Journal of Statistical Software

MMMMMM YYYY, Volume VV, Issue II. *doi: 10.18637/jss.v000.i00*

dame-flame: A Python Library Providing Fast Interpretable Matching for Causal Inference

Neha R. Gupta
Duke University

Vittorio Orlandi
Duke University

Chia-Rui Chang
Harvard University

Tianyu Wang
Fudan University

Marco Morucci
New York University

Pritam Dey
Duke University

Thomas J. Howell
Duke University

Xian Sun
Duke University

Angikar Ghosal
Duke University

Sudeepa Roy
Duke University

Cynthia Rudin
Duke University

Alexander Volfovsky
Duke University

Abstract

dame-flame is a Python package for performing *matching* for *observational causal inference* on datasets containing discrete covariates. This package implements the *Dynamic Almost Matching Exactly (DAME)* and *Fast, Large-Scale Almost Matching Exactly (FLAME)* algorithms, which match treatment and control units on subsets of the covariates. The resulting matched groups are interpretable, because the matches are made directly on covariates, and high-quality, because machine learning is used to determine which covariates are important to match on instead of human inputs. The package provides several adjustable parameters to adapt the algorithms to specific applications, and can calculate treatment effects after matching. The most recent source code of the implementation is available at https://github.com/almost-matching-exactly/DAME-FLAME-Python-Package

Keywords: Causal inference, machine learning, matching, Python.

1. Introduction

The **dame-flame** Python package is the first major implementation of two algorithms, the *Dynamic Almost Matching Exactly* (DAME) algorithm (Dieng, Liu, Roy, Rudin, and Volfovsky 2019, published in AISTATS'19), and the *Fast, Large-Scale Almost Matching Exactly* (FLAME) algorithm (Wang, Morucci, Awan, Liu, Roy, Rudin, and Volfovsky 2019, published in JMLR'21), which provide *almost exact* matching of treatment and control units in discrete observational data for causal analysis. As discussed in Dieng *et al.* (2019), and Wang *et al.* (2019), the two algorithms produce high-quality interpretable matched groups, by using machine learning on a holdout training set to learn distance metrics. DAME solves an optimization problem that matches units on as many covariates as possible, prioritizing matches on important covariates. FLAME approximates the solution found by DAME via a much faster backward feature selection procedure.

The DAME and FLAME algorithms are discussed in the remainder of this section. We also provide testing and installation details. In Section 2, we discuss the class structure in the **dame-flame** package, detail special features of **dame-flame**, and compare **dame-flame** to other matching packages. In Section 3, we offer examples and a user guide.

1.1. Algorithms Overview

The advantage of matching is that it accounts for confounding of treatment effect estimates, and permits interpretable analyses that are easier to troubleshoot than other types of analysis for observational causal studies. However, matching is not trivial; in high dimensional settings, few individuals can be matched exactly on all covariates, so other ways must be found. We offer the Python package **dame-flame** to support *almost exact matching* in a way that focuses on identifying important subsets of the covariates using machine learning and matching units exactly on those subsets.

dame-flame is designed for causal inference problems with a binary treatment variable, an observed outcome variable, and any number of pre-treatment covariates. Several assumptions standard in observational causal inference must be made in order for the DAME and FLAME algorithms to be applicable. One is the Stable Unit Treatment Value Assumption (SUTVA), which assumes that treatments applied to one unit do not affect the outcome of other units, and there is only one version of treatment. A second requirement is that of unconfoundedness, or ignorability. It is important that the outcome is independent of the treatment assignment. A final requirement is overlap of treatment and control groups. The treatment and control groups are said to not have any overlap at some location in a distribution when the probability of being treated at that location is either exactly 0 or 1. If there is no overlap for all covariates, then FLAME and DAME algorithms would not be able to find any matches, and no treatment effect estimates would be possible, since either treatment or control would never be observed at x. A more moderate issue is when only few treated and control units overlap in covariate values, or partial overlap, where we may not find both treatment and control units with sufficient overlap to match with. In this case, the user's settings on the **dame-flame** package would determine what *quality of matches* would be acceptable. Match quality is discussed later in this section. Units that were not able to be matched are not included in treatment effect calculations. Finally, discrete observed covariate data is a requirement of **dame-flame**. We do not recommend that users bin continuous data, with an exception for scenarios in which users are confident they are binning variables in a way that is meaningful for their

research.

dame-flame is efficient, owing to a combination of fast bit-vector computations, and a backwards feature elimination process (for FLAME) or a type of downwards closure property for systematic feature elimination (for DAME). Therefore, FLAME is faster, but DAME is able to match units on more covariates.

We also support a hybrid execution of FLAME and DAME methods. The combination of FLAME (at earlier iterations) and DAME (at later iterations) permits faster elimination of irrelevant covariates in the earlier iterations and then a more careful elimination of covariates in the later iterations, thereby achieving a trade-off between scalability and quality.

1.2. Algorithm Methodology

In this section we describe the algorithms implemented. First we discuss the mathematical problem that FLAME and DAME aim to solve, then we describe the steps of each algorithm.

Suppose we have n units, indexed by i, and p covariates. We may interchangeably refer to units as 'individuals' or 'observations'. Formally, consider a dataframe $D = [X, Y, T]$, including $n \times p$ matrix $X \in \{0, 1, \ldots, k\}^{n \times p}$ where X contains the categorical covariates for all units, $Y \in \mathbb{R}^n$ denotes the outcome vector, and $T \in \{0, 1\}^n$ denotes the treatment indicator vector (1 for treated, 0 for control); $\mathbf{x}_i$ denotes the covariate vector of unit i. We will use $\theta \in \{0, 1\}^p$ to denote the variable selection indicator vector for a subset of covariates to match on. A unit is a triplet (covariate value $\mathbf{x}_i$, observed outcome y_i, treatment indicator t_i). Given dataset $\mathcal{S}$, define the *matched group* for unit i with respect to covariates selected by θ as the units in $\mathcal{S}$ that match i exactly on covariates θ:

$$\texttt{MG}_i(\theta, \mathcal{S}) = \{i^{'} \in \mathcal{S} : \mathbf{x}_{i^{'}} \circ \theta = \mathbf{x}_i \circ \theta\},$$

where $\circ$ denotes Hadamard product. Under the assumption of no unobserved confounding the question of the causal effect of T on Y then becomes which covariates θ we should match unit i on.

In FLAME and DAME, the value of a set of covariates θ is determined by how well these covariates can be used together to predict outcomes. However, we often prefer not to look at the outcomes of our dataset to determine how to match, to avoid risk of biasing the estimates. Thus, we consider a separate training dataset $\mathcal{S}^{tr}$. Let $\mathcal{S}_0^{tr}$ be the subset (of $\mathcal{S}^{tr}$) of control units ((X^{tr}, Y^{tr}) with $T^{tr} = 0$), and let $\mathcal{S}_1^{tr}$ be the subset (of $\mathcal{S}^{tr}$) of treated units ((X^{tr}, Y^{tr}) with $T^{tr} = 1$). The empirical prediction error $\hat{\texttt{PE}}_{\mathcal{F}_{\|\theta\|_0}}$ is defined with respect to a class of functions $\mathcal{F}_k := \{f : \{0, 1\}^k \to [0, 1]\}$ $(1 \leq k \leq d)$ as:

$$\hat{\texttt{PE}}_{\mathcal{F}_{\|\theta\|_0}}(\theta, \mathcal{S}^{tr}) = \min_{f^{(1)} \in \mathcal{F}_{\|\theta\|_0}} \frac{1}{|\mathcal{S}_1^{tr}|} \sum_{(\mathbf{x}_i, y_i) \in \mathcal{S}_1^{tr}} (f^{(1)}(\mathbf{x}_i \circ \theta) - y_i)^2 + \min_{f^{(0)} \in \mathcal{F}_{\|\theta\|_0}} \frac{1}{|\mathcal{S}_0^{tr}|} \sum_{(\mathbf{x}_i, y_i) \in \mathcal{S}_0^{tr}} (f^{(0)}(\mathbf{x}_i \circ \theta) - y_i)^2.$$

That is, $\hat{\texttt{PE}}_{\mathcal{F}_{\|\theta\|_0}}$ is the smallest prediction error we can get on both treatment and control populations using the features specified by θ. Thus, given a matching dataset $\mathcal{S}^{ma}$ and a training dataset $\mathcal{S}^{tr}$, the best selection indicator we could achieve for a nontrivial matched group that contains treatment unit i would be:

$$\theta^*_{i,\mathcal{S}^{ma}} \in \arg\min_{\theta} \hat{\texttt{PE}}_{\mathcal{F}_{\|\theta\|_0}}(\theta, \mathcal{S}^{tr}) \text{ s.t. } \exists \ell \in \texttt{MG}_i(\theta, \mathcal{S}^{ma}) \text{ s.t. } t_\ell = 0. \tag{1}$$

This constraint says that the matched group contains at least one control unit. It also matches on covariates that together can be used to predict well on the training set. The covariates selected by $\theta^*_{i,\mathcal{S}^{ma}}$ are those that predict the outcome best, provided that at least one control unit has the same exact covariate values as i on the covariates selected by $\theta^*_{i,\mathcal{S}^{ma}}$. And, we know that covariates $\theta^*_{i,\mathcal{S}^{ma}}$ together can predict the outcome well. Please note that the predictive error is not the sole determinant of a matched group, and the covariates used in a matched group is determined based on an iterative procedure. This is discussed in more detail below.

The *main matched group* for i is then defined as $\texttt{MG}_i(\theta^*_{i,\mathcal{S}^{ma}}, \mathcal{S}^{ma})$. Users can choose whether units are matched with replacement; that is, whether a previously matched unit can be matched in a subsequent iteration of the algorithm. The first time a unit is matched, that matched set is its *main matched group*, from which its treatment effect estimates are calculated. If units are allowed to be matched with replacement, a unit can become a member of another unit's main matched group. Any additional groups which a unit belongs to other than it's *main matched group* is it's *auxiliary matched group*.

The goal of FLAME and DAME is to calculate the main matched group $\texttt{MG}_i(\theta^*_{i,\mathcal{S}^{ma}}, \mathcal{S}^{ma})$ *for as many units i as possible.* Then, the matched groups can be used to estimate treatment effects.

The implementation of the above mathematical descriptions in both DAME and FLAME algorithms requires us to iterate over two nested loops, shown in Figure 1.

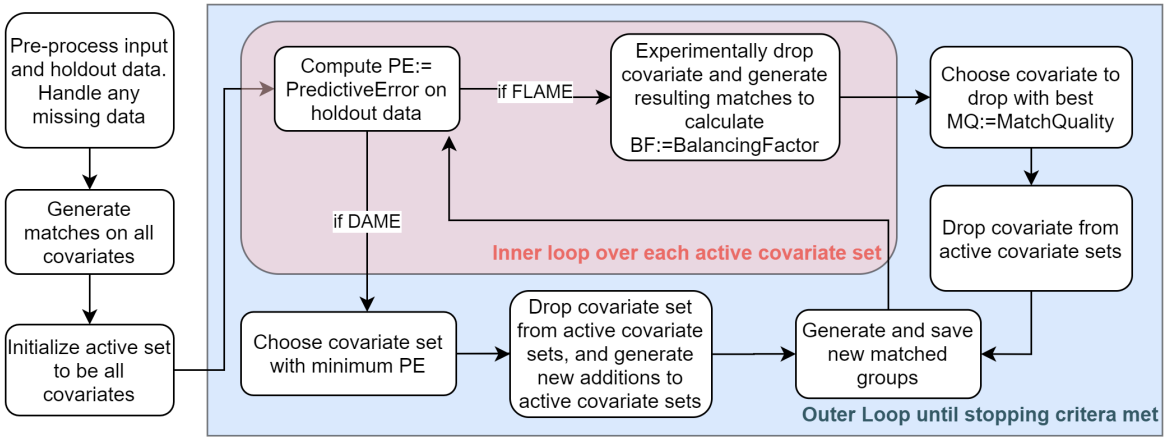

Figure 1: Flow diagram: DAME/FLAME algorithms

Users must begin by providing a dataset with discrete observational covariates, a binary treatment indicator, and a continuous or discrete outcome column. If users to not supply a separate holdout training dataset with the same covariates as the input dataset, they can partition the input dataset to create the holdout training set.

Users have a variety of options for handling *missing covariate data.* They can (1) exclude units with missing values from the procedure, (2) impute the missing data via the Multiple Imputation by Chained Equations (MICE) method (Buuren and Groothuis-Oudshoorn 2010), or (3) specify that matches should not occur on missing values without imputing them. In

this third case, matches can still be made for a unit on its covariates that are not missing.

As described above, after handling missing data, the algorithms begin by matching any units that can be matched exactly on all covariates, where at least one treatment unit as well as at least one control unit are contained in each matched group. The algorithms then execute the outer loop: updating sets of covariates to match on, referred to as *active covariate sets*, until a stopping criterion is reached. In each iteration, the algorithms execute the inner loop, examining each covariate set to select the best one to match on. Units that have identical values for all of the covariates that are part of the chosen covariate set form a matched group, as long as at least one treatment unit and at least one control unit are present in the group.

To determine the best covariate set, FLAME selects the covariates yielding the highest *match quality* `MQ`, defined as $\texttt{MQ} = \texttt{C} \cdot \texttt{BF} - \texttt{PE}$, where `C` is a user-specified hyperparameter. DAME selects the covariate set minimizing `PE`. Here, `PE` denotes the *predictive error* as described above. The *balancing factor*, `BF`, measures the proportion of treatment and control units that are matched on a covariate set. DAME iterates efficiently over covariate sets, prioritizing matching on large covariate sets if they can be used to effectively predict the outcome on the holdout training set. FLAME approximates this solution for scalability: it consistently matches on a smaller set of covariates than in the previous iteration, while still ensuring that each covariate set can be used to effectively predict the outcome on the holdout training set.[1]

The outer loop has a number of possible stopping criteria. It must stop when all units are placed in matched groups or all covariate sets have been dropped. Additionally, users can enforce stopping based off other criteria, e.g., (1) when there are too few unmatched (treatment or control) units, (2) after a certain number of iterations, (3) when predictive error rises too much, or (4) when the balancing factor for a given round is not high enough. If the third criteria is chosen (when predictive error rises too much), then any units that are matched will always be matched on a set of covariates that together can be used to predict the outcome well.

1.3. Installation, Setup, and Testing

The package is designed for Python 3.6 and above. **dame-flame** depends on **scikit-learn** version 0.23.2 and above, **pandas** version 0.11.0 and above, and **numpy** version 1.16.5.

dame-flame is available for download on PyPi and on GitHub[2]. The public documentation website[3] covers the API, installation instructions, a quick-start tutorial, several examples, and a contributing guide. In harmony with the best open source practices, users are invited to report any unexpected bugs, assist with cleaning or maintain code, add details or use-cases to the documentation, and add more test cases. They are welcome to do so via the GitHub bug reporting and pull requesting features, or by directly emailing the core development and research team.

There is also an accompanying R package for the FLAME and DAME algorithms, that can also be found on GitHub[4]. The R package is also an open source package welcoming user

[1] It is important to note that MQ can be replaced by any other measure of quality that a user might be interested in (eg propensity score, human in the loop machine learning evaluation).

[2] Code is publicly available at:
https://github.com/almost-matching-exactly/DAME-FLAME-Python-Package/.

[3] https://almost-matching-exactly.github.io/DAME-FLAME-Python-Package/

[4] https://github.com/almost-matching-exactly/R-FLAME

questions and contributions.

Testing was done to ensure that **R-FLAME** and the **dame-flame** Python package yield consistent results on a range of datasets and parameter options.

For further reliability testing, **dame-flame** offers Continuous Integration through Travis-CI, and the independent Coveralls API was used to verify the test suite offers an extensive code coverage.

2. Code and its explanation

2.1. Class Structure and Advanced Features

The API consists of standard Pythonic design established by **scikit-learn**. The main feature of the API is the class `matching.MatchingParent`, and two subclasses `DAME` and `FLAME`. These offer standard methods `fit`, where a user provides a holdout training dataset, and `predict`, where a user provides a matching dataset, and the matches of the algorithm are computed.

Continuing to follow **scikit-learn**'s standards, any post-processing is done using `utils` functions, which take as arguments a `matching.DAME` or `matching.FLAME` object, and use these to compute matched groups of units, and estimate treatment effects, including the *average treatment effect* (ATE) for the population, and *conditional average treatment effect* (CATE) of a selected unit. These are mathematically defined in section 2.2.

In the rest of this section, we proceed to discuss advanced features for computing predictive error, early stopping features, missing data handling options, and FLAME specific options. For each of these advanced features, we discuss the theory, list the specific parameter for users to select, and the package default for this feature. We conclude this section by comparing these features to features of other popular matching packages.

2.2. Definitions of Estimands and Estimators

We continue with notation previously introduced: Units are indexed by i, which ranges from 1 to N. We may interchangeably refer to units as 'individuals' or 'observations'. Also, note that all units which contribute to treatment effect estimates must have been matched.

There are p pre-treatment covariates $x_1, \ldots, x_p$ and for a given unit i, we will refer to its vector of covariates as X_i. Let the binary treatment indicator for unit i be denoted by T_i. We let Y_i be the observed outcome for individual i where $Y_i = Y_i(1)T_i + Y_i(0)(1 - T_i)$ and $Y_i(0), Y_i(1)$ are the potential outcomes of unit i under control and treatment, respectively. Lastly, we introduce notation for matched groups, which we index by m, which ranges from 1 to M. The size of a matched group m is $\|m\|$, which is the number of units in the group.

The conditional average treatment effect, or CATE, is defined as the average treatment effect conditional on particular covariates. Formally, given a set of covariates X_i, the CATE is:

$$\text{CATE}(X_i) = \frac{1}{N} \sum_{i=1}^{N} \mathbb{E}[Y(1) - Y(0) \| X_i].$$

Our implementation of CATE estimation allows users to input a unit i and receive its CATE estimate, based on its main matched group.

Since our units are matched almost-exactly, all units sharing the same main matched group will have the same CATE estimate. For a unit i whose main matched group m is of size $\|m\|$, we estimate its CATE as:

$$\frac{1}{\|m\|}\sum_{i:T_i=1}[\hat{Y}_i(1)] - \frac{1}{\|m\|}\sum_{i:T_i=0}[\hat{Y}_i(0)]$$

where $\hat{Y}_i(0), \hat{Y}_i(1)$ are calculated using the units in m with treatment $1 - T_i$.

The Average Treatment Effect (ATE) is unconditional on X: ATE $= \mathbb{E}[Y(1) - Y(0)]$. We offer two estimators of the ATE. For one, we will create a weighted combination of CATE estimates from the various matched groups. Let q_i denote the number of matched groups that unit i appears in. Note this quantity can be greater than 1 when matching is done with replacement, via the 'Repeats = True' argument. We then define the weight of a matched group m as $w_m = \sum_{i=1}^{\|m\|}\frac{1}{q_i}$. We denote the CATE estimate of group m as CATE_m. We estimate the ATE as:

$$\text{ATE} = \frac{\sum_m \text{CATE}_m \times w_m}{\sum_m w_m}.$$

Note that this expression downweights units that were matched many times so that they do not dominate the ATE estimate.

The other ATE estimator implemented is the simple matching estimator described in Abadie, Drukker, Herr, and Imbens (2004). We also offer the variance estimator of equation 4 in (Abadie *et al.* 2004). Please note that this estimator assumes constant treatment effects and homoscedasticity. It is not asymptotically normal, and the standard implications this has on confidence intervals or hypothesis tests apply.

2.3. Predictive Error

As discussed in Section 1, the algorithm's decision of best covariates to match on relies on a computation of predictive error, or PE, based on a user-chosen machine learning algorithm run on the holdout dataset. The **dame-flame** Python package offers different options for the machine learning algorithm used, as well as a simplified FLAME and simplified DAME that does not use machine learning, but instead allows the users to input feature importance information for matching. We use **scikit-learn** for the underlying learning algorithms, and refer the reader to their documentation and references to learn more about these popular machine learning algorithms, as well as their specific implementations, applied separately to the treated and control units in the holdout set.

Users can easily modify the code to feature a new machine learning algorithm of their choice. The options we provide at this time for the computation of PE include the following.

- Ridge Regression. Ridge regression minimizes a residual sum of squares plus a regularization term measuring the ℓ_2 norm of the coefficient vector, multiplied by a shrinkage parameter, α. For this option, a larger α should be chosen if it is believed that there is greater multicollinearity in the data, meaning that many covariates are linearly correlated. This option can be chosen using the parameter `adaptive_weights='ridge'`. The α parameter can also be adjusted using parameter `alpha` when declaring a matching object.

- Cross-Validated Ridge Regression. This is a ridge regression with built-in cross validation to determine the best α parameter. We use the **scikit-learn** `ridgeCV` class, but the default array of α options that we provide the function to iterate over is larger than the default they provide, for greater flexibility. This option is advantageous over the 'ridge' option without cross validation in the case when a user is uncertain about the α parameter, and a minor speed decrease owing to cross validation is acceptable. This option can be chosen using the parameter `adaptive_weights='ridgeCV'`.
- Decision Tree. Designed on a variation of the CART algorithm, this is the only option that can be used for unordered discrete data. This option can be chosen using the parameter `adaptive_weights='decision-tree'` as described in section 4.

The option a user chooses can be selected using the specified value for the parameter `adaptive_weights` when declaring a matching object, as shown in examples in Section 3. If, instead of allowing the algorithm to select covariates via the PE parameter, the user prefers to pre-specify covariate importance, they can do so by specifying `adaptive_weights = False`. The weights to the covariates in `input_data` can be specified using the parameter `weight_array` in the 'fit' function. The values in that array must sum to 1.

2.4. Early-stopping Options

The FLAME and DAME algorithms will stop after running to completion, or based on user-defined early stopping criteria. The default option is that the algorithm runs until all units are matched, or until there is a large spike in predictive error. If runtime or high accuracy of estimates of treatment effects are important, then we recommend users experiment with their stopping criteria based on their specific needs and dataset size. A large dataset will have a longer runtime, and an early stop will take less time. Without early stopping, the matches could degrade in quality in later iterations, where units that are farther from each other in covariate space would now be matched, leading to worse overall performance of the method.

Below, we define and discuss the early stopping criteria that users can choose. All criteria are controlled via a parameter to the classes defined in Appendix 4.

- The maximum number of iterations of the FLAME or DAME algorithm, via the parameter `early_stop_iterations`. If FLAME is used, then this is the maximum number of covariates that can be dropped, meaning when the total number of covariates is m, no unit will be matched on fewer than $m-$`early_stop_iterations` covariates. This is useful when the user wants only matches of a specific high level of quality, or when the user is concerned about computational time.
- Unmatched units in treatment or control, via the parameters `stop_unmatched_c` and `stop_unmatched_t`. When the algorithm is set with the `repeats=True` parameter, then previously matched units (that is, units whose main matched groups have already been determined) can still be placed in the main matched groups of other units. The algorithm will by default stop iterating when there are no more units that have not been placed in any group.
- Proportion of unmatched units, via the parameters `early_stop_un_c_frac`, and `early_stop_un_t_frac`. This stops the algorithm when the fraction of control units

or treatment units are unmatched goes below a user-defined value. One specific case in which this could be useful is where a user thinks that some percent of the input is unlikely to result in good matches.

- Predictive error, via the parameters `early_stop_pe`, and `early_stop_pe_frac`. The predictive error measures how important a covariate set is for predicting the outcome on the holdout training dataset, using a machine learning algorithm. It is the sole determiner of the covariate set to match on for DAME, and one of two factors for FLAME. The stopping criterion's setting would be determined as follows: stop when the best set of covariates increases the predictive error by more than 1 minus this value above the previous iteration.

2.5. Missing Data Handling Options

Users are offered a variety of options for handling missing covariate data. Imputing missing values in datasets is possible, but matches become less interpretable when matching on imputed values, in that it is more difficult to discern why a match was recommended by the matching algorithm. Here, we discuss the options we provide in detail and make recommendations. The parameter to select in the **dame-flame** Python package is mentioned here, and more details on usage is provided in Section 4.

There can be missing data in either the input matching data, the holdout training data, or both. The specific character that is used to denote missing value can be selected via the parameter `missing_indicator`, which can be a character, integer, or numpy.nan.

For the input dataset, three options exist:

- Omit units with missing values. We recommend using this if missing values indicate data fidelity issues in a unit. The algorithms handle this by ensuring that units in the input dataset that have missing data are dropped from the dataset prior to running the algorithms finding the matches. This option is selected via the parameter `missing_data_replace=1`.
- Match units with missing values, but ignore missing values when considering which units to match to. We recommend this for the majority of cases. The underlying algorithm will handle this when pre-processing the input. This option is selected via the parameter `missing_data_replace=2`.
- Impute missing values with MICE. This is computationally costly and would reduce the interpretability of the matches. The algorithm would create several imputed datasets and iterate over each to find a match according to each dataset. This option is selected via the parameter
  `missing_data_replace=3`. The number of MICE imputations is selected via the parameter `missing_data_imputations`.

For the holdout dataset, the following two options exist:

- Omit units that have any missing values. We recommend this option only if a missing completely at random assumption is tenable in both holdout and matching datasets (Little 1988). In the underlying algorithm, units in the holdout dataset that have missing

data are dropped from the dataset prior to running the DAME or FLAME algorithm to find the matches. This option is chosen by using the parameter `missing_holdout_replace=1`.

- Impute missing values with MICE. In the underlying algorithm, we begin by running MICE to create several imputed training holdout datasets. The DAME or FLAME algorithm is run once, and the best covariate set is chosen based on the predictive error over all imputed datasets. This option is chosen by using the parameter `missing_holdout_replace=2`.

The underlying MICE implementation is done using scikit learn's experimental IterativeImpute package, and relies on DecisionTreeRegressions in the imputation process, to ensure that the data generated is fit for unordered categorical data.

2.6. Additional Parameters Available

As discussed in Section 2, users can adjust whether they match units with or without replacement. This is controlled via the boolean parameter `repeats`.

Output style can also be controlled by the user, via a range of parameters. All of these parameters are used when declaring a matching object.

- The parameter `verbose`. This is a number that will range from 0 to 3 and higher numbers result in additional information being output. If true, the output of the algorithm will include the predictive error of the covariate sets used for matching in each iteration.
- The boolean parameter `want_bf`. If true, the output will include the balancing factor for each iteration.
- The boolean parameter `want_bf`. If true, the output will include the balancing factor for each iteration.

There are two FLAME specific parameters, which users would provide in the final 'fit' step of the algorithm. These are:

- `C`, type float. This is the tradeoff parameter between the balancing factor and the predictive error when deciding which covariates to match on.
- `pre_dame`, type bool, integer, default=False. If an integer is provided, this is the number of iterations to run the FLAME algorithm for before switching to DAME, in order for a hybrid FLAME-DAME option.

2.7. Comparison to other Matching Packages

Many other matching methods either produce low-quality matches(leading to potentially poor treatment effect estimates), uninterpretable matches (e.g., in which matches include units with highly dissimilar covariates values), or matches that are manually defined by an analyst. One of the most widely used algorithms is nearest neighbor propensity score matching, provided by the R package **MatchIt** (Stuart, King, Imai, and Ho 2011). Propensity score matching

Language: Package	built-in treatment-effect estimations	missing data handling options	provide matched groups
Python: **dame-flame**	Average, Conditional	✓	✓
Python: **DoWhy**	Average		
Python: **PyMatch**			✓
R: **cem**	Average	✓	✓
R: **MatchIt** Propensity Score	Average	✓	✓

Table 1: **Features of Matching Packages**

reduces units' covariate information to one dimension, allowing matches to contain units even at extreme ends of the covariate space; such matches are uninterpretable. MatchIt allows other matching metrics, such as Mahalanobis distance, but does not allow for learning the proper metric as FLAME and DAME.

Another common matching algorithm is Coarsened Exact Matching (CEM), popularly available in the R package **cem** (Iacus, King, and Porro 2009). CEM requires the user to manually coarsen variables, requiring humans to know detailed information about a high-dimensional space in advance,a task at which humans are not naturally adept (Dieng *et al.* 2019). Coarsening covariatess via default histogram binning methods fails to take into consideration their impacts on treatment and outcome, resulting in poor matches. Instead of requiring a human to manually input how matches should be constructed (or to use histogram binning), FLAME and DAME use machine learning on a training set to determine this information.

Many of the features described in Section 2.4 and Section 1.1 are unavailable in other matching packages. Table 1 compares the characteristics of **dame-flame** against popular alternatives. Most matching packages are implemented in R. R's **cem** package only supports ATT treatment effects (Iacus *et al.* 2009). The **MatchIt** package focuses on estimation of average treatment effects and not conditional average treatment effects, both of which are handled in the same coherent manner by **dame-flame**. Users of any propensity score matching algorithm can adjust matched group sizes only by entering a ratio of treatment to control units, forcing all matched groups to be of the same size. Python's **PyMatch** and **DoWhy** offer propensity score matching, but **DoWhy** does not emphasize matched groups, favoring to present treatment effects (ATT, ATE, and ATC), and other output (Sharma, Kiciman *et al.* 2019; Miroglio *et al.* 2017). R's **cem** package is a good choice for datasets with multi-level, non-binary treatment variables, whereas the current version of **dame-flame** does not yet offer a multi-level treatment solution.

A further advantage of **dame-flame** is the higher quality of the matched groups generated by DAME and FLAME relative to propensity score matching, as shown by Dieng *et al.* (2019).

A drawback of **dame-flame** is the requirement that covariates be discrete. The packages **DoWhy**, **PyMatch**, **cem** and **MatchIt** do allow users to use continuous covariates without any pre-processing steps or manual binning. Although a user could manually bin continuous covariate values prior to using **dame-flame**, we do not recommend this asides scenarios in which users are confident they are binning variables in a way that is meaningful for their research. A user interested in a matching package that does allow for continuous covariates that is still in the Almost Matching Exactly framework may consider exploring **R-MALTS** or **pymalts**. These packages implement the algorithm Matching After Learning To Stretch (MALTS), which will use exact matching for discrete variables, and will learn Mahalanobis distances for continuous variables. Instead of a predetermined distance metric like **MatchIt**, MALTS gives covariates that contribute more towards predicting the outcome higher weights

(Parikh, Rudin, and Volfovsky 2018).

3. Examples

3.1. Basic Example

Here we offer an example to illustrate API usage, using a simple, small, 4 unit and 4 covariate simulated dataset to illustrate matched groups easily. An example focused on analysis using a real dataset and its corresponding replication is discussed in Section 3.2. All classes, functions, and parameters used here, as well as additional options for parameters are defined and discussed in Section 4.

The first step is importing the package. We show the dataframe used here as well. The Pandas dataframe places units in rows and covariates in columns, and requires a column with a boolean variable indicating treatment, and a column for the outcome variable.

```
import pandas as pd
import dame_flame

df = pd.DataFrame([[0,1,1,1,0,5.1], [0,0,1,0,0,5.11], [1,0,1,1,1,6.5],
        [1,1,1,1,1,6.]],
        columns=["x1", "x2", "x3", "x4", "treated", "outcome"])

print(df.head())
```

```
  x1  x2  x3  x4  treated      outcome
0  0   1   1   1           0        5.10
1  0   0   1   0           0        5.11
2  1   0   1   1           1        6.50
3  1   1   1   1           1        6.0
```

The first step in the matching procedure is instantiating a 'matching' object, with optional parameters that can specify the early stopping criteria, missing data handling methodology, output style, or the machine learning method used to compute the predictive error. All optional parameters are described in more detail in section 4. Here, we choose the default options, which includes no missing data handling, no early stopping procedures, and computes predictive error with ridge regression. We choose these options because this dataset does not have any missing data that needs to be handled and because it is a small example, we do not need to stop the algorithm early.

```
model = dame_flame.matching.DAME()
```

The next step is to call the 'fit' method on the 'matching' object created above. Here, users must provide a file location of the holdout training dataset, a Pandas dataframe, or a fraction of the input dataset to use for matching, in the parameter `holdout_data`.

Additionally, the name of the treatment column and the name of the outcome column can be provided.

```
model.fit(df, "treated", "outcome")
```

At this point, simply calling the 'predict' method with the input dataset produces matched results. The return value from the `predict` command contains an output table, which consists of the units that were matched to at least one other unit. For each unit that was matched, the table indicates which of the covariates were used for matching, and the covariate values that each unit was matched on. The covariates that were not used to match the unit are denoted with "*" as their value.

```
result = model.predict(df)
print(result)
```

```
   x1   x2   x3   x4
0   *   1    1    1
1   *   0    1    *
2   *   0    1    *
3   *   1    1    1
```

Various result summaries are available, including a printout of all matched groups, and the units belonging to each group. The result of the `predict` function, shown above, can also be retrieved by using the following attribute `df_units_and_covars_matched` of the matching class. The `units_per_group` attribute of the matching class provides an array of arrays. Each sub-array is a matched group, and each item in each sub-array is an int, indicating the unit in that matched group. If matching is done with the parameter `repeats=False` when defining the matching class, then no unit will appear more than once. If `repeats=True` then the first group in which a unit appears is its main matched group.

```
print(model.units_per_group)
```

```
[[0, 3], [1,2]]
```

This shows us that unit 0 and unit 3 are in a matched group, and that unit 1 and unit 2 are in another matched group.

The `utils` functions offer post-processing. In these functions, users must pass as parameters the matching object declared earlier, and for many of the functions, users must pass in a `unit_ids` parameter, which can be a single unit or a list of unit ids.

The function that provides matched groups of each unit is `MG`. If one unit id was provided, this is a single dataframe containing the main matched group of the unit. If the unit does not have a match, the return will be np.nan. If multiple unit ids were provided, this will be a list of dataframes with the main matched group of each unit provided.

```
mmg = dame_flame.utils.post_processing.MG(matching_object=model, unit_ids=0)
print(mmg)
```

```
     x1    x2    x3    x4    treated    outcome
0    *     1     1     1     0          5.1
3    *     1     1     1     1          6.
```

This shows the main matched group of unit 0 is unit 3, and that covariates that unit 0 and unit 3 matched on are covariates 'x2', 'x3', and 'x4'.

The functions in the `utils` library also include treatment effect estimators, as defined in section 2.2, including an estimate for CATE. If one unit id was provided, the return value will be a single float representing the conditional average treatment effect estimate of the unit. This is equal to the CATE of the group that the unit is in. If the unit does not have a match, the return will be np.nan. If multiple unit ids were provided, the return value will be a list of floats with the CATE estimate of each unit provided.

```
cate = dame_flame.utils.post_processing.CATE(matching_object=model, unit_ids=0)
print(f'{cate:.6f}')
```

```
0.900000
```

The ATE function, to get the Average Treatment Effect estimator only requires a matching object, but does not require a unit id, and returns a float.

```
ate = dame_flame.utils.post_processing.ATE(matching_object=model)
print(f'{ate:.6f}')
```

```
1.145000
```

As discussed in Section 2.2, the package also offers a second ATE estimator with a corresponding variance estimator. Again, the required parameter is the matching object used earlier.

```
ate, var = dame_flame.utils.post_processing.var_ATE(matching_object=model)
print(ate)
print(var)
```

```
1.145000
0.030012
```

As is expected, we see that this ATE estimate is the same as or close to the ATE estimate from the other ATE function.

3.2. Example Analysis

dame-flame is an interpretable matching package because it allows users to quickly and easily understand which covariates were selected to be important for causal inference. This can be useful for practitioners in determining who benefits from treatment the most and where

resources should be spent for future treatment. It also allows users to view various other aspects of the matching process such as the stopping criteria as they use the package.

Here we demonstrate an experimental use-case for the DAME algorithm on the Tennessee's Student Teachers Achievement Ratio (STAR) Dataset. This dataset originates from an experiment beginning in 1985, in which elementary school students and their teachers across 79 schools in Tennessee were randomly assigned to classes of small or regular sizes from Kindergarten through 3rd grade (Achilles, Bain, Bellott, Boyd-Zaharias, Finn, Folger, Johnston, and Word 2008). The results showed that a small class size attendance leads to higher standardized test performance, and long run benefits in terms of increased college entrance exam taking, especially among minority students (Krueger and Whitmore 2001).

We aim to examine the aggregate effect of small kindergarten class sizes on a student's kindergarten reading scores, so we limit to the experimental dataset in which treatment was random. Our cleaned dataset has around 5000 students with reading scores ranging from 315 to 627. Our covariates include children's characteristics, teacher's characteristics and school characteristics. The children's characteristics are gender, race (binary, with White and Asian in one group, and all other races in the other group), free lunch status, and age in months (binned into deciles). The teacher characteristics include race, gender, and having a higher degree than bachelors. The school characteristics are urbanicity (rural, urban, suburban, and inner city) and a school identification number, with one for each of the 79 schools.

We randomly selected 80% of the dataset for matching and 20% for the holdout training dataset.

First, we ensure that the analysis on this dataset is appropriate with DAME or FLAME matching by ensuring that there is a lack of sensitivity to the train/test split, that the algorithm matches a sufficient number of units, and that the aggregate treatment effect estimates are reasonable. So, we run four different trials of DAME on random splits. We will declare an object of the matching class using the early stopping criteria of stopping when there is a sharp rise in `PE`, and run the `.fit` and `.predict` functions on the matching class to run the match

```
models = [] # We store the matching objects to compute ATE with later
random_seeds = [1111, 2222, 3333, 4444]
for i in range(len(random_seeds)):
    matching_df, holdout_df = train_test_split(df_trunc, test_size=0.2,
        random_state=random_seeds[i])
    model_dame = dame_flame.matching.DAME(repeats=False, verbose=0,
        adaptive_weights='decisiontree',
        early_stop_pe=True)
    model_dame.fit(holdout_data=holdout_df, outcome_column_name='gktreadss')
    model_dame.predict(matching_df)
    models.append(model_dame)
```

```
3004 units matched. We finished with no more treated units to match
2947 units matched. We finished with no more treated units to match
2891 units matched. We finished with no more treated units to match
2894 units matched. We finished with no more treated units to match
```

It is a good sign that in each of the random test/train partitions, they all matched to completion, and that some did not stop matching because of a jump in `PE`. This indicates that computation of `PE` is not sensitive to which particular units are in the matching or holdout training set.

Next, we compute the ATE estimate on each of the trials. We do this by iterating over the matching class objects declared above, and calling the `utils` function `var_ATE`, which is defined further in Section 4.

```
for model in models:
    ate, var = dame_flame.utils.post_processing.var_ATE(matching_object=model)
    print("ATE of trial", i, ":", ate,". Variance: ", var)
```

```
Run 0 ATE: 5.998078614036495 & Variance of ATE: 1.3608548636512854
Run 1 ATE: 4.848867498485764 & Variance of ATE: 1.431203792460095
Run 2 ATE: 5.7532774126599735 & Variance of ATE: 1.4481064663464616
Run 3 ATE: 5.5627909687256585 & Variance of ATE: 1.405629905925439
```

Note that the ATE estimate was approximately the same in each trial, and that the variance of the ATE estimate is small each trial. This indicates that the experiment placing students in smaller class sizes caused those students to achieve a higher kindergarten reading test score by a few points. We conclude this is a reasonable ATE estimate because in the analysis done in Krueger and Whitmore (2001), the estimate for the impact of small class sizes provided by Figure 1 in their work, which is based on a linear regression on many similar covariates, is between 5 and 6 for average percentile of math and reading scores.

Next, we consider whether DAME or FLAME is a better matching method for this dataset. We run four trials of FLAME with the same random test/train split. Again, we define an object of the matching class, this time of the FLAME subclass, and we run the `fit` and `predict` functions.

```
flame_models = [] # Again, save matching class objects for later analysis
random_seeds = [1111, 2222, 3333, 4444]
for i in range(len(random_seeds)):
    matching_df, holdout_df = train_test_split(df_trunc, test_size=0.2,
        random_state=random_seeds[i])
    model_flame = dame_flame.matching.FLAME(repeats=False, verbose=3,
        adaptive_weights='decisiontree', missing_holdout_replace=1,
        missing_data_replace=1, early_stop_pe=True)
    model_flame.fit(holdout_data=holdout_df,outcome_column_name='gktreadss')
    model_flame.predict(matching_df)
    flame_models.append(model_flame)
```

We omit the output of this matching result for space, however, it is available with the full replication on our github. We observe that the FLAME algorithm matches on fewer units when using the stopping criteria `early_stop_pe`, whereas DAME is able to match all treated units before stopping the matching. When we use the `var_ATE` function on the FLAME matching objects as we did above with the DAME matching objects, we observe similar ATE

Trial 1	Trial 2	Trial 3	Trial 4
All covariates, 944	All covariates, 950	All covariates, 954	All covariates, 982
Teacher gender, 0	Teacher gender, 0	Teacher gender, 0	Teacher gender, 0
Urbanicity, 0	Urbanicity, 0	Urbanicity, 0	Urbanicity, 0
Student race, 38	Teacher higher degree, 484	Teacher race, 145	Student race, 40
Teacher race, 210	Student race, 559	Student race, 212	Teacher higher degree, 524
Teacher higher degree, 779	Teacher race, 837	Teacher higher degree, 822	Teacher race, 754
Student free lunch, 1155	Student free lunch, 1178	Student free lunch, 1128	Student free lunch, 1135
Student gender, 1514	Student gender, 1569	Student gender, 1479	

Table 2: Covariate Dropping in order from FLAME. Each entry is a iteration of the algorithm, listing the covariate dropped in that iteration, and the number of units matched after that covariate is dropped

estimates as DAME. The tradeoff between number of unit matched and time duration for matching alogrithm to run can be considered when practitioners choose between DAME or FLAME.

We summarize the dropping criteria from the full verbose output of FLAME, generated using the parameter `verbose=3` in Table 2. The fact that teacher gender is dropped first in each trial makes sense, since all teachers are female in the cleaned data set. Having relatively few matches prior to removing teachers race and teachers gender makes sense, as in Krueger and Whitmore (2001), these covariates were not used in their linear regression estimating the impact of small class sizes on percentile test scores.

We plot the log transform of CATEs of the matched groups from DAME against the number of units of each group in Figure 2. We omit the code to create this plot as it highlights Python semantics, not specific to **dame-flame**. The code is also available on the GitHub repository. This plot highlights heterogeneity in the treatment effects of groups. Note that the most extreme CATE values are only observed in small matched groups where estimation is expected to be unreliable.

Across the trials, when we examine the matched groups that correspond to the largest group sizes, or the rightmost points on the graphs, we notice that the large matched group contains different units in each trial, so it is sensitive to the test/train split. Additionally, when we examine the matched groups of the youngest minority male students, we find the CATE estimate is typically positive, and is notably small but positive in the largest matched group matching these students. That group has eight students and the matched group matched on all covariates asides from teacher's higher degree and age of student.

4. API Documentation

In this section, we provide details on the matching class definitions and function parameters. The API consists of standard Pythonic design established by scikit-learn. To begin matching, the user declares an object of type `DAME` or type `FLAME`. Both of these inherit from the base class `MatchingParent`. Following this, the user calls the method `fit`, providing a holdout training dataset, and finally `predict`, where a user provides a matching dataset, and the matches of the algorithm are computed.

The user provides the input data as a data frame or file in the function `predict`, which must contain an outcome column, and a treatment column. FLAME and DAME by default

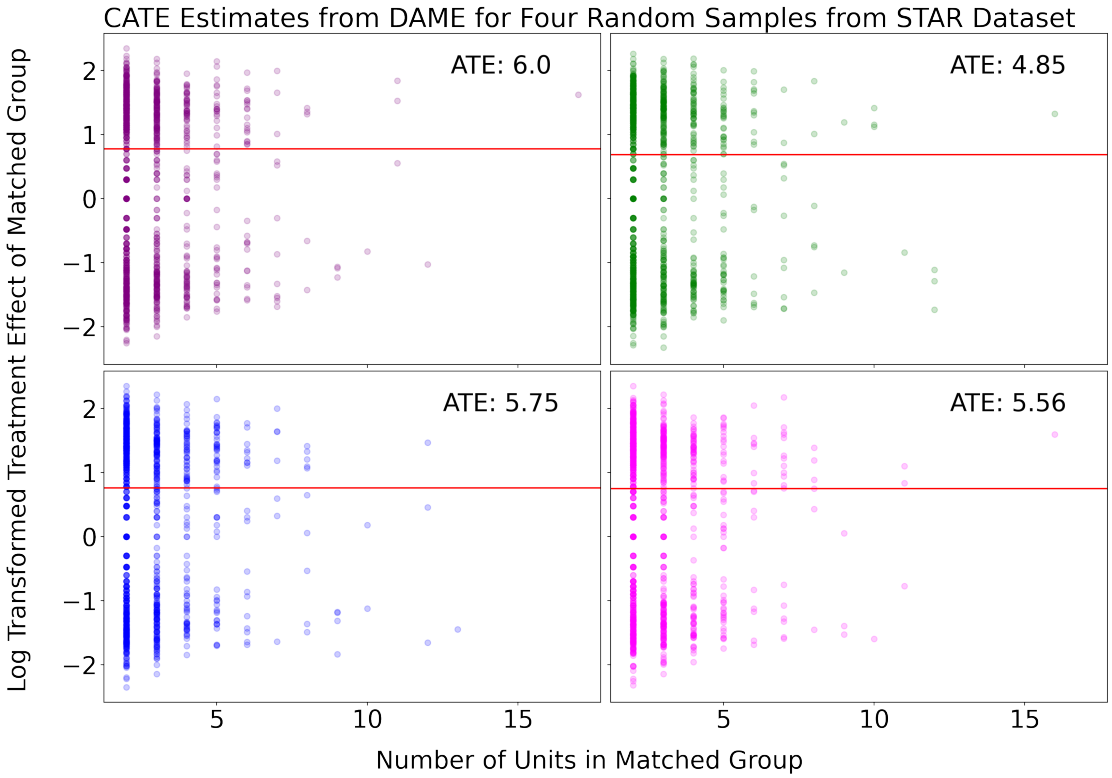

Figure 2: Graph of CATE of matched groups. Plots were done over four different random splits of the dataset into matching and holdout training datasets.

produce an output table consisting of the units that were matched to at least one other unit. For each unit that was matched, the table indicates which of the covariates were used for matching, and the covariate values that each unit was matched on. The covariates that were not used to match the unit are denoted with "*" as their values.

Class Definition

```
dame_flame.matching.DAME(adaptive_weights='ridge', alpha=0.1,
        repeats=True, verbose=2, early_stop_iterations=False,
        stop_unmatched_c=False, early_stop_un_c_frac=False,
        stop_unmatched_t=False, early_stop_un_t_frac=False,
        early_stop_pe=False, early_stop_pe_frac=0.01,
        missing_indicator=np.nan, missing_data_replace=0,
        missing_holdout_replace=0, missing_holdout_imputations=10,
        missing_data_imputations=1, want_pe=False, want_bf=False)

dame_flame.matching.FLAME(adaptive_weights='ridge', alpha=0.1,
        repeats=True, verbose=2, early_stop_iterations=False,
```

```
stop_unmatched_c=False, early_stop_un_c_frac=False,
stop_unmatched_t=False, early_stop_un_t_frac=False,
early_stop_pe=False, early_stop_pe_frac=0.01,
missing_indicator=np.nan, missing_data_replace=0,
missing_holdout_replace=0, missing_holdout_imputations=10,
missing_data_imputations=1, want_pe=False, want_bf=False)
```

Arguments:

- `adaptive_weights`, type {bool, 'ridge', 'decisiontree', 'ridgeCV', 'decisiontreeCV'}, default='ridge'. The method used to decide what covariate set should be dropped next.
- `alpha`, type float, default=0.1. If `adaptive_weights` is set to ridge, this is the alpha for ridge regression.
- `repeats`, type bool, default=True. Whether or not units for whom a main matched has been found can be used again, and placed in an auxiliary matched group.
- `verbose`, type int: 0,1,2,3, default=2. Style of printout while algorithm runs. If 0, no output. If 1, provides iteration number. If 2, provides iteration number and additional information on the progress of the matching at every 10th iteration. If 3, provides iteration number and additional information on the progress of the matching at every iteration.
- `early_stop_iterations`, type int, default=0. If provided, a number of iterations after which to hard stop the algorithm.
- `stop_unmatched_c`, type bool, default=False. If True, then the algorithm terminates when there are no more control units to match.
- `stop_unmatched_t`, type bool, default=False. If True, then the algorithm terminates when there are no more treatment units to match.
- `early_stop_un_c_frac`, type float, default=0.1. Must be between 0.0 and 1.0. This provides a fraction of unmatched control units. When the threshold is met, the algorithm will stop iterating. For example, using an input dataset with 100 control units, the algorithm will stop when 10 control units are unmatched and 90 are matched (or earlier, depending on other stopping conditions).
- `early_stop_un_t_frac`, type float, default=0.1. Must be between 0.0 and 1.0. This provides a fraction of unmatched treatment units. When the threshold is met, the algorithm will stop iterating. For example, using an input dataset with 100 treatment units, the algorithm will stop when 10 control units are unmatched and 90 are matched (or earlier, depending on other stopping conditions).
- `early_stop_pe`, type bool, default=True. If this is true, then if the covariate set chosen for matching has a change in predictive error between two iterations that is higher than 1-`early_stop_pe_frac`, the algorithm will stop.

- `early_stop_pe_frac`, type float, default=0.05. If `early_stop_pe` is true, then this determines stopping criteria based on whether the covariate set chosen for matching has a change in predictive error between two iterations that is higher than 1-`early_stop_pe_frac`.
- `want_pe`, type bool, default=False. If true, the output of the algorithm will include the predictive error of the covariate sets used for matching in each iteration.
- `want_bf`, type bool, default=False. If true, the output will include the balancing factor for each iteration.
- `missing_indicator`, type character, integer, numpy.nan, default=numpy.nan. This is the indicator for missing data in the dataset.
- `missing_holdout_replace`, type int: 0,1,2, default=0. If 0, assume no missing holdout training data and proceed. If 1, the algorithm excludes units with missing values from the holdout dataset. If 2, do MICE on holdout dataset. If this option is selected, it will be done for a number of iterations equal to `missing_holdout_imputations`.
- `missing_data_replace`, type int: 0,1,2,3, default=0. If 0, assume no missing data in matching data and proceed. If 1, the algorithm does not match on units that have missing values. If 2, prevent all `missing_indicator` values from being matched on. If 3, do MICE on matching dataset. This is not recommended. If this option is selected, it will be done for a number of iterations equal to `missing_data_imputations`.
- `missing_holdout_imputations`, type int, default=10. If `missing_holdout_replace=2`, this is the number of imputations.
- `missing_data_imputations`, type int, default=1. If `missing_data_replace=3`, this is the number of imputations.

`fit` function

```
fit(self, holdout_data=False, treatment_column_name=`treated',
    outcome_column_name=`outcome', weight_array=False))
```

Arguments:

- `holdout_data`, type string, dataframe, float, False , default=False. This is the holdout training dataset. If a string is given, that should be the location of a CSV file to input. If a float between 0.0 and 1.0 is given, that corresponds the percent of the input dataset to randomly select for holdout data. If False, the holdout data is equal to the entire input data. If users choose to use units repeatedly in both the holdout and training dataset, they should be careful that the data do not have a special situation that need to be respected in subsampling such as a hierarchy.
- `treatment_column_name`, type string, default="treated". This is the name of the column with a binary indicator for whether a row is a treatment or control unit.

- `outcome_column_name`, type string, default="outcome".This is the name of the column with the outcome variable of each unit.
- `weight_array`, type array, optional. If `adaptive_weights` = False, these are the weights to the covariates in `input_data`, for the non adaptive version of DAME. Must sum to 1. In this case, we do not use machine learning for the weights, they are manually entered as `weight_array`.

`predict` **function**

`predict(self, input_data)`

Argument for both FLAME and DAME objects:

- `input_data`, type string, dataframe, Required Parameter. The dataframe on which to perform the matching, or the location of the CSV with the dataframe.

Arguments for FLAME object only:

- `C`, type float, default=0.1. The tradeoff parameter between the balancing factor and the predictive error when deciding which covariates to match on.
- `pre_dame`, type bool, integer, default=False. If an integer is provided, this is the number of iterations to run the FLAME algorithm for before switching to DAME, in order for a hybrid FLAME-DAME option.

Return Values:

- `Result`. Pandas dataframe of matched units and covariates matched on, with a "*" at each covariate that a unit did not use in matching.

Matching Class Attributes

- `units_per_group`, type array. This is an array of arrays. Each sub-array is a matched group, and each item in each sub-array is an int, indicating the unit in that matched group. If matching is done with 'repeats=False' then no unit will appear more than once. If 'repeats=True' then the first group in which a unit appears is its main matched group.
- `df_units_and_covars_matched`, type dataframe. This is the resulting matches of DAME. Each matched unit is in this array, and the covariates they were matched on have the value used to match. The covariates units were not matched on are indicated with a '*'.
- `groups_per_unit`, type array. The length of this is equal to the number of units in the input array. Each item in this array corresponds to the number of times that each item was matched. If matching is done with repeats=False, then this number will be either 0 or 1.

- `bf_each_iter`, array. If `want_bf` parameter is True, this will contain the balancing factor of the chosen covariate set at each iteration.

- `pe_each_iter`, array. If `want_pe` parameter is True, this will contain the predictive error of the chosen covariate set at each iteration.

Post-processing Utils

`dame_flame.utils.post_processing.MG(matching_object, unit_ids, output_style=1)`
`dame_flame.utils.post_processing.ATE(matching_object)`
`dame_flame.utils.post_processing.CATE(matching_object, unit_ids)`
`dame_flame.utils.post_processing.ATT(matching_object)`

Arguments:

- `matching_object`, type `dame_flame.matching.DAME`, `dame_flame.matching.FLAME`. The matching object used to run DAME and FLAME, after the `.fit()` and `.predict()` methods have been called to create the matches. If the `matching_object`'s parameter for `verbose` is not 0, then as units without matches appear, the function will print this.

- `unit_ids`, type int, list. A unit id or list of unit ids.

- `output_style`, int: 0,1. Default=1. If 1, the covariates which were not used in matching for the group of the unit will have a "*" rather than the covariate value. Otherwise, it will output all covariate values.

Return Values:

- Matched Groups (MG). type list, dataframe, np.nan. If one unit id was provided, this is a single dataframe containing the main matched group of the unit. If the unit does not have a match, the return will be np.nan. If multiple unit ids were provided, this will be a list of dataframes with the main matched group of each unit provided. If any unit does not have a match, rather than a dataframe, at its place will be np.nan.

- Average Treatment Effect (ATE), type float, np.nan. A float representing the ATE of the dataset. If no units were matched, then the output will be np.nan.

- Conditional Average Treatment Effect (CATE), type list, float, np.nan. If one unit id was provided, this is a single float representing the conditional average treatment effect of the unit. This is equal to the CATE of the group that the unit is in. If the unit does not have a match, the return will be np.nan. If multiple unit ids were provided, this will be a list of floats with the CATE of each unit provided. If any unit does not have a match, rather than a float within the list, at its place will be np.nan.

- Average Treatment Effect on Treated (ATT), type float, np.nan. A float representing the ATT of the dataset. If no units were matched, then the output will be np.nan.

5. Conclusions

The FLAME and DAME algorithms for matching of observational data with discrete covariates provide interpretable and high-quality matches. The **dame-flame** open-source Python package offers efficient, easy-to-use implementations of these algorithms. The package is easily accessible, and here, we provide detailed documentation, with concrete examples. The package is written in a highly modular manner, facilitating the introduction of new features and variations of the DAME and FLAME algorithms. It is available at https://github.com/almost-matching-exactly/DAME-FLAME-Python-Package. We hope this package will be of use to social science researchers, health researchers, and beyond.

Acknowledgments

This work was supported in part by awards NIH R01EB025021, NSF IIS-1703431, and the Duke University Energy Initiative Energy Research Seed Fund.

References

Abadie A, Drukker D, Herr JL, Imbens GW (2004). "Implementing matching estimators for average treatment effects in Stata." *The stata journal*, **4**(3), 290–311.

Achilles C, Bain HP, Bellott F, Boyd-Zaharias J, Finn J, Folger J, Johnston J, Word E (2008). "Tennessee's Student Teacher Achievement Ratio (STAR) project." doi:10.7910/DVN/SIWH9F. URL https://doi.org/10.7910/DVN/SIWH9F.

Buuren Sv, Groothuis-Oudshoorn K (2010). "mice: Multivariate imputation by chained equations in R." *Journal of Statistical Software*, pp. 1–68.

Dieng A, Liu Y, Roy S, Rudin C, Volfovsky A (2019). "Interpretable Almost-Exact Matching for Causal Inference." In *The 22nd International Conference on Artificial Intelligence and Statistics, AISTATS 2019, 16-18 April 2019, Naha, Okinawa, Japan*, pp. 2445–2453.

Iacus S, King G, Porro G (2009). "cem: Software for Coarsened Exact Matching." *Journal of Statistical Software, Articles*, **30**(9), 1–27. ISSN 1548-7660. doi:10.18637/jss.v030.i09. URL https://www.jstatsoft.org/v030/i09.

Krueger AB, Whitmore DM (2001). "The effect of attending a small class in the early grades on college-test taking and middle school test results: Evidence from Project STAR." *The Economic Journal*, **111**(468), 1–28.

Little RJ (1988). "A test of missing completely at random for multivariate data with missing values." *Journal of the American statistical Association*, **83**(404), 1198–1202.

Miroglio B, *et al.* (2017). "pymatch." https://github.com/benmiroglio/pymatch.

Parikh H, Rudin C, Volfovsky A (2018). "MALTS: Matching After Learning to Stretch." *arXiv preprint arXiv:1811.07415*.

Sharma A, Kiciman E, *et al.* (2019). "DoWhy: A Python package for causal inference." https://github.com/microsoft/dowhy.

Stuart EA, King G, Imai K, Ho D (2011). "MatchIt: nonparametric preprocessing for parametric causal inference." *Journal of Statistical Software*.

Wang T, Morucci M, Awan MU, Liu Y, Roy S, Rudin C, Volfovsky A (2019). "FLAME: A Fast Large-scale Almost Matching Exactly Approach to Causal Inference." *CoRR*, **abs/1707.06315v7**. URL http://arxiv.org/abs/1707.06315.

Affiliation:

Neha R. Gupta
Duke University
Durham, North Carolina
Email: neha.r.gupta@duke.edu

Sudeepa Roy
Department of Computer Science
Duke University
Durham, North Carolina
E-mail: sudeepa@cs.duke.edu

Cynthia Rudin
Department of Computer Science, Department of Electrical and Computer Engineering
Duke University
Durham, North Carolina
E-mail: cynthia@cs.duke.edu

Alexander Volfovsky
Department of Statistical Science
Duke University
Durham, North Carolina
E-mail: alexander.volfovsky@duke.edu

Journal of Statistical Software http://www.jstatsoft.org/
published by the Foundation for Open Access Statistics http://www.foastat.org/
MMMMMM YYYY, Volume VV, Issue II *Submitted:* yyyy-mm-dd
doi:10.18637/jss.v000.i00 *Accepted:* yyyy-mm-dd