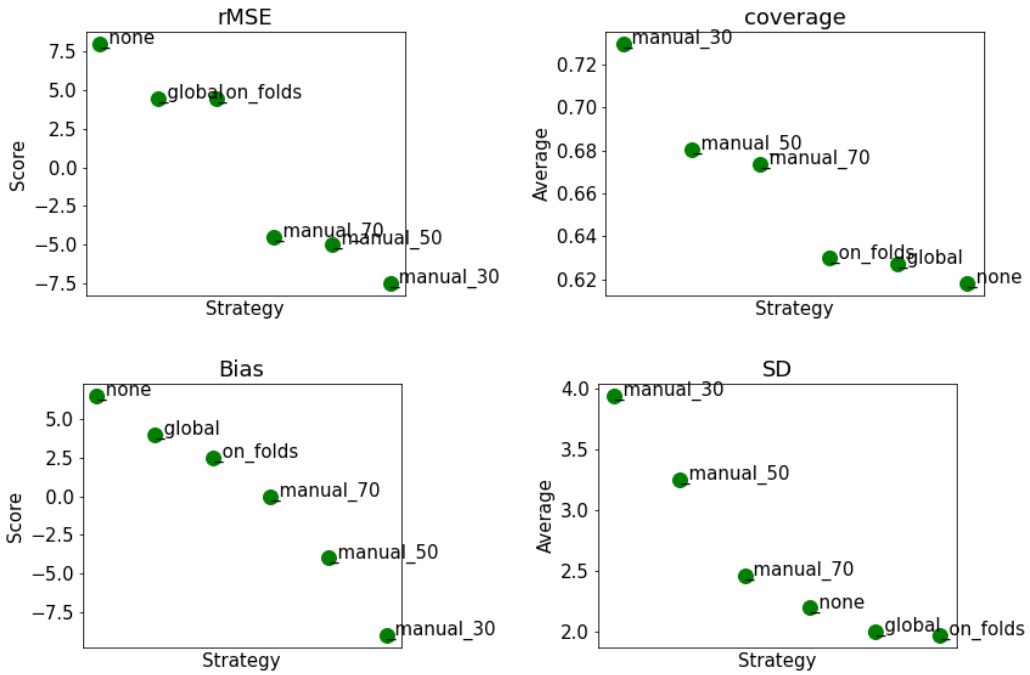
Figure 4: Lasso Classifier Treatment



3.2.2 Random Forest

In figure 5 are the results for the random forest regression for both nuisance components. It is observed that the no-tuning strategy tends to have the lowest rMSE and Bias. Although the random forest was tuned on a single data set, where there were clear improvements from a tuning grid, it seems that over multiple data sets the default settings seem to yield better results. The global and on folds strategies seem to lower the standard deviation the most. Coverage similarly to the other learners is especially high for hold-out strategies. This is begining to look like a pattern. The results are consistent with the random forest classifier treatment, which is not shown.

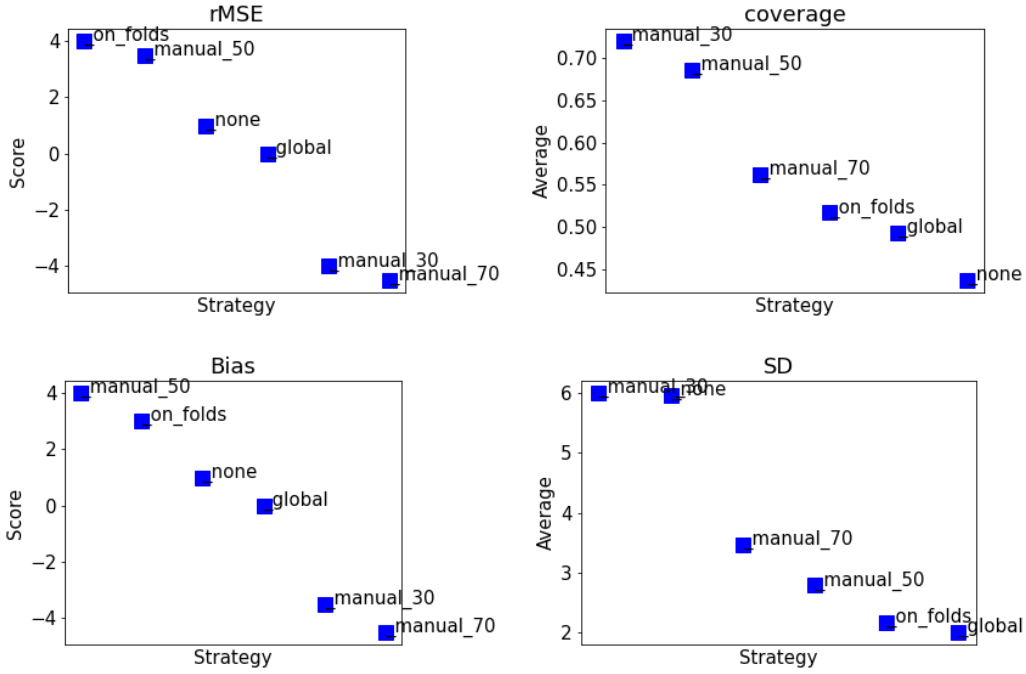
Figure 5: Random Forest Regressor Treatment



3.2.3 Extreme Gradient Boosting

The extreme gradient boosting scores in figure 6 show that the on folds and hold-out-50 strategies tend to provide the lowest rMSE and bias. The none strategy is not last in the performance for these two metrics, wich may also signify that a larger more robust tuning grid is necessery. The random forest sufured from the same issue. Similarly to the other learners the hold out strategies provide the best coverage and the results for the xgb classifier are consistent with those of the regressor.

Figure 6: Extreme Gradient Boosting Regression Composite Scores



3.3 Results-Summary

The xgb learners seem to be tuned better than the random forest learners, because they don't show the *no-tuning* strategy as the best strategy. When creating the tuning grid for the random forest it was specifically tested on a single data generating process and for that DGP it outperformed the *no-tuning* strategy, however when including the remaining 15 DGPs in the estimation, it shows the the tuning-grid is clearly not robust enough to outperform the default settings. Looking at lasso classification on-folds and global strategies seem to perform best in estimating the bias of the ATE, rMSE and SD of rMSE, but have lower coverage rates than the hold-out-strategies. The lasso results are difficult to rank and score because there are two strategies that score exactly the same. This is an exception, which is due to the lack of a tuning grid. When tuning grid is lacking the dml-tuning does not function. Combined with the provided random seed, this yields the same result for the lasso regressor global and on folds tuning strategies.

3.4 Best Learner Analysis

Because of the granularity of the results and the inability to average over learners or over metrics a best learner analysis was conducted. This analysis covers only the selected best learner and its performance over the tuning strategies. It should give clear results, which the best tuning strategy is, because it is only one learner instead of six. The results are shown for each metric: bias, MSE, coverage and SD with averaging within (over observations) and over the 16 GDP-ids. The best learner is chosen based on the tuning residual that results after the parameter search is finished. The metrics are calculated using the squared error, computed with the help of the *true-ATE*. The true-ATE is provided by the challenge organizers, since the data is simulated.

The results of the best learner are reduced, because there are dgp-ids the ATEs of which are not efficiently estimated during the training. For example DGP-id 12 and 14 have poor estimates of rMSE as shown in table 12. By these two dgp-ids big squared errors were observed, which affect averaging over dgp-ids and offset strategy averages to the point that one bad estimate can make an otherwise consistent strategy fall behind. These aberrations are consistent across the four metrics. As a result a good comparison for those DGP-ids can not be achieved. Therefore the estimates of data sets that came from these two dgp-ids where removed. The analyzed data sets per DGP-id are shown in table 13. Because of the random draws some DGP-ids have more chosen that sets than others. The best learner is chosen from a strategy/learner combination. For example specifically poor performing strategies such as the no-tuning strategy is chosen the least. Since not all strategies have best learners chosen from them, there were missing values such as seen in table 14. Missing values did not affect averages adversely. The results of the best learner analysis are shown in table 15. The on folds strategy has the lowest rMSE, highest coverage, lowest bias and lowest standard deviation. The results from table 15 have been graphed in figure 7. Due to the detail view the none strategy is not to be seen in the graphics, because its results are not near the other strategies. The graphics show that the on folds strategy is the best tuning strategy by a small margin.

dgp_id	$rMSE_{avg}$
1	0.022
2	0.009
3	0.035
4	0.005
5	0.041
6	0.120
7	0.009
8	0.011
9	0.078
10	0.078
11	0.085
12	2.689
13	0.002
14	12.128
15	0.004
16	0.001

Table 12: rMSE Average per DGP-id

	dgp_id
7	20
15	17
1	15
6	14
11	14
13	14
2	13
4	13
8	13
3	11
9	11
10	11
5	8
16	8

Table 13: Data Sets per DGP-id Counts

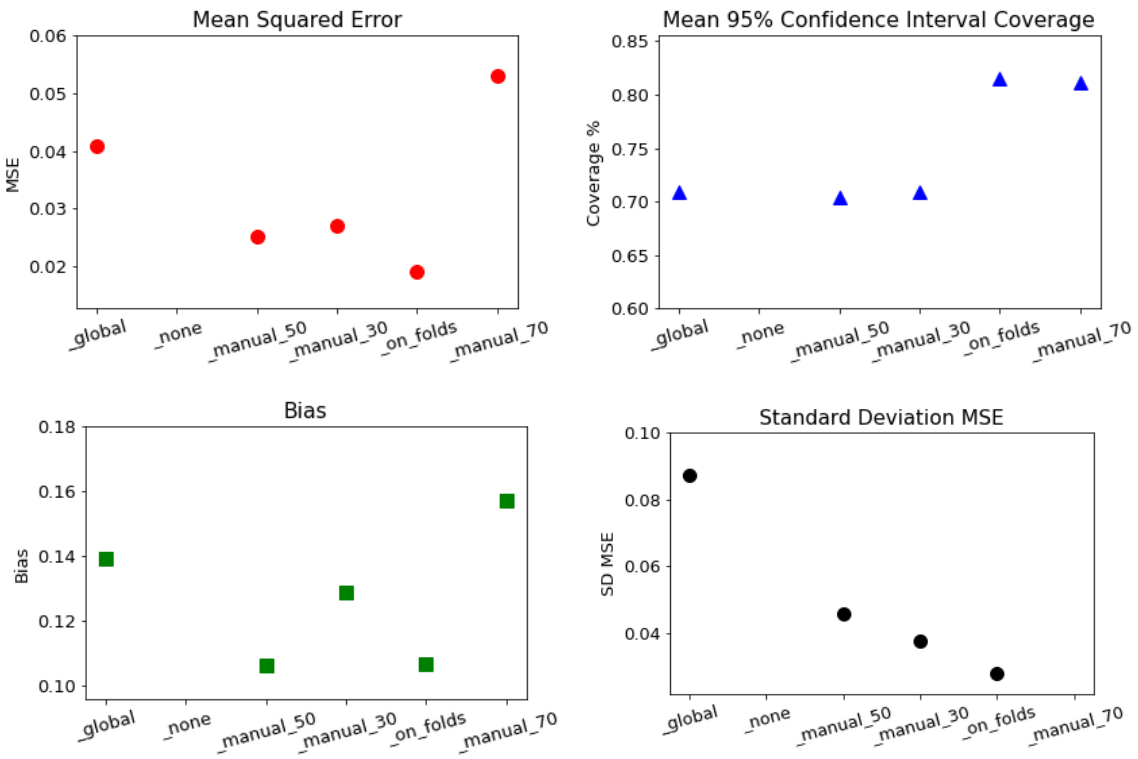
Strategy	Missing
_global	0
_none	12
_manual_50	5
_manual_30	2
_on_folds	0
_manual_70	5

Table 14: Missing Strategies

	MSE	95% CI	Bias	SD
_global	0.041	0.708	0.139	0.087
_none	0.146	0.000	0.315	0.157
_manual_50	0.025	0.704	0.106	0.046
_manual_30	0.027	0.708	0.128	0.037
_on_folds	0.019	0.815	0.107	0.028
_manual_70	0.053	0.811	0.157	0.151

Table 15: Best Learner Results

Figure 7: Best Learner Results



4 Conclusion

4.1 Results and Discussion

Looking at each learner separately the results tend to vary, because each learner has been tuned to a different degree. Nevertheless the best performing learner from the six learners tends to be the lasso regression. Because of the tuning grid of the random random forest is not robust for all DGPs, it is best not to consider its results. The extreme gradient boosting shows better signs of being tuned than the random forest, but not enough because the no-tuning strategy still performs moderate. The best learner analysis, where according to the lowest tuning residual the learners for the estimation of nuisance components are chosen, shows a clear advantage of the on folds tuning strategy. The results of the best learner had to be reduced, by removing the data set estimates of two DGPs, because of their poor estimation. In this version of the estimation, the partially linear regression (PLR) model is used. It could be the case that a non-parametric estimation procedure performs better than the chosen PLR model. For example the more flexible interactive regression model (IRM) contained in the DoubleML package. Further changes to the code would allow the training of low-dimensional data sets in addition to the high-dimensional ones. Another area of concern are the equal estimation results for the global and on-folds tuning strategies. Because of the lack of grid for LassoCV no DoubleML-internal tuning takes place. Combined with the implemented seed, this leads to equal rMSE, bias, coverage and sd results for the lasso regression only. However the estimation procedure is parallelized using job arrays and can be used to estimate more data sets on a high performance computing server with multiple processor-nodes. How to use this procedure to train further data sets is explained in Appendix 4.1.

Appendix A: Training Procedure

A Data directory

After all data sets have been extracted in the data directory, the path to it is assigned to a variable. From this variable the ".csv" file-endings are removed. 420 data sets are drawn randomly to be trained. The data sets are split into smaller groups of 20 among 8 training directories on the high performance computing (HPC) server. Access to the server directory is gained through FileZilla. As show in figure 8 directory contains the appropriate settings.py file, the six training*.py job arrays (one for each tuning-strategy) and an .sh file for the allocation of job resources through the Slurm Job Manager.

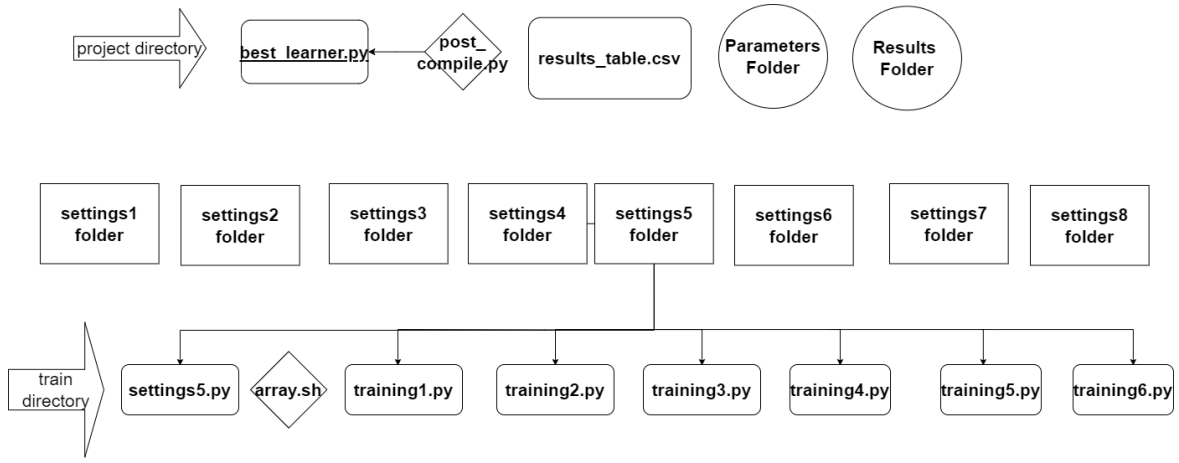


Figure 8: HPC Directory Structure

B Settings.py

The training files only work with the variable *data_sets*. Therefore for each of the eight directories on the HPC server, the local settings.py file needs to be adjusted by deleting the last number. In the case below the settings.py file is prepared for the directory named *settings1* and will train the 200th till the 220th data set. This level of data fragmentation allows for a short 4,5 hour training time per 20 data sets. Parallelized over 8 instances of the job arrays, this adds up to 8*20=160 data sets in 4,5 hours. Abour 3*160=480 should be a large enough sample for the project. The results of the project are currently evaluated at 200 samples.

```
#loaded data sets
data_sets_dir = os.listdir(path_to_data)

#remove '.csv' so can find common set below
data_sets_dir = [x[:-4] for x in data_sets_dir]

random.seed(10)
data_sets = random.sample(list(data_merged.Renamed), 420)    #how many to train
len(set(data_sets_dir).intersection(set(data_sets)))

#test-run
```

```
#data_sets = ('high956', 'high1072')

#split data sets into smaller groups for job arrays
per_data_set = int(len(data_sets)/84)      #420/84=5 data sets to train per job
#per_data_set = 5

data_sets = data_sets[per_data_set*40:per_data_set*44]                    #20 data sets
    per job array
data_sets2 = data_sets[per_data_set*44:per_data_set*48]
data_sets3 = data_sets[per_data_set*48:per_data_set*52]
data_sets4 = data_sets[per_data_set*52:per_data_set*56]
data_sets5 = data_sets[per_data_set*56:per_data_set*60]
data_sets6 = data_sets[per_data_set*60:per_data_set*64]
data_sets7 = data_sets[per_data_set*64:per_data_set*68]
data_sets8 = data_sets[per_data_set*68:per_data_set*72]
```

C Job Arrays

Then the HPC server is accessed through PuTTy SSH and using the Slurm Workload Manager the following command is given.

```
cd $WORK/Projekt/settings1
module switch env env/2015Q4-gcc-openmpi
dos2unix $WORK/Projekt/settings1/array.sh
sbatch $WORK/Projekt/settings1/array.sh
module switch env env/2021Q2-gcc-openmpi
```

This command sets the working directory to that particular directory, changes the module to an older 2015 module in order to convert the .sh file from dos to unix, then run the batch using the converted file in a new 2021 module with the latest version of python. This is a standard procedure when using the Slurm Workload Manager. The .sh file itself accessed through Notepad++ looks like

```
#!/bin/bash
#SBATCH --job-name=training_array
#SBATCH --partition=std
#SBATCH --nodes=1
#SBATCH --time=04:30:00
#SBATCH --export=NONE
#SBATCH --mail-user=georgi.zhelev@studium.uni-hamburg.de
#SBATCH --mail-type=ALL
#SBATCH --array=1-6

#1.Basi
set -e
source /sw/batch/init.sh

#2.Module
module load python/3.8.5
module switch env env/2021Q2-gcc-openmpi
```

```
#3.Arbeit
cd $WORK/Projekt/settings1

#install packages on HPC-Server
#pip install doubleml
#pip install xgboost
#pip install seaborn

export OMP_NUM_THREADS=16
/usr/bin/time python3 $WORK/Projekt/settings1/training$SLURM_ARRAY_TASK_ID.py

exit
```

D Data Compilation

The training time is set to 4.5 hours with 1-6 job arrays with a single node of the standard partition made up of 16 processors. This manages a processor utilization of about 100%. The most recent python module and environment on which the anaconda packages are installed is loaded and the working directory is set to the appropriate training directory. Finally the job arrays named *training** are executed. Each .sh file is executed using the slurm commands above, but only the directory is changed from 1 to 8 in order to execute it for all $8 * 20$ data sets. After training is complete the *post_compile* module is executed using the slumrm command.

```
cd $WORK/Projekt
module switch env env/2015Q4-gcc-openmpi
dos2unix $WORK/Projekt/post_compile_job.sh
sbatch $WORK/Projekt/post_compile_job.sh
module switch env env/2021Q2-gcc-openmpi
```

This module uses the *post_compile_job.sh* file. The module fits the best learners, which for 200 data sets could take up to 2-3 hours and compiles all the results in a single file called *results_table.csv*. Then the analysis.py files can be used locally with the csv file.

E Analysis Files

The analysis files built graphics from the results. There are three of them:

- analysis3.py builds results per estimation method (such in ACIC 2019 Data Challenge)
- analysis4.py builds results per tuning strategy (Section 3)
- analysis5.py builds results of the bet learner (Section 3)

References

[1] Philipp Bach, Victor Chernozhukov, Malte S. Kurz, Martin Spindler: *DoubleML - An Object-Oriented Implementation of Double Machine Learning in R.* Available at: https://arxiv.org/abs/2103.09603.

[2] Victor Chernozhukov, Denis Chetverikov, Mert Demirer, Esther Duflo, Christian Hansen, Whitney Newey, James Robins: Double/debiased machine learning for treatment and structural parameters. *The Econometrics Journal*,21(1): C1-68, 2018. Available at: https://onlinelibrary.wiley.com/doi/abs/10.1111/ectj.12097.

[3] Philipp Bach, Victor Chernozhukov, Malte S. Kurz, Martin Spindler: *DoubleML - An Object-Oriented Implementation of Double Machine Learning in R.* Available at: https://arxiv.org/abs/2103.09603.

[4] Gruber S., (2019): *Atlantic Causal Inference Conference 2019 Data Challenge.* Available at: https://sites.google.com/view/acic2019datachallenge/home?authuser=0.