BPt

Release 1.3.6

sahahn

Jan 13, 2021

INSTALL

The following documentation includes information about the usage of this toolbox.

CHAPTER

ONE

FROM SCRATCH

Don't have python installed? Or new to python? It is recomended to first download a version of anaconda, though this is optional. https://www.anaconda.com/distribution/#download-section as that will take care of a number of depedecies right away (and ease cross os difficulties), and give you a jupyter notebook environment to optionally work in.

CHAPTER

TWO

PIP INSTALLATION

To download the latest stable release, you can do this through pip, pythons built in installer. Run on the command line,

```
pip install brain-pred-toolbox
```

CHAPTER

THREE

GITHUB / PIP INSTALLATION

Optionally, you can choose to download the latest development version through github. You need git installed for this, but on the plus side you are ensured to have the latest version. Run on the command line or anaconda prompt on windows, (In the location where you want BPt installed!)

```
git clone https://github.com/sahahn/BPt.git
```

Then navigate into the BPt folder, and run

```
cd BPt
pip install .
```

In the future, to grab the latest updated versions, navigate into the folder where you installed BPt, and run

```
git pull
pip install .
```

CHAPTER

FOUR

EXTRA LIBRARIES

There are a number of libraries which extend the functionality of the BPt. These can be found under docs in the file requiriments.txt. Notably, depending on your operating system, some of these additional libraries may not be installable through pip alone, and will require taking further library specific steps.

CHAPTER

FIVE

NEW USERS

BPt is provided as a python based library and api, with workflows designed to be run in jupyter notebook-esque environments. That said, a complementary web interface is under active development and can be downloaded and used from https://github.com/sahahn/BPt_app. Users can choose to either use the more flexible python based library and / or make use of the web interface application. Trade-offs to consider namely revolve around prior user experience (i.e., those without coding or python experience may find the web interface easier, whereas more experienced users might prefer the greater flexibility and integration with the rest of the python data science environment that the python api offers) and personal preference.

A few general introductory resources for learning python, jupyter notebooks and machine learning are provided below:

- Introduction to python: https://jakevdp.github.io/WhirlwindTourOfPython/
- Intro to Jupyter-notebook: https://www.dataquest.io/blog/jupyter-notebook-tutorial/
- Brief intro to Machine Learning in python / jupyter environment: https://www.kaggle.com/learn/intro-to-machine-learning

CHAPTER

SIX

WHY BPT?

BPt seeks to integrate a number of lower level packages towards providing a higher level package and user experience. On the surface it may appear that this utility overlaps highly with libraries like scikit-learn, or a combination of scikit-learn with nilearn. While this is in some cases true with respect to the base python library, BPt seeks to provide extended functionality beyond what is found in the base scikit-learn package. The most notable is perhaps direct integration with a web based interface, allowing most of the library functionality to be accessed by those without prior programming experience. That said, the python based library, independent of the GUI, seeks to offer new utility beyond that found in scikit-learn and other packages in a number of key ways:

- Scikit-learn is fundamentally a lower level package, one which provides a number of the building blocks used in our package. That said, scikit-learn does not impose any sort of inherent order on the analysis workflow, and is far more flexible than our library. Flexibility is highly useful for a number of use cases, but still leaves room for higher level packages. Specifically, higher level packages like ours can help to both impose a conceptual ordering of recommended steps, as well as abstracting away boilerplate code in addition to providing other convenience functions. While it is obviously possible for users to re-create all of the functionality found in our library with the base libraries we employ, it would require in some cases immense effort, thus the existence of our package.
- We offer advanced and flexible hyper-parameter optimization via tight integration with the python nevergrad package. This allows BPt to offer a huge range of optimization strategies beyond the Random and Grid Searches found in scikit-learn. Beyond offering differing optimization methods, Nevergrad importantly supports more advanced functionality such as nesting hyperparameters with dictionaries or choice like objects. This functionality allows users to easily implement within BPt almost auto-ML like powerful and expressive parameter searches.
- BPt allows hyper-parameter distributions to be associated directly with machine learning pipeline objects. This functionality we believe makes it easier to optimize hyper-parameters across a number of pipeline pieces concurrently. We further allow preset hyper-parameter distributions to be selected or modified. This functionality allows less experienced users to still be able to perform hyper-parameter optimization without specifying the sometimes complex parameters themselves. In the future we hope to support the sharing of user defined hyper-parameter distributions (note: this feature is already supported in the multi-user version of the web interface).
- We introduce meta hyper-parameter objects, e.g., the *Select* object, which allows specifying the choice between different pipeline objects as a hyper-parameter. For example, the choice between base two or more base models, say a Random Forest model and an Elastic Net Regression (each with their own associated distributions of hyper-parameters), can be specified with the Select wrapper as itself a hyper-parameter. In this way, broader modelling strategies can be defined explicitly within the hyper-parameter search. This allows researchers the ability to easily perform properly nested model selection, and thus avoid common pitfalls associated with performing too many repeated internal validations. Along similar lines, we introduce functionality where features can themselves be set as binary hyper-parameters. This allows for the possibility of higher level optimizations.
- BPt introduced the concept of Scopes across all pipeline pieces. Scopes are a way for different pipeline pieces to optionally act on only a subset of features. While a similar functionality is provided via ColumnTransformers in scikit-learn, our package improves on this principle in a number of ways. In particular, a number of transformers and feature selection objects can alter in sometimes unpredictable ways the number of features (i.e., the number

of features input before a PCA transformation, vs. the number of output features). Our package tracks these changes. What tracking these changes allows is for scopes to be set in downstream pipeline pieces, e.g., the model itself, and have that functional set of features passed to the piece still reflect the intention of the original scope. For example, if one wanted to perform one hot encoding on a feature X, then further specify that a sub-model within an ensemble learner should only be provided feature X, our implemented scope system would make this possible. Importantly, the number of output features from the one hot encoding does not need to be known ahead of time, which makes properly nesting transformations like this much more accessible. Scopes can be particularly useful in neuroimaging based ML where a user might have data from a number of different modalities, and further a combination of categorical and continuous variables, all of which they may want to dynamically treat differently.

- We seek to more tightly couple in general the interaction between data loading, defining cross validation strategies, evaluating ML pipelines and making sense of results. For libraries like scikit-learn, this coupling is explicitly discouraged (which allows them to provide an extremely modular set of building blocks). On the other hand, by coupling these processes on our end, we can introduce a number of conveniences to the end-user. These span a number of common use cases associated with neuroimaging based ML, including: Allowing the previously introduced concept of Scope. Abstracting away a number of the considerations that must be made when dealing with loading, plotting and modelling across different data types (e.g., categorical vs. continuous). Abstracting away a number of the considerations that must be made when dealing with loading, plotting and modelling with missing data / NaN's Data loading and visualization related functions, e.g., dropping outliers and automatically visually viewing the distributions of a number of input variables. The generation of automatic feature importances across different cross validation schemes, and their plotting. Exporting of loaded variable summary information to .docx format. The ability to easily save and load full projects. And more!
- At the cost of some flexibility, but with the aims of reducing potentially redundant and cognitively demanding choices, the Model Pipeline's constructed within BPt restrict the order in which the different pieces can be specified (e.g., Imputation must come before feature selection). That said, as all pipeline pieces are designed to accept custom objects, experienced users can easily get around this by passing in their own custom pipelines in relevant places. Whenever possible, we believe it to be a benefit to reduce researcher degrees of freedom.
- BPt provides a loader module which allows the arbitrary inclusion of non-typical 2D scikit-learn input directly integrated into the ML pipeline. This functionality is designed to work with the hyper-parameter optimization, scope and other features already mentioned, and allows the usage of common neuroimaging features such as 3D MRI volumes, timeseries, connectomes, surface based inputs, and more. Explicitly, this functionality is designed to be used and evaluated in a properly nested machine learning context. An immensely useful package in this regard is nilearn, and by extension nibabel. Nilearn provides a number of functions which work very nicely with this loader module. This can be seen as combining the functionality of our package and nilearn, as an alternative to combining scikit-learn and nilearn. While the latter is obviously possible and preferable for some users, we hope that providing a higher level interface is still useful to others.
- Along with the loader module, we currently include a number of extension objects beyond those found in the base nilearn library. These span the extraction of Network metrics from an adjacency matrix, support for extracting regions of interest from static or timeseries surfaces, the generation of randomly defined surface parcellations,and the automatic caching of user defined, potentially computationally expensive, loading transformations. We plan to continue to add more useful functions like these in the future.
- Additional measures of feature importance can be specified to be automatically computed. Further, by tracking how features change, it can be useful in certain circumstances to back project computed feature importances to their original space (e.g., in the case of pipeline where surfaces from a few modalities are loaded from disk along with a number of phenotypic categorical variables, a parcellation applied on just the surface volumes, feature selection performed separately for say a number of different modalities, and then a base model evaluated, feature importances from the base model can be automatically projected back to the different modalities surfaces).
- BPt integrates useful pieces from a number of other scikit-learn adjacent packages. These span from popular gradient boosting libraries lightgbm and xgboost, to ensemble options offered by deslib, feature importances as computed by the SHAP library, the Categorical Encoders library for categorical encoding options and more. By providing a unified interface for accessing these popular and powerful tools, we hope to make it easier for users

to easily integrate the latest advances in machine learning.

CHAPTER

SEVEN

CORE CONCEPTS

This section is devoted as a placeholder with more detailed information about different core components of the library. In particular, you will often find within other sections of the documentation links to sub-sections within the sections as a way of referring to a more detailed explanation around a concept when warranted.

7.1 Pipeline Objects

Across all base *Model_Pipeline* pieces, e.g., *Model* or *Scaler*, there exists an *obj* param when initizalizing these objects. This parameter can broadly refer to either a str, which indicates a valid pre-defined custom obj for that piece, or depending on the pieces, this parameter can be passed a custom object directly.

7.2 Params

On the back-end, if a *Param_Search* object is passed when creating a *Model_Pipeline*, then a hyperparameter search will be conducted. All Hyperparameter search types are implemented on the backend with facebook's Nevergrad library.

Specific hyperparameters distributions in which to search over are set within their corresponding base Model_Pipeline object, e.g., the params argument is *Model*. For any object with a params argument you can set an associated hyperparameter distribution, which specifies values to search over (again assuming that param_search != None, if param_search is None, only passed params with constant values will be applied to object of interest, and any with associated Nevergrad parameter distributions will just be ignored).

You have two different options in terms of input that params can accept, these are:

- **Select a preset distribution** To select a preset, BPt defined, distribution, the selected object must first have atleast one preset distribution. These options can be found for each object specifically in the documentation under where that object is defined. Specifially, they will be listed with both an integer index, and a corresponding str name (see *Models*).

  For example, in creating a binary *Model* we could pass:

```
# Option 1 - as int
model = Model(obj = "dt classifier",
              params = 1)

# Option 2 - as str
model = Model(obj = "dt classifier",
              params = "dt classifier dist")
```

  In both cases, this selects the same preset distribution for the decision tree classifier.

- **Pass a custom nevergrad distribution** If you would like to specify your own custom hyperparameter distribution to search over, you can, you just need to specify it as a python dictionary of nevergrad parameters (follow the link to learn more about how to specify nevergrad params). You can also go into the source code for BPt, specifically BPt/helpers/Default_Params.py, to see how the preset distributions are defined, as a further example.

  Specifically the dictionary of params should follow the scikit_learn param dictionary format, where the each key corresponds to a parameter, but the value as a nevergrad parameter (instead of scikit_learn style). Further, if you need to specify nested parameters, e.g., for a custom object, you seperate parameters with '__', so e.g., if your custom model has a base_estimator param, you can pass:

  ```
  params = {'base_estimator__some_param' : nevergrad dist}
  ```

  Lastly, it is worth noting that you can pass either just static values or a combination of nevergrad distributions and static values, e.g.,

  ```
  {'base_estimator__some_param' : 6}
  ```

  (Note: extra params can also be used to pass static values, and extra_params takes precedence if a param is passed to both params and extra_params).

The special input wrapper *Select* can also be used to implicitly introduce hyperparameters into the *Model_Pipeline*.

7.3 Scopes

During the modeling and testing phases, it is often desirable to specify a subset of the total loaded columns/features. Within BPt the way subsets of columns can be specifed to different functions is through scope parameters.

The *scope* argument can be found across different *Model_Pipeline* pieces and within Problem_Spec.

The base preset str options that can be passed to scope are:

- **'all'** To specify all features, everything, regardless of data type.
- **'float'** To apply to all non-categorical columns, in both loaded data and covars.
- **'data'** To apply to all loaded data columns only.
- **'data files'** To apply to just columns which were originally loaded as data files.
- **'float covars' or 'fc'** To apply to all non-categorical, float covars columns only.
- **'cat' or 'categorical'** To apply to just loaded categorical data.
- **'covars'** To apply to all loaded covar columns only.

Beyond these base options, their exists a system for passing in either an array-like or tuple of keys to_use, wildcard stub strs for selecting which columns to use, or a combination. We will discuss these options in more detail below:

In the case that you would like to select a custom array-like of column names, you could simply pass: (where selected columns are the features that would be selected by that scope)

```
# As tuple
scope = ('name1', 'name2', 'name3')

# This is the hypothetical output, not what you pass
selected_columns = ['name1', 'name2', 'name3']
```

(continues on next page)

(continued from previous page)

```
# Or as array
scope = np.array(['some long list of specific keys'])

selected_columns = ['some long list of specific keys']
```

In this case, we are assuming the column/feature names passed correspond exactly to loaded column/ feature names. In this case, if all items within the array-like scope are specific keys, the columns used by that scope will be just those keys.

The way the wildcard systems works is similar to the custom array option above, but instead of passing an array of specific column names, you can pass one or more wildcard strs where in order for a column/feature to be included that column/feature must contain as a sub-string ALL of the passed substrings. For example: if the loaded data had columns 'name1', 'name2', 'name3' and 'somethingelse3'. By passing different scopes, you can see the corresponding selected columns:

```
# Single wild card
scope = '3'

selected_columns = ['name3', 'somethingelse3']

# Array-like of wild cards
scope =  ['3', 'name']

selected_columns = ['name3']
```

You can further provide a composition of different choices also as an array-like list. The way this composition works is that every entry in the passed list can be either: one of the base preset str options, a specific column name, or a substring wildcard.

The selected columns can then be thought of as a combination of these three types, where the output will be the same as if took the union from any of the preset keys, specific key names and the columns selected by the wildcard. For example, assuming we have the same loaded columns as above, and that 'name2' is the only loaded feature with datatype 'float':

```
scope = ['float', 'name1', 'something']

# 'float' selects 'name2', 'name1' selects 'name1', and wildcard something selects
↪'somethingelse3'
# The union of these is
selected_columns = ['name2', 'name1', 'somethingelse3']

# Likewise, if you pass multiple wildcard sub-strs, only the overlap will be taken as␣
↪before
scope = ['float', '3', 'name']

selected_columns = ['name2', 'name3']
```

Scopes more generally are associated 1:1 with their corresponding base Model_Pipeline objects (except for the Problem_Spec scope). One useful function designed specifically for objects with Scope is the *Duplicate* Inute Wrapper, which allows us to conviently replicate pipeline objects across a number of scopes. This functionality is especially useful with *Transformer* objects, (though still usable with other pipeline pieces, though other pieces tend to work on each feature independenly, ruining some of the benefit). For example consider a case where you would like to run a PCA tranformer on different groups of variables seperately, or say you wanted to use a categorical encoder on 15 different categorical variables. Rather then having to manually type out every combination or write a for loop, you can use *Duplicate*.

See *Duplicate* for more information on how to use this funcationality.

7.4 Extra Params

All base *Model_Pipeline* have the input argument *extra params*. This parameter is designed to allow passing additional values to the base objects, seperate from *Params*. Take the case where you are using a preset model, with a preset parameter distribution, but you only want to change 1 parameter in the model while still keeping the rest of the parameters associated with the param distribution. In this case, you could pass that value in extra params.

extra params are passed as a dictionary, where the keys are the names of parameters (only those accessible to the base classes init), for example if we were selecting the 'dt' ('decision tree') *Model*, and we wanted to use the first built in preset distribution for *Params*, but then fix the number of *max_features*, we could do it is as:

```
model = Model(obj = 'dt',
              params = 1,
              extra_params = {'max_features': 10})
```

7.5 Custom Input Objects

Custom input objects can be passed to the *obj* parameter for a number of base *Model_Pipeline* pieces.

There are though, depending on which base piece is being passed, different considerations you may have to make. More information will be provided here soon.

7.6 Subjects

Various functions within BPt can accept subjects as an argument. The parameter can accept a range of values.

First, you may pass the location to a text file with subject names seperated one on each line.

Secondly, you can pass any array-like (e.g., list, set, pandas Index, etc...), to pass an explicit list of subjects.

There are also a few reserved str key words which specify pre-defining groups of subjects. These are: 'all' to select all valid loaded subjects, 'train' and 'test' which specifies that the full set of globally defined training subjects, or testing subjects be used. See `Define_Train_Test_Split`.

Lastly, you can consider passing special input wrappers.

These are `Value_Subset` and `Values_Subset`. See each for more details on how they can be used.

CHAPTER

EIGHT

IN DEV

- **New Dataset class**
  - The new Dataset class is designed to eventually replace the old system for data loading.
- **Full refactor of feature importance plotting**
  - The plotting interface will eventually be moved fully to within the feature importance object as return as a result.

CHAPTER

NINE

RELEASE 1.3.6

- **Removed cache option from Model_Pipeline**
  - Use cache_loc parameter instead in each individual piece for more flexibility.
- **New search_only_params param**
  - In the Param_Search object, there is now a parameter for search_only_params.
  - This parameter allows some advanced behavior, w.r.t. to only passing params when searching for params.

CHAPTER

TEN

RELEASE 1.3.5

- **GridSearchCV support**
  - Added new abstract BPtSearchCV class.
  - Added in if search_type = 'grid' will try and convert parameters to grid search compatible, and use on the backend sklearn's GridSearchCV.
  - n_jobs will propegate correctly.
- **Replaced LGBM with BPtLBGM**
  - Replaces LGBMRegressor and Classifier with BPtLGBMRegressor and BPtLGBMClassifier.
  - These classes act as wrappers which automatically pass categorical features to the LGBM fit.
  - These classes also allow setting parameters 'early_stopping_rounds' and 'eval_split'.
- **Update Nevergrad version**
  - Update to nevergrad version 0.4.2.post5
  - Warning: The list of avaliable search types may be a little out of date.
- **New CV_Split class**
  - This can be used for passing single splits
- **New parameter fix_n_wrapper_jobs for Loader**
  - This parameter allows setting a fixed number of jobs for the Loading Wrapper.
  - In the future a better system for fixing n_jobs may be added.
- **Fix/Change internal representation for Scope Models + Transformers**
  - Introduce new internal classes for ScopeModel + ScopeTransformer.
  - These classes fixed a few existing bugs, and should make behavior moving forward more consistent.
- **Fix bug with Loader transform_df**
  - Fixed a bug with the transform_df function for Loaders.
  - This resulted in a error with computing feature importances for data loaded with a Loader.
- **Better pipeline names**
  - When using sklearn verbose, or inspecting models, a few names have been changed to look better / be more informative.

CHAPTER

ELEVEN

RELEASE 1.3.4

- **Added support for pandas >= 1**
  - Previously didn't support latest pandas.
- **Add sklearn OneHotEncoder**
  - Previously used category_encoders, use scikit-learn's instead for better and more reliable performance.
  - This object can be accessed as a transformer under 'one hot encoder'.
- **Added initial support for in-place FIs**
  - Moving from plotting via ML to plotting from the Feature Importance object itself
  - Only fully supports global right now.
- **Allow transformers to be skipped if out of scope**
  - Previously would cause error.

CHAPTER

TWELVE

RELEASE 1.3.3

- **Fixed bug with problem type**
  - There was an error which was mistakenly setting categorical problem type instead of regression.
- **Fixed internal bug with mapping**
  - Effected Transformer.
- **Added base_dtype option**
  - Evaluate and Test now have base dtype options, which allow changing dtype of data
  - Changed default data dtype from float64 to float32, should provide general speed ups

CHAPTER

THIRTEEN

RELEASE 1.3.1 AND 1.3.2

- **New AutoGluon option**
  - Can now specify the auto machine learning package AutoGluon as a *BPt.Model*
- **New SurfMaps extension loader**
  - Added new extension Loader `BPt.SurfMaps`
- **only_fold parameter**
  - New optional parameter in Evaluate for running only one fold.
- **Better support for scikit-learns VotingClassifier and VotingRegressor**
  - Similar to Stacking update from 1.3, but for voting classifier + regressor.
- **More support for nested pipelines**
  - Can now have nested pipelines propagate their parameter distributions to a parameter search in the top level pipeline.
- **Bug Fix with CV**
  - Fixed rare bug with CV expecting pandas Series, added support for passing numpy array.

CHAPTER

FOURTEEN

RELEASE 1.3

- **Support for nested parameter searches**
  - *BPt.Model* and *BPt.Ensemble* now support a param_search parameter.
  - The parameter param_search accepts a *BPt.Param_Search* object, and turns the model or ensemble into a nested search object.
- **Initial support for passing nested *BPt.Model_Pipeline***
  - Now can pass nested *BPt.Model_Pipeline* if wrapped in a *BPt.Model*
  - Warning: there are still cases which will not work.
- **Better support for stacking ensembles**
  - Stacking ensembles are ported from scikit-learn's StackingClassifier and StackingRegressor.
  - The `Ensemble` object can now support the arguments base_model and cv_splits.
  - The parameter, base_model allows passing in BPt compatible models to act as the final_estimator in stacking.
  - cv_splits allows passing a new input class *BPt.CV_Splits* which in the context of stacking, allows for custom CV behavior to train the base estimators.
- **Add experimental auto data type to loading targets**
  - You can now pass 'a' or 'auto' when loading targets to the data_type parameter to specify that the data type should be automatically inferred.
- **Change input parameter CV to cv**
  - In order to be more compatible with other libraries and intuative, now CV always refers to classes and cv an input parameter.
- **New Loky multi-processing support**
  - Changed to the new default mp_context.
  - Loky is a python library https://pypi.org/project/loky/ with better multiprocessing support than python's default.
- **New Dask multi-processing support**
  - Experimental support for dask multiprocessing
- **Fixed how n_jobs propegates in complex model pipelines**
  - New parameter in *BPt.Ensemble* n_jobs_type, which allows more controls over how n_jobs are spread out in the context of Ensembles.
- **Fixed bug with RandomParcels**

  - The RandomParcels object can be imported through from BPt.extensions import RandomParcels
  - A previous bug would allow some vertex labelled as medial wall, to be mislabeled, this has been fixed.

- **Add view to `BPt.Model`**
  - Initial support for an experimental *view* method for the `BPt.Model` class.
- **Improve the outputted results from Evaluate and Test**
  - Default feature importance to calculate is now None.
  - Added more optional parameters here.
  - Added new returned single metric.
  - Optional parameter for returning the trained model(s).
- **Add default case for `BPt.Problem_Spec`**
  - Now with default detecting of problem type, can optionally not specify a problem spec in Evaluate or Test.
- **Add default problem type**
  - Now if no target_type is specified, a default type will be set based on the type of the loaded target.
- **New default scorers**
  - The default scorers have changed, now provides multiple scorers for each type by default
- **Speed up working with Data Files**
  - Some improved performance in loading Data Files
- **Seperate caching for transformers and loaders**
  - Loaders and Transformers can now be cached via a cache_loc parameter.
- **Added experimental support for target transformation**
  - In some cases it is useful to allow nested transformations to the target variable.
  - `BPt.Model` and `BPt.Ensemble` now support an experimental argument for specifying a target transformation.
- **Introduce new `BPt.Values_Subset`**
  - In addition, added better description of *subjects* as a parameter type, with more universal behavior.
- **Large amounts of internal refactoring**
  - From docstrings, to structure of code, big amounts of re-factoring.
- **Name change from ABCD_ML to BPt**
  - Along with this change, the import of the ML object changed.
- **New support for k bins encoding when loading targets**
  - When loading targets, you may now specify a k-bins encoding scheme directly.
- **Renamed metric to scorer**
  - The argument metric has been renamed to scorer
  - The scorers accepted have also been re-defined to more closely align with scikit-learn's scorers.
- **Added support for categorical encoders and the categorical encoder library**

  - The new encouraged way to perform categorical encoding is by specifying transformers, via added options from the categorical encoders library.

- **New, now all parameter objects can accept scope as an argument**
  - In previous versions, input objects differed in which could accept a *scope* argument, now all can.
- **New ML verbosity options**
  - Some new ML verbosity options
- **Support latest scikit-learn version**
  - Backend changes allowing full compat. with latest scikit-learn versions.
- **Add more print information**
  - In an effort to make more of the library behavior transparent, more verbose print info has been added by default.
- **Removed ML class eval and test scores**
  - Depreciated the class wide eval and test scores previously stored in ML object

CHAPTER

FIFTEEN

MODEL_PIPELINE

class BPt.**Model_Pipeline**(*loaders=None*, *imputers='default'*, *scalers=None*, *transformers=None*, *feat_selectors=None*, *model='default'*, *param_search=None*, *n_jobs='default'*, *cache='depreciated'*, *feat_importances='depreciated'*)

Model_Pipeline is defined as essentially a wrapper around all of the explicit modelling pipeline parameters. This object is used as input to *Evaluate* and *Test*

The ordering of the parameters listed below defines the pre-set order in which these Pipeline pieces are composed (params up to model, param_search is not an ordered pipeline piece). For more flexibility, one can always use custom defined objects, or even pass custom defined pipelines directly to model (i.e., in the case where you have a specific pipeline you want to use already defined, but say just want to use the loaders from BPt).

Parameters

- **loaders** (*Loader*, list of or None, optional) – Each *Loader* refers to transformations which operate on loaded Data_Files (See *Load_Data_Files*). See *Loader* explcitly for more information on how to create a valid object, with relevant params and scope.

  In the case that a list of Loaders is passed to loaders, if a native python list, then passed loaders will be applied sequentially (likely each passed loader given a seperate scope, as the output from one loader cannot be input to another- note to create actual sequential loader steps, look into using the *Pipe* wrapper argument when creating a single *Loader* obj).

  Passed loaders can also be wrapped in a *Select* wrapper, e.g., as either

  ```
  # Just passing select
  loaders = Select([Loader(...), Loader(...)])

  # Or nested
  loaders = [Loader(...), Select([Loader(...), Loader(...)])]
  ```

  In this way, most of the pipeline objects can accept lists, or nested lists with param wrapped, not just loaders!

  ```
  default = None
  ```

- **imputers** (*Imputer*, list of or None, optional) – If there is any missing data (NaN's) that have been kept within data or covars, then an imputation strategy must be defined! This param controls what kind of imputation strategy to use.

  Each *Imputer* contains information around which imputation strategy to use, what scope it is applied to (in this case only 'float' vs. 'cat'), and other relevant base parameters (i.e., a base model if an iterative imputer is selected).

In the case that a list of *Imputer* are passed, they will be applied sequentially, though note that unless custom scopes are employed, at most passing only an imputer for float data and an imputer for categorical data makes sense. You may also use input wrapper, like *Select*.

In the case that no NaN data is passed, but imputers is not None, it will simply be set to None.

```
default = [Imputer('mean', scope='float'),
           Imputer('median', scope='cat')]
```

- **scalers** (*Scaler*, list of or None, optional) – Each *Scaler* refers to any potential data scaling where a transformation on the data (without access to the target variable) is computed, and the number of features or data points does not change. Each *Scaler* object contains information about the base object, what scope it should be applied to, and saved param distributions if relevant.

  As with other pipeline params, scalers can accept a list of *Scaler* objects, in order to apply sequential transformations (or again in the case where each object has a seperate scope, these are essentially two different streams of transformations, vs. when two Scalers with the same scope are passed, the output from one is passed as input to the next). Likewise, you may also use valid input wrappers, e.g., *Select*.

  By default no scaler is used, though it is reccomended.

  ```
  default = None
  ```

- **transformers** (*Transformer*, list of or None, optional) – Each *Transformer* defines a type of transformation to the data that changes the number of features in perhaps non-deterministic or not simply removal (i.e., different from feat_selectors), for example applying a PCA, where both the number of features change, but also the new features do not 1:1 correspond to the original features. See *Transformer* for more information.

  Transformers can be composed sequentially with list or special input type wrappers, the same as other objects.

  ```
  default = None
  ```

- **feat_selectors** (*Feat_Selector*, list of or None, optional) – Each *Feat_Selector* refers to an optional feature selection stage of the Pipeline. See *Feat_Selector* for specific options.

  Input can be composed in a list, to apply feature selection sequentially, or with special Input Type wrapper, e.g., *Select*.

  ```
  default = None
  ```

- **model** (*Model*, *Ensemble*, optional) – model accepts one input of type *Model* or *Ensemble*. Though, while it cannot accept a list (i.e., no sequential behavior), you may still pass Input Type wrapper like *Select* to perform model selection via param search.

  See *Model* for more information on how to specify a single model to BPt, and *Ensemble* for information on how to build an ensemble of models.

  Note: You must have provide a model, there is no option for None. Instead default behavior is to use a ridge regression.

  ```
  default = Model('ridge')
  ```

- **param_search** (*Param_Search* or None, optional) – *Param_Search* can be provided in order to specify a corresponding hyperparameter search for the provided pipeline pieces. When defining each piece, you may set hyperparameter distributions for that piece. If param search is None, these distribution will be essentially ignored, but if *Param_Search* is passed here, then they will be used along with the strategy defined in the passed *Param_Search* to conduct a nested hyper-param search.

  Note: If using input wrapper types like *Select*, then a param search must be passed!

  ```
  default = None
  ```

- **n_jobs** (*int or 'default', optional*) – The number of cores to be used with this pipeline. In general, this parameter should be left as 'default', which will set it based on the n_jobs as set in the problem spec- and will attempt to automatically change this value if say in the context of nesting.

  ```
  default = 'default'
  ```

- **cache** (*depreciated*) – The cache parameter has been depreciated, use the cache_loc params within individual pieces instead.

  ```
  default = 'depreciated'
  ```

feat_importances [depreciated] Feature importances in a past version of BPt were specified via this Model Pipeline object. Now they should be provided to either *Evaluate* and *Test*

```
default = 'depreciated'
```

CHAPTER

SIXTEEN

PROBLEM_SPEC

class BPt.**Problem_Spec**(*problem_type='default'*, *target=0*, *scorer='default'*, *weight_scorer=False*, *scope='all'*, *subjects='all'*, *n_jobs='default'*, *random_state='default'*)

Problem Spec is defined as an object of params encapsulating the set of parameters shared by modelling class functions *Evaluate* and *Test*

Parameters

- **problem_type** (*str or 'default', optional*) – This parameter controls what type of machine learning should be conducted. As either a regression, or classification where 'categorical' represents a special case of binary classification, where typically a binary classifier is trained on each class.
  - **'default'** Determine the problem type based on how the requested target variable is loaded.
  - **'regression', 'f' or 'float'** For ML on float/continuous target data.
  - **'binary' or 'b'** For ML on binary target data.
  - **'categorical' or 'c'** For ML on categorical target data, as multiclass.

```
default = 'default'
```

- **target** (*int or str, optional*) – The loaded target in which to use during modelling. This can be the int index (as assigned by order loaded in, e.g., first target loaded is 0, then the next is 1), or the name of the target column. If only one target is loaded, just leave as default of 0.

```
default = 0
```

- **scorer** (*str or list, optional*) – Indicator str for which scorer(s) to use when calculating average validation score in Evaluate, or Test set score in Test.

  A list of str's can be passed as well, in this case, scores for all of the requested scorers will be calculated and returned.

  Note: If using a Param_Search, the Param_Search object has a scorer parameter as well. This scorer describes the scorer optimized in a parameter search.

  For a full list of supported scorers please view the scikit-learn docs at: https://scikit-learn.org/stable/modules/model_evaluation.html#the-scoring-parameter-defining-model-evaluation-rules

  If left as 'default', assign a reasonable scorer based on the passed problem type.

  - 'regression' : ['explained_variance', 'neg_mean_squared_error']
  - 'binary' : ['matthews', 'roc_auc', 'balanced_accuracy']

– 'categorical' : ['matthews', 'roc_auc_ovr', 'balanced_accuracy']

```
default = 'default'
```

- **weight_scorer** (*bool, list of, optional*) – If True, then the scorer of interest will be weighted within each repeated fold by the number of subjects in that validation set. This parameter only typically makes sense for custom split behavior where validation folds may end up with differing sizes. When default CV schemes are employed, there is likely no point in applying this weighting, as the validation folds will have simmilar sizes.

  If you are passing mutiple scorers, then you can also pass a list of values for weight_scorer, with each value set as boolean True or False, specifying if the corresponding scorer by index should be weighted or not.

```
default = False
```

- **scope** (*key str or Scope obj, optional*) – This parameter allows the user to optionally run an expiriment with just a subset of the loaded features / columns.

  See *Scopes* for a more detailed explained / guide on how scopes are defined and used within BPt.

```
default = 'all'
```

- **subjects** (*str, array-like or Value_Subset, optional*) – This parameter allows the user to optionally run Evaluate or Test with just a subset of the loaded subjects. It is notably distinct from the *train_subjects*, and *test_subjects* parameters directly avaliable to Evaluate and Test, as those parameters typically refer to train/test splits. Specifically, any value specified for this subjects parameter will be applied AFTER selecting the relevant train or test subset.

  One use case for this parameter might be specifying subjects of just one sex, where you would still want the same training set for example, but just want to test sex specific models.

  If set to 'all' (as is by default), all avaliable subjects will be used.

  subjects can accept either a specific array of subjects, or even a loc of a text file (formatted one subject per line) in which to read from.

  A special wrapper, Value_Subset, can also be used to specify more specific, specifically value specific, subsets of subjects to use. See *Value_Subset* for how this input wrapper can be used.

```
default = 'all'
```

- **n_jobs** (*int, or 'default'*) – n_jobs are employed witin the context of a call to Evaluate or Test. If left as default, the class wide BPt value will be used.

  In general, the way n_jobs are propegated to the different pipeline pieces on the backend is that, if there is a parameter search, the base ML pipeline will all be set to use 1 job, and the n_jobs budget will be used to train pipelines in parellel to explore different params. Otherwise, if no param search, n_jobs will be used for each piece individually, though some might not support it.

```
default = 'default'
```

- **random_state** (*int, RandomState instance, None or 'default', optional*) – Random state, either as int for a specific seed, or if None then the random seed is set by np.random.

This parameter is used to ensure replicability of expirements (wherever possible!). In some cases even with a random seed, depending on the pipeline pieces being used, if any have a component that occassionally yields different results, even with the same random seed, e.g., some model optimizations, then you might still not get exact replicicability.

If 'default', use the saved class value. (Defined in *ML*)

```
default = 'default'
```

CHAPTER

SEVENTEEN

PIECES

17.1 Loader

class BPt.**Loader**(*obj*, *params=0*, *scope='data files'*, *cache_loc=None*, *extra_params=None*, *fix_n_wrapper_jobs=False*)

Loader refers to transformations which operate on loaded Data_Files. (See `Load_Data_Files()`). They in essence take in saved file locations, and after some series of transformations pass on compatible features. Notably loaders define operations which are computed on single files indepedently.

Parameters

- **obj** (str, custom obj or *Pipe*) – *obj* selects the base loader object to use, this can be either a str corresponding to one of the preset loaders found at *Loaders*. Beyond pre-defined loaders, users can pass in custom objects (they just need to have a defined fit_transform function which when passed the already loaded file, will return a 1D representation of that subjects features.

  obj can also be passed as a *Pipe*. See *Pipe*'s documentation to learn more on how this works, and why you might want to use it.

  See *Pipeline Objects* to read more about pipeline objects in general.

  For example, the 'identity' loader will load in saved data at the stored file location, lets say they are 2d numpy arrays, and will return a flattened version of the saved arrays, with each data point as a feature. A more practical example might constitute loading in say 3D neuroimaging data, and passing on features as extracted by ROI.

- **params** (int, str or dict of *params*, optional) – *params* determines optionally if the distribution of hyper-parameters to potentially search over for this loader. Preset param distributions are listed for each choice of obj at *Loaders*, and you can read more on how params work more generally at *Params*.

  If obj is passed as *Pipe*, see *Pipe* for an example on how different corresponding params can be passed to each piece individually.

  ```
  default = 0
  ```

- **scope** (*valid scope*, optional) – *scope* determines on which subset of features the specified loader should transform. See *Scopes* for more information on how scopes can be specified.

  You will likely want to use either custom key based scopes, or the 'data files' preset scope, as something like 'covars' won't make much sense, when atleast for now, you cannot even load Covars data files.

  ```
  default = 'data files'
  ```

- **cache_loc** (*str, Path or None, optional*) – Optional location in which to cache loader transformations.
- **extra_params** (*:ref`extra params dict<Extra Params>`, optional*) – See *Extra Params*

```
default = None
```

- **fix_n_wrapper_jobs** (*int or False, optional*) – Typically this parameter is left as default, but in special cases you may want to set this. It controls the number of jobs fixed for the Loading Wrapper.

  This parameter can be used to set that value.

```
default = False
```

17.2 Imputer

class BPt.**Imputer**(*obj*, *params=0*, *scope='all'*, *base_model=None*, *base_model_type='default'*, *extra_params=None*)

If there is any missing data (NaN's) that have been kept within data or covars, then an imputation strategy must be defined! This object allows you to define an imputation strategy. In general, you should need at most two Imputers, one for all *float* type data and one for all categorical data, assuming you have been present, and they both have missing values.

Parameters

- **obj** (*str*) – *obj* selects the base imputation strategy to use. See *Imputers* for all avaliable options. Notably, if 'iterative' is passed, then a base model must also be passed! Also note that the *sample_posterior* argument within *iterative* imputer is not currently supported.

  See *Pipeline Objects* to read more about pipeline objects in general.

- **params** (int, str or dict of *params*, optional) – *params* set an associated distribution of hyper-parameters to potentially search over with the Imputer. Preset param distributions are listed for each choice of params with the corresponding obj at *Imputers*, and you can read more on how params work more generally at *Params*.

```
default = 0
```

- **scope** (*{'float', 'cat', custom}, optional*) – *scope* determines on which subset of features the specified imputer will have access to.

  The main options that make sense for imputer are one for *float* data and one for *categorical* / 'cat' datatypes. Though you can also pass a custom set of keys.

  Note: If using iterative imputation you may want to carefully consider the scope passed. For example, while it may be beneficial to impute categorical and float features seperately, i.e., with different base_model_type's (categorical for categorical and regression for float), you must also consider that in predicting the missing values under this setup, the categorical imputer would not have access to to the float features and vice versa.

  In this way, you may want to either just treat all features as float, or instead of imputing categorical features, load missing values as a seperate category - and then set the scope here to be 'all', such that the iterative imputer has access to all features. Essently why this is neccisary is the iterative imputer will try to replace any NaN value present in its input features.

See *Scopes* for more information on how scopes can be specified.

```
default = 'all'
```

- **scope** – *scope* determines on which subset of features the imputer should act on.

  Scopes.

  ```
  default = 'float'
  ```

- **base_model** (*Model*, *Ensemble* or None, optional) – If 'iterative' is passed to obj, then a base_model is required in order to perform iterative imputation! The base model can be any valid Model_Pipeline Model.

  ```
  default = None
  ```

- **base_model_type** (*'default' or Problem Type, optional*) – In setting a base imputer model, it may be desirable to have this model have a different 'problem type', then your over-arching problem. For example, if performing iterative imputation on categorical features only, you will likely want to use a categorical predictor - but for imputing on float-type features, you will want to use a 'regression' type base model.

  Choices are {'binary', 'regression', 'categorical'} or 'default'. If 'default', then the following behavior will be applied: If the scope of the imputer is set to 'cat' or 'categorical', then the 'categorical' problem type will be used for the base model. If anything else, then the 'regression' type will be used.

  ```
  default = 'default'
  ```

extra_params : :ref'extra params dict<Extra Params>', optional

See *Extra Params*

```
default = None
```

17.3 Scaler

class BPt.**Scaler**(*obj*, *params=0*, *scope='float'*, *extra_params=None*)

Scaler refers to a piece in the *Model_Pipeline*, which is responsible for performing any sort of scaling or transformation on the data which doesn't require the target variable, and doesn't change the number of data points or features.

Parameters

- **obj** (*str or custom obj*) – *obj* if passed a str selects a scaler from the preset defined scalers, See *Scalers* for all avaliable options. If passing a custom object, it must be a sklearn compatible transformer, and further must not require the target variable, not change the number of data points or features.

  See *Pipeline Objects* to read more about pipeline objects in general.

- **params** (int, str or dict of *params*, optional) – *params* set an associated distribution of hyper-parameters to potentially search over with this Scaler. Preset param distributions are listed for each choice of params with the corresponding obj at *Scalers*, and you can read more on how params work more generally at *Params*.

```
default = 0
```

- **scope** (*valid scope*, optional) – *scope* determines on which subset of features the specified scaler should transform. See *Scopes* for more information on how scopes can be specified.

```
default = 'float'
```

- **extra_params** (*:ref`extra params dict<Extra Params>`, optional*) – See *Extra Params*

```
default = None
```

17.4 Transformer

class BPt.**Transformer**(*obj*, *params=0*, *scope='float'*, *cache_loc=None*, *extra_params=None*, *fix_n_wrapper_jobs='default'*)

The Transformer is base optional component of the *Model_Pipeline* class. Transformers define any type of transformation to the loaded data which may change the number of features in a non-simple way (i.e., conceptually distinct from *Feat_Selector*, where you know in advance the transformation is just selecting a subset of existing features). These are transformations like applying Principle Component Analysis, or on the fly One Hot Encoding.

Parameters

- **obj** (*str or custom_obj*) – *obj* if passed a str selects from the avaliable class defined options for transformer as found at *Transformers*.

  If a custom object is passed as *obj*, it must be a sklearn api compatible transformer (i.e., have fit, transform, get_params and set_params methods, and further be cloneable via sklearn's clone function). See *Custom Input Objects* for more info.

  See *Pipeline Objects* to read more about pipeline objects in general.

- **params** (int, str or dict of *params*, optional) – *params* determines optionally if the distribution of hyper-parameters to potentially search over for this transformer. Preset param distributions are listed for each choice of obj at *Transformers*, and you can read more on how params work more generally at *Params*.

```
default = 0
```

- **scope** (*valid scope*, optional) – *scope* determines on which subset of features the specified transformer should transform. See *Scopes* for more information on how scopes can be specified.

  Specifically, it may be useful to consider the use of *Duplicate* here.

```
default = 'float'
```

- **extra_params** (*:ref`extra params dict<Extra Params>`, optional*) – See *Extra Params*

```
default = None
```

- **fix_n_wrapper_jobs** (*int or 'default', optional*) – This parameter is ignored right now for Transformers

```
default = 'default'
```

17.5 Feat_Selector

class BPt.**Feat_Selector**(*obj*, *params=0*, *scope='all'*, *base_model=None*, *extra_params=None*)

Feat_Selector is a base piece of *Model_Pipeline*, which is designed to preform feature selection.

Parameters

- **obj** (*str*) – *obj* selects the feature selection strategy to use. See *Feat Selectors* for all avaliable options. Notably, if 'rfe' is passed, then a base model must also be passed!

  See *Pipeline Objects* to read more about pipeline objects in general.

- **params** (int, str or dict of *params*, optional) – *params* set an associated distribution of hyper-parameters to potentially search over with this Feat_Selector. Preset param distributions are listed for each choice of params with the corresponding obj at *Feat Selectors*, and you can read more on how params work more generally at *Params*.

  ```
  default = 0
  ```

- **scope** (*valid scope*, optional) – *scope* determines on which subset of features the specified feature selector will have access to. See *Scopes* for more information on how scopes can be specified.

  ```
  default = 'all'
  ```

- **base_model** (*Model*, *Ensemble* or None, optional) – If 'rfe' is passed to obj, then a base_model is required in order to perform recursive feature elimination. The base model can be any valid argument accepts by param *model* in *Model_Pipeline*.

  ```
  default = None
  ```

- **extra_params** (*:ref`extra params dict<Extra Params>`, optional*) – See *Extra Params*

  ```
  default = None
  ```

17.6 Model

class BPt.**Model**(*obj*, *params=0*, *scope='all'*, *param_search=None*, *target_scaler=None*, *extra_params=None*)

Model represents a base components of the *Model_Pipeline*, specifically a single Model / estimator. Model can also be used as a component in building other pieces of the model pipeline, e.g., *Ensemble*.

Parameters

- **obj** (*str, or custom obj*) – *obj* selects the base model object to use from either a preset str indicator found at *Models*, or from a custom passed user model (compatible w/ sklearn api).

  See *Pipeline Objects* to read more about pipeline objects in general.

  obj should be wither a single str indicator or a single custom model object, and not passed a list-like of either. If an ensemble of models is requested, then see *Ensemble*.

- **params** (int, str or dict of *params*, optional) – *params* optionally set an associated distribution of hyper-parameters to this model object. Preset param distributions are listed for each choice of obj at *Models*, and you can read more on how params work more generally at *Params*.

```
default = 0
```

- **scope** (*valid scope*, optional) – *scope* determines on which subset of features the specified model should work on. See *Scopes* for more information on how scopes can be specified.

```
default = 'all'
```

- **param_search** (*Param_Search*, None, optional) – If None, by default, this will be a base model. Alternatively, by passing a *Param_Search* instance here, it specifies that this model should be wrapped in a Nevergrad hyper-parameter search object.

  This can be useful to create Model's which have a nested hyper-parameter tuning independent from the other pipeline steps.

```
default = None
```

- **target_scaler** (*Scaler, None, optional*) – Still somewhat experimental, can pass a Scaler object here and have this model perform target scaling + reverse scaling.

  Note: Has not been fully tested in complicated nesting cases, e.g., if Model is wrapping a nested Model_Pipeline, this param will likely break.

```
default = None
```

- **extra_params** (*:ref`extra params dict<Extra Params>`, optional*) – See *Extra Params*

```
default = None
```

17.7 Ensemble

class BPt.**Ensemble**(*obj*, *models*, *params=0*, *scope='all'*, *param_search=None*, *target_scaler=None*, *base_model=None*, *cv_splits=None*, *is_des=False*, *single_estimator=False*, *des_split=0.2*, *n_jobs_type='ensemble'*, *extra_params=None*)

The Ensemble object is valid base *Model_Pipeline* piece, designed to be passed as input to the *model* parameter of *Model_Pipeline*, or to its own models parameters.

This class is used to create a variety ensembled models, typically based on *Model* pieces.

Parameters

- **obj** (*str*) – Each str passed to ensemble refers to a type of ensemble to train, based on also the passed input to the *models* parameter, and also the additional parameters passed when init'ing Ensemble.

  See *Ensemble Types* to see all avaliable options for ensembles.

  Passing custom objects here, while technically possible, is not currently full supported. That said, there are just certain assumptions that the custom object must meet in order to work, specifially, they should have simmilar input params to other simmilar existing ensembles, e.g., in the case the *single_estimator* is False and *needs_split* is also False, then the passed object needs to be able to accept an input parameter *estimators*, which accepts a list of (str,

estimator) tuples. Whereas if needs_split is still False, but single_estimator is True, then the passed object needs to support an init param of *base_estimator*, which accepts a single estimator.

- **models** (*Model*, *Ensemble* or list of) – The *models* parameter is designed to accept any single model-like pipeline parameter object, i.e., *Model* or even another *Ensemble*. The passed pieces here will be used along with the requested ensemble object to create the requested ensemble.

  See *Model* for how to create a valid base model(s) to pass as input here.

- **params** (int, str or dict of *params*, optional) – *params* sets as associated distribution of hyper-parameters for this ensemble object. These parameters will be used only in the context of a hyper-parameter search. Notably, these *params* refer to the ensemble obj itself, params for base *models* should be passed accordingly when creating the base models. Preset param distributions are listed at *Ensemble Types*, under each of the options for ensemble obj's.

  You can read more about generally about hyper-parameter distributions as associated with objects at *Params*.

```
default = 0
```

- **scope** (*valid scope*, optional) – *scope* determines on which subset of features the specified ensemble model should work on. See *Scopes* for more information on how scopes can be specified.

```
default = 'all'
```

- **param_search** (*Param_Search*, None, optional) – If None, by default, this will be a base ensemble model. Alternatively, by passing a *Param_Search* instance here, it specifies that this model should be wrapped in a Nevergrad hyper-parameter search object.

  This can be useful to create Model's which have a nested hyper-parameter tuning independent from the other pipeline steps.

```
default = None
```

- **target_scaler** (*Scaler, None, optional*) – Still somewhat experimental, can pass a Scaler object here and have this model perform target scaling + reverse scaling.

  scope in the passed scaler is ignored.

  Note: Has not been fully tested in complicated nesting cases, e.g., if Model is wrapping a nested Model_Pipeline, this param will likely break.

```
default = None
```

- **base_model** (*Model*, None, optional) – In the case that an ensemble is passed which has the parameter *final_estimator* (not base model!), for example in the case of stacking, then you may pass a Model type object here to be used as that final estimator.

  Otherwise, by default this will be left as None, and if the requested ensemble has the final_estimator parameter, then it will pass None to the object (which is typically for setting the default).

```
default = None
```

- **cv_splits** (*CV_Splits* or None, optional) – Used for passing custom CV split behavior to ensembles which employ splits, e.g., stacking.

```
default = None
```

- **is_des** (*bool, optional*) – *is_des* refers to if the requested ensemble obj requires a further training test split in order to train the base ensemble. As of right now, If this parameter is True, it means that the base ensemble is from the DESlib library . Which means the base ensemble obj must have a *pool_classifiers* init parameter.

  The following *des_split* parameter determines the size of the split if is_des is True.

```
default = False
```

- **single_estimator** (*bool, optional*) – The parameter *single_estimator* is used to let the Ensemble object know if the *models* must be a single estimator. This is used for ensemble types that requires an init param *base_estimator*. In the case that multiple models are passed to *models*, but *single_estimator* is True, then the models will automatically be wrapped in a voting ensemble, thus creating one single estimator.

```
default = False
```

- **des_split** (*float, optional*) – If *is_des* is True, then the passed ensemble must be fit on a seperate validation set. This parameter determines the size of the further train/val split on initial training set passed to the ensemble. Where the size is comptued as the a percentage of the total size.

```
default = .2
```

- **n_jobs_type** (*'ensemble' or 'models', optional*) – Valid options are either 'ensemble' or 'models'.

  This parameter controls how the total n_jobs are distributed, if 'ensemble', then the n_jobs will be used all in the ensemble object and every instance within the sub-models set to n_jobs = 1. Alternatively, if passed 'models', then the ensemble object will not be multi-processed, i.e., will be set to n_jobs = 1, and the n_jobs will be distributed to each base model.

  If you are training a stacking regressor for example with n_jobs = 16, and you have 16+ models, then 'ensemble' is likely a good choice here. If instead you have only 3 base models, and one or more of those 3 could benefit from a higher n_jobs, then setting n_jobs_type to 'models' might give a speed-up.

```
default = 'ensemble'
```

- **extra_params** (*:ref`extra params dict<Extra Params>`, optional*) – See *Extra Params*

```
default = None
```

17.8 Param_Search

class BPt.**Param_Search**(*search_type='RandomSearch', splits=3, n_repeats=1, cv='default', n_iter=10, scorer='default', weight_scorer=False, mp_context='default', n_jobs='default', dask_ip=None, memmap_X=False, search_only_params=None, CV='depreciated', _random_state=None, _splits_vals=None, _cv=None, _scorer=None, _n_jobs=None*)

Param_Search is special input object designed to be used with *Model_Pipeline*. Param_Search defines

a hyperparameter search strategy. When passed to *Model_Pipeline*, its search strategy is applied in the context of any set *Params* within the base pieces. Specifically, there must be atleast one parameter search somewhere in the object Param_Search is passed!

All backend hyper-parameter searches make use of the <https://github.com/facebookresearch/nevergrad>‘_ library.

Parameters

- **search_type** (*str, optional*) – The type of nevergrad hyper-parameter search to conduct. See *Search Types* for all avaliable options. Also you may further look into nevergrad’s experimental varients if you so choose, this parameter can accept those as well.

  New: You may pass ‘grid’ here in addition to the supported nevergrad searches. This will use sklearn’s GridSearch. Note in this case some of the other parameters are ignored, these are: weight_scorer, mp_context, dask_ip, memmap_X, search_only_params

  ```
  default = 'RandomSearch'
  ```

- **splits** (*int, float, str or list of str, optional*) – In order to optimize hyper-parameters, some sort of internal cross validation must be specified, such that combinations of hyper-parameters can be evaluated on different data then they were trained on. *splits* allows you to specify the base of what CV strategy should be used to evaluate every *n_iter* combination of hyper-parameters.

  Specifically, options for split are:

  – **int** The number of k-fold splits to conduct. (E.g., 3 for 3-fold CV split to be conducted at every hyper-param evaluation).

  – **float** Must be 0 < *splits* < 1, and defines a single train-test like split, with *splits* % of the current training data size used as a validation set.

  – **str** If a str is passed, then it must correspond to a loaded Strat variable. In this case, a leave-out-group CV will be used according to the value of the indicated Strat variable (E.g., a leave-out-site CV scheme).

  – **list of str** If multiple str passed, first determine the overlapping unique values from their corresponing loaded Strat variables, and then use this overlapped value to define the leave-out-group CV as described above.

  Also note that *n_repeats* will work with any of these options, but say in the case of a leave out group CV, would be awfully redundant, versus, with a passed float value, very reasonable.

  ```
  default = 3
  ```

- **n_repeats** (*int, optional*) – Given the base hyper-param search CV defined / described in the *splits* param, this parameter further controls if the defined train/val splits should be repeated (w/ different random splits in all cases but the leave-out-group passed str option).

  For example, if *n_repeats* is set to 2, and *splits* is 3, then a twice repeated 3-fold CV will be performed to evaluate every choice of *n_iter* hyper-params.

  ```
  default = 1
  ```

- **cv** (*CV* or ‘default’, optional) – If left as default ‘default’, use the class defined CV behavior for the splits, otherwise can pass custom behavior.

```
default = 'default'
```

- **n_iter** (*int, optional*) – The number of hyper-parameters to try / budget of the underlying search algorithm. How well a hyper-parameter search works and how long it takes will be very dependent on this parameter and the defined internal CV strategy (via *splits* and *n_repeats*). In general, if too few choices are provided the algorithm will likely not select high performing hyper-paramers, and alternatively if too high a value/budget is set, then you may find overfit/non-generalize hyper-parameter choices. Other factors which will influence the 'right' number of *n_iter* to specify are:

  – ***search_type*** Depending on the underlying search type, it may take a bigger or smaller budget on average to find a good set of hyper-parameters

  – **The dimension of the underlying search space** If you are only optimizing a few, say 2, underlying parameter distributions, this will require a far smaller budget then say a really high dimensional search space.

  – **The CV strategy** The CV strategy defined via *splits* and *n_repeats* may make it easier or harder to overfit when searching for hyper-parameters, thus conceptually a good choice of CV strategy can serve to increase the number *n_iter* you can use before overfitting, or conversely a bad choice may limit it.

  – **Number of data points / subjects** Along with CV strategy, the number of data points/subjects will greatly influence how quickly you overfit, and therefore a good choice of *n_iter*.

  Notably, one can always if they have the resources simply expiriment with this parameter.

```
default = 10
```

- **scorer** (*str or 'default', optional*) – In order for a set of hyper-parameters to be evaluated, a single scorer must be defined.

  For a full list of supported scorers please view the scikit-learn docs at: https://scikit-learn.org/stable/modules/model_evaluation.html#the-scoring-parameter-defining-model-evaluation-rules

  If left as 'default', assign a reasonable scorer based on the passed problem type.

  – 'regression' : 'explained_variance'

  – 'binary' : 'matthews'

  – 'categorical' : 'matthews'

  Be careful to make sure to select an appropriate scorer for the problem type.

  Only one value of *scorer* may be passed here.

```
default = 'default'
```

- **weight_scorer** (*bool or 'default', optional*) – *weight_scorer* describes if the scorer of interest should be weighted by the number of subjects within each validation fold. So, for example, if a leave-out-group CV scheme is specified to *splits*, and the groups have drastically different numbers of subjects, then you may want to consider weighting the final average validation metric (as computed across in this case all groups used by themselves) by the number of subjects in each fold.

```
default = False
```

- **mp_context** (*str, optional*) – When a hyper-parameter search is launched, there are different ways through python that the multi-processing can be launched (assuming n_jobs > 1). Occassionally some choices can lead to unexpected errors.

  Choices are: - 'default': If 'default' use the BPt mp_context.

  – 'loky': Create and use the python library loky backend.

  – 'fork': Python default fork mp_context

  – 'forkserver': Python default forkserver mp_context

  – 'spawn': Python default spawn mp_context

```
default = 'default'
```

- **n_jobs** (*int or 'default', optional*) – The number of cores to be used for the search. In general, this parameter should be left as 'default', which will set it based on the n_jobs as set in the problem spec- and will attempt to automatically change this value if say in the context of nesting.

```
default = 'default'
```

- **dask_ip** (*str or None, optional*) – If None, default, then ignore this parameter..

  For experimental Dask support. This should be the ip of a created dask cluster. A dask Client object will be created and passed this ip in order to connect to the cluster.

```
default = None
```

- **memmap_X** (*bool, optional*) – When passing large memory arrays in each parameter search, it can be useful as a memory reduction technique to pass numpy memmap'ed arrays. This solves an issue where the loky backend will not properly pass too large arrays.

  Warning: This can slow down code, and only reduces the actual memory consumption of each job by a little bit.

  Note: If passing a dask_ip, this option will be skipped, as if using the dask backend, then large X's will be pre-scattered instead.

```
default = False
```

- **search_only_params** (*dict or None, optional*) – In some rare cases, it may be the case that you want to specify that certain parameters be passed only during the nested parameter searches. A dict of parameters can be passed here to accomplish that. For example, if passing:

  search_only_params = {'svm classifier__probability': False}

  And assuming that the default / selecting parameter for this svm classifier for probaility is True by default, then only when exploring nested hyper-parameter options will probability be set to False, but when fitting the final model with the best parameters found from the search, it will revert back to the default, i.e., in this case probability = True.

  Note: this may be a little bit tricky to use as you need to know how to represent the parameters correctly!

  To ignore this parameter / option. simply keep the default value of None

```
default = None
```

- **CV** (*'depreciated'*) – Switching to passing cv parameter as cv instead of CV. Will raise error if anything is passed here.

```
default = 'depreciated'
```

17.9 Feat_Importance

class BPt.**Feat_Importance**(*obj*, *scorer='default'*, *shap_params='default'*, *n_perm=10*, *inverse_global=False*, *inverse_local=False*)

There are a number of options for creating Feature Importances in BPt. See *Feat Importances* to learn more about feature importances generally. The way this object works, is that you can a type of feature importance, and then its relevant parameters. This object is designed to passed directly to *Model_Pipeline*.

Parameters

- **obj** (*str*) – *obj* is the str indiciator for which feature importance to use. See *Feat Importances* for what options are avaliable.
- **scorer** (*str or 'default', optional*) – If a permutation based feature importance is being used, then a scorer is required.

  For a full list of supported scorers please view the scikit-learn docs at: https://scikit-learn.org/stable/modules/model_evaluation.html#the-scoring-parameter-defining-model-evaluation-rules

  If left as 'default', assign a reasonable scorer based on the passed problem type.

  – 'regression' : 'explained_variance'

  – 'binary' : 'matthews'

  – 'categorical' : 'matthews'

```
default = 'default'
```

- **shap_params** (Shap_Params or 'default', optional) – If a shap based feature importance is used, it is neccicary to define a number of relevant parameters for how the importances should be calculated. See Shap_Params for what these parameters are.

  If 'default' is passed, then shap_params will be set to either the default values of Shap_Params if shap feature importances are being used, or None if not.

```
default = 'default'
```

- **n_perm** (*int, optional*) – If a permutation based feature importance method is selected, then it is neccicary to indicate how many random permutations each feature should be permuted.

```
default = 10
```

- **inverse_global** (*bool*) – Warning: This feature, along with inverse_local, is still expirimental.

  If there are any loaders, or transformers specified in the Model_Pipeline, then feature importance becomes slightly trickier. For example, if you have a PCA transformer, and what to calculate averaged feature importance across 3-folds, there is no gaurentee 'pca feature 1' is the same from one fold to the next. In this case, if set to True, global feature importances will be inverse_transformed back into their original feature space - per fold. Note: this will

only work if all transformers / loaders have an implemented reverse_transform function, if one does not for transformer, then it will just return 0 for that feature. For a loader w/o, then it will return 'No inverse_transform'.

There are also other cases where this might be a bad idea, for example if you are using one hot encoders in your transformers then trying to reverse_transform feature importances will yield nonsense (NaN's).

```
default = False
```

- **inverse_local** (*bool*) – Same as inverse_global, but for local feature importances. By default this is set to False, as it is more memory and computationally expensive to inverse_transform this case.

```
default = False
```

17.10 CV

class BPt.**CV**(*groups=None*, *stratify=None*, *train_only_loc=None*, *train_only_subjects=None*)

This objects is used to encapsulate a set of parameters for a CV strategy.

Parameters

- **groups** (*str, list or None, optional*) – In the case of str input, will assume the str to refer to a column key within the loaded strat data, and will assign it as a value to preserve groups by during any train/test or K-fold splits. If a list is passed, then each element should be a str, and they will be combined into all unique combinations of the elements of the list.
- **::** – default = None
- **stratify** (*str, list or None, optional*) – In the case of str input, will assume the str to refer to a column key within the loaded strat data, or a loaded target col., and will assign it as a value to preserve distribution of groups by during any train/test or K-fold splits. If a list is passed, then each element should be a str, and they will be combined into all unique combinations of the elements of the list.

  Any target_cols passed must be categorical or binary, and cannot be float. Though you can consider loading in a float target as a strat, which will apply a specific k_bins, and then be valid here.

  In the case that you have a loaded strat val with the same name as your target, you can distinguish between the two by passing either the raw name, e.g., if they are both loaded as 'Sex', passing just 'Sex', will try to use the loaded target. If instead you want to use your loaded strat val with the same name - you have to pass 'Sex' + self.self.strat_u_name (by default this is '_Strat').

```
default = None
```

- **train_only_loc** (*str, Path or None, optional*) – Location of a file to load in train_only subjects, where any subject loaded as train_only will be assigned to every training fold, and never to a testing fold. This file should be formatted as one subject per line.

  You can load from a loc and pass subjects, the subjects from each source will be merged.

  This parameter is compatible with groups / stratify.

```
default = None
```

- **train_only_subjects** (*set, array-like, 'nan', or None, optional*) – An explicit list or array-like of train_only subjects, where any subject loaded as train_only will be assigned to every training fold, and never to a testing fold.

  You can also optionally specify 'nan' as input, which will add all subjects with any NaN data to train only.

  If you want to add both all the NaN subjects and custom subjects, call `Get_Nan_Subjects()` to get all NaN subjects, and then merge them yourself with any you want to pass.

  You can load from a loc and pass subjects, the subjects from each source will be merged.

  This parameter is compatible with groups / stratify.

```
default = None
```

17.11 CV_Splits

class BPt.**CV_Splits**(*cv='default', splits=3, n_repeats=1, _cv=None, _random_state=None, _splits_vals=None*)

This object is used to wrap around a CV strategy at a higher level.

Parameters

- **cv** ('default' or *CV*, optional) – If left as default 'default', use the class defined CV behavior for the splits, otherwise can pass custom behavior.
- **splits** (*int, float, str or list of str, optional*) – *splits* allows you to specify the base of what CV strategy should be used.

  Specifically, options for split are:

  – **int** The number of k-fold splits to conduct. (E.g., 3 for 3-fold CV split to be conducted at every hyper-param evaluation).

  – **float** Must be 0 < *splits* < 1, and defines a single train-test like split, with *splits %* of the current training data size used as a validation set.

  – **str** If a str is passed, then it must correspond to a loaded Strat variable. In this case, a leave-out-group CV will be used according to the value of the indicated Strat variable (E.g., a leave-out-site CV scheme).

  – **list of str** If multiple str passed, first determine the overlapping unique values from their corresponing loaded Strat variables, and then use this overlapped value to define the leave-out-group CV as described above.

  n_repeats is designed to work with any of these choices.

```
default = 3
```

- **n_repeats** (*int, optional*) – The number of times to repeat the defined strategy as defined in *splits*.

  For example, if *n_repeats* is set to 2, and *splits* is 3, then a twice repeated 3-fold CV will be performed

```
default = 1
```

CHAPTER

EIGHTEEN

INPUT TYPES

18.1 Select

class BPt.**Select**

The Select object is an BPt specific Input Wrapper designed to allow hyper-parameter searches to include not just choice of hyper-parameter, but also choosing between objects (as well as their relevant distributions).

Select is used to cast lists of base *Model_Pipeline* pieces as different options. Consider a simple example, for specifying a selection between two different *Models*

```
model = Select([Model('linear'), Model('random forest')])
```

In this example, the model passed to *Model_Pipeline* becomes a meta object for selecting between the two base models. Note: this does require a *Param_Search* object be passed to *Model_Pipeline*. Notably as well, if further param distributions are defined within say the *Model('random forest')*, those will still be optimized, allowing for potentially even a hyper-parameter search to select hyper-parameter distribution... (i.e., if both select options had the same base model obj, but only differed in the passed hyper-param distributions) if one were so inclined...

Other notable features of Select are, you are not limited to passing only two options, you can pass an arbitrary number... you can even, and I'm not even sure I want to tell you this... pass nested calls to Select... i.e., one of the Select options could be another Select, with say another Select...

Lastly, explcitly note that Select is not restricted for use with Models, it can be used on any of the base class:*Model_Pipeline* piece params (i.e., every param but param_search, feat_importances and cache...).

18.2 Duplicate

class BPt.**Duplicate**

The Duplicate object is an BPt specific Input wrapper. It is designed to be cast on a list of valid scope parameters, e.g.,

```
scope = Duplicate(['float', 'cat'])
```

Such that the corresponding pipeline piece will be duplicated for every entry within Duplicate. In this case, two copies of the base object will be made, where both have the same remaining non-scope params (i.e., obj, params, extra_params), but one will have a scope of 'float' and the other 'cat'.

Consider the following exentended example, where loaders is being specified when creating an instance of *Model_Pipeline*:

```
loaders = Loader(obj='identity', scope=Duplicate(['float', 'cat']))
```

Is transformed in post processing / equivalent to

```
loaders = [Loader(obj='identity', scope='float'),
           Loader(obj='identity', scope='cat')]
```

18.3 Pipe

class BPt.**Pipe**

The Pipe object is an BPt specific Input wrapper, designed for now to work specifically within *Loader*. Because loader objects within BPt are designed to work on single files at a time, and further are resitricted in that they must go directly from some arbitrary file, shape and charteristics to outputted as a valid 2D (# Subects X # Features) array, it restricts potential sequential compositions. Pipe offers some utilty towards building sequential compositions.

For example, say one had saved 4D neuroimaging fMRI timeseries, and they wanted to first employ a loader to extract timeseries by ROI (with say hyper-parameters defined to select which ROI to use), but then wanted to use another loader to convert the timeseries ROIs to a correlation matrix, and only then pass along the output as 1D features per subject. In this case, the Pipe wrapper is a greate canidate!

Specifically, the pipe wrapper works at the level of defining a specific Loader, where basicially you are requesting that the loader you want to use be a Pipeline of a few different loader options, where the loader options are ones compatible in passing input to each other, e.g., the output from fit_transform as called on the ROI extractor is valid input to fit_transform of the Timeseries creator, and lastly the output from fit_transform of the Timeseries creator valid 1D feature array per subjects output.

Consider the example in code below, where we assume that 'rois' is the ROI extractor, and 'timeseries' is the correlation matrix creator object (where these could be can valid loader str, or custom user passed objects)

```
loader = Loader(obj = Pipe(['rois', 'timeseries']))
```

We only passed arguments for obj above, but in our toy example as initially described we wanted to further define parameters for a parameter search across both objects. See below for what different options for passing corresponding parameter distributions are:

```
# Options loader1 and loader2 tell it explicitly no params

# Special case, if just default params = 0, will convert to 2nd case
loader1 = Loader(obj = Pipe(['rois', 'timeseries']),
                 params = 0)

# You can specify just a matching list
loader2 = Loader(obj = Pipe(['rois', 'timeseries']),
                 params = [0, 0])

# Option 3 assumes that there are pre-defined valid class param dists
# for each of the base objects
loader3 = Loader(obj = Pipe(['rois', 'timeseries']),
                 params = [1, 1])

# Option 4 lets set params for the 'rois' object, w/ custom param dists
loader4 = Loader(obj = Pipe(['rois', 'timeseries']),
                 params = [{'some custom param dist'}, 0])
```

Note that still only one scope may be passed, and that scope will define the scope of the new combined loader. Also note that if extra_params is passed, the same extra_params will be passed when creating both individual objects. Where extra params behavior is to add its contents, only when the name of that param appears in the base classes init, s.t. there could exist a case where, if both 'rois' and 'timeseries' base objects had a parameter with the same name, passing a value for that name in extra params would update them both with the passed value.

18.4 Value_Subset

class BPt.**Value_Subset**(*name*, *value*)

Value_Subset is special wrapper class for BPt designed to work with *Subjects* style input. As seen in *Param_Search*, or to the *train_subjects* or *test_subjects* params in *Evaluate* and *Test*.

This wrapper can be used as follows, just specify an object as

```
Value_Subset(name, value)
```

Where name is the name of a loaded Strat column / feature, and value is the subset of values from that column to select subjects by. E.g., if you wanted to select just subjects of a specific sex, and assuming a variable was loaded in Strat (See *Load_Strat*) you could pass:

```
subjects = Value_Subset('sex', 0)
```

Which would specify only subjects with 'sex' equal to 0. You may also pass a list-like set of multiple columns to the name param. In this case, the overlap across all passed names will be computed, for example:

```
subjects = Value_Subset(['sex', 'race'], 0)
```

Where 'race' is another valid loaded Strat, would select only subjects with a value of 0 in the computed unique overlap across 'sex' and 'race'.

Note it might be hard to tell what a value of 0 actually means, especially when you compose across multiple variables. With that in mind, as long as verbose is set to True, upon computation of the subset of subjects a message with be printed indicating what the passed value corresponds to in all of the combined variables, e.g., in the example above you would get the print out 'sex' = 0, 'race' = 0.

18.5 Values_Subset

class BPt.**Values_Subset**(*name*, *values*)

Value_Subsets is special wrapper class for BPt designed to work with *Subjects* style input.

This wrapper is very similar to `Value_Subject`, and will actually function the same in the case that one value for name and one value for values is selected, e.g. the below are equivilent.

```
subjects = Value_Subset(name='sex', value=0)
subjects = Values_Subset(name='sex', values=0)
```

That said, where Value_Subset, allows passing multiple values for name, but only allows one value for value, Values_Subset only allows one value for name, and multiple values for values.

Values_Subset therefore lets you select the subset of subjects via one or more values in a loaded Strat variable. E.g.,

```
subjects = Values_Subset(name='site', values=[0,1,5])
```

Would select the subset of subjects from sites 0, 1 and 5.

CHAPTER

NINETEEN

INIT PHASE

19.1 Import

To start, you must import the module. Assuming that it has been downloaded of course. Import and then make an object, in this example the obj is called "ML"

```
from BPt import BPt_ML
ML = BPt_ML(**init_params)
```

Alternatively, if you wish to load from an already saved object, you would do as follows

```
from BPt import Load
ML = Load(saved_location)
```

19.2 Load

BPt_ML.**Load**(*exp_name='default'*, *log_dr='default'*, *existing_log='default'*, *verbose='default'*, *notebook='default'*, *random_state='default'*)

This function is designed to load in a saved previously created BPt_ML object.

See *Save* for saving an object. See *Init* for the rest of changable param descriptions, e.g., log_dr, existing_log, ect...

Parameters **loc** (*str or Path*) – A path/str to a saved BPt_ML object, (One saved with *Save*), then that object will be loaded. Notably, if any additional params are passed along with it, e.g., exp_name, notebook, ect... they will override the saved values with the newly passed values. If left as 'default', all params will be set to the loaded value, though see the warning below.

> **Warning:** The exp_name or log_dr may need to be changed, especially in the case where the object is being loaded in a new location or enviroment from where the original was created, as it will by default try to create logs with the saved path information as the original.

You can only change exp_name, log_dr, existing_log, verbose, notebook and random_state when loading a new object, for the remaining params, even if a value is passed, it will not be applied. If the user really wishes to change one of these params, they can change it manually via self.name_of_param = whatever.

To init params as referenced above are those listed here under Init.

19.3 Init

class BPt.**BPt_ML**(*exp_name='My_Exp'*, *log_dr=''*, *existing_log='append'*, *verbose=True*, *notebook=True*, *use_abcd_subject_ids=False*, *low_memory_mode=False*, *strat_u_name='_Strat'*, *random_state=534*, *n_jobs=1*, *dpi=100*, *mp_context='loky'*)

Main class used within BPt for interfacing with Data Loading and Modeling / Other funcationality.

Parameters

- **exp_name** (*str, optional*) – The name of this experimental run, used explicitly in saving logs, and figures, where the passed *exp_name* is used as the name of the log folder. If log_dr is not set to None, (if not None then saves logs and figures) then a folder is created within the log dr with the exp_name.

  ```
  default = 'My_Exp'
  ```

- **log_dr** (*str, Path or None, optional*) – The directory in which to store logs... If set to None, then will not save any logs! If set to empty str, will save in the current dr.

  ```
  default = ''
  ```

- **existing_log** (*{'new', 'append', 'overwrite'}, optional*) – This parameter dictates different choices for when an a folder with exp_name already exists in the specified log_dr.

  These choices are:

  - **'new'** If the log folder already exists, then just increment *exp_name* until a free name is found, and use that as the log folder / *exp_name*.
  - **'append'** If existing_log is 'append' then log entries and new figures will be added to the existing folder.
  - **'overwrite'** If existing_log is 'overwrite', then the existing log folder with the same exp_name will be cleared upon __init__.

  ```
  default = 'append'
  ```

- **verbose** (*bool, optional*) – If *verbose* is set to True, the BPt_ML object will print output, diagnostic and more general, directly to std out. If set to False, no output will be printed, though output will still be recorded within the logs assuming log_dr is not None.

  ```
  default = True
  ```

- **notebook** (*bool, optional*) – If True, then assumes the user is running the code in an interactive jupyter notebook. In this case, certain features will either be enabled or disabled, e.g., type of progress bar.

  ```
  default = Trues
  ```

- **use_abcd_subject_ids** (*bool, optional*) – Flag to determine the usage of ABCD speficic 'default' subject id behavior. If set to True, this will convert input NDAR subject ids into upper case, with prepended NDAR - type format. If set to False, then all input subject names must be entered explicitly the same, no preprocessing will be done on them.

```
default = False
```

- **low_memory_mode** (*bool, optional*) – This parameter dictates behavior around loading in data, specifically, If set to True, individual dataframes self.data, self.covars ect... will be deleted from memory as soon as modeling begins. This parameter also controls the pandas read_csv behavior, which also has a low_memory flag.

```
default = False
```

- **strat_u_name** (*str, optional*) – A unique str identifier to be appended to every loaded strat value (to keep them seperate from covars and data).

  You should only need to change or ever worry about this in the case that one of your input variables happens to have the default value of '_Strat' in it...

```
default = '_Strat'
```

- **random_state** (*int, RandomState instance or None, optional*) – The default random state, either as int for a specific seed, or if None then the random seed is set by np.random. This parameters if set will be the default random_state class-wide, so any place random_state is left to default, unless a different default is set (e.g. default load value or default ML value) this random state will be used.

```
default = 534
```

- **n_jobs** (*int, optional*) – The default number of jobs / processors to use (if avaliable) where ever avaliable class-wide across the BPt.

```
default = 1
```

- **dpi** (*int, optional*) – The default dpi in which to save any automatically saved fiugres with. Where this parameter can also be set to specific values for specific plots.

```
default = 1
```

- **mp_context** (*str, optional*) – When a hyper-parameter search is launched, there are different ways through python that the multi-processing can be launched (assuming n_jobs > 1). Occassionally some choices can lead to unexpected errors.

  Choices are:

  - **'loky': Create and use the python library** loky backend.
  - 'fork': Python default fork mp_context
  - 'forkserver': Python default forkserver mp_context
  - 'spawn': Python default spawn mp_context

```
default = 'loky'
```

CHAPTER

TWENTY

LOADING PHASE

The next 'phase', is the where all of the loading is done, and the structure of the desired expiriments set up.

20.1 Set_Default_Load_Params

BPt_ML.**Set_Default_Load_Params**(*dataset_type='default'*, *subject_id='default'*, *eventname='default'*, *eventname_col='default'*, *overlap_subjects='default'*, *merge='default'*, *na_values='default'*, *drop_na='default'*, *drop_or_na='default'*)

This function is used to define default values for a series of params accessible to all or most of the different loading functions. By setting common values here, it reduces the need to repeat params within each loader (e.g. Load_Data, Load_Targets, ect...)

Parameters

- **dataset_type** (*{'basic', 'explorer', 'custom'}, optional*) – The dataset_type / file-type to load from. Dataset types are,
  - **'basic'** ABCD2p0NDA style (.txt and tab seperated). Typically the default columns, and therefore not neuroimaging data, will be dropped, also not including the eventname column.
  - **'explorer'** 2.0_ABCD_Data_Explorer style (.csv and comma seperated). The first 2 columns before self.subject_id (typically the default columns, and therefore not neuroimaging data - also not including the eventname column), will be dropped.
  - **'custom'** A user-defined custom dataset. Right now this is only. supported as a comma seperated file, with the subject names in a column called self.subject_id, and can optionally have 'eventname'. No columns will be dropped, (except eventname) or unless specific drop keys are passed.

  If loading multiple locs as a list, dataset_type can be a list with inds corresponding to which datatype for each loc.

  if 'default', and not already defined, set to 'basic'

  ```
  default = 'default'
  ```

- **subject_id** (*str, optional*) – The name of the column with unique subject ids in different dataset, for default ABCD datasets this is 'src_subject_id', but if a user wanted to load and work with a different dataset, they just need to change this accordingly (in addition to setting eventname most likely to None and use_abcd_subject_ids to False)

  if 'default', and not already defined, set to 'src_subject_id'.

```
default = 'default'
```

- **eventname** (*value, list of values or None, optional*) – Optional value to provide, specifying to optional keep certain rows when reading data based on the eventname flag, where eventname is the value and eventname_col is the name of the value.

  If a list of values are passed, then it will be treated as keeping a row if that row's value within the eventname_col is equal to ANY of the passed eventname values.

  As ABCD is a longitudinal study, this flag lets you select only one specific time point, or if set to None, will load everything.

  For selecting only baseline imagine data one might consider setting this param to 'baseline_year_1_arm_1'.

  if 'default', and not already defined, set to None. (default = 'default')

- **eventname_col** (*str or None, optional*) – If an eventname is provided, this param refers to the column name containing the eventname. This could also be used along with eventname to be set to any arbitrary value, in order to perform selection by specific column value.

  Note: The eventname col is dropped after proc'ed!

  if 'default', and not already defined, set to 'eventname' (default = 'default')

- **overlap_subjects** (*bool, optional*) – This parameter dictates when loading data, covars, targets or strat (after initial basic proc and/or merge w/ other passed loc's), if the loaded data should be restricted to only the overlapping subjects from previously loaded data, targets, covars or strat - important when performing intermediate proc. If False, then all subjects will be kept throughout the rest of the optional processing - and only merged at the end AFTER processing has been done.

  Note: Inclusions and Exclusions are always applied regardless of this parameter.

  if 'default', and not already defined, set to False (default = 'default')

- **merge** (*{'inner' or 'outer'}*) – Simmilar to overlap subjects, this parameter controls the merge behavior between different df's. i.e., when calling Load_Data twice, a local dataframe is merged with the class self.data on the second call. There are two behaviors that make sense here, one is 'inner' which says, only take the overlapping subjects from each dataframe, and the other is 'outer' which will keep all subjects from both, and set any missing subjects values to NaN.

  if 'default', and not already defined, set to 'inner' (default = 'default')

- **na_values** (*list, optional*) – Additional values to treat as NaN, by default ABCD specific values of '777' and '999' are treated as NaN, and those set to default by pandas 'read_csv' function. Note: if new values are passed here, it will override these default '777' and '999' NaN values, so if it desired to keep these, they should be passed explicitly, along with any new values.

  if 'default', and not already defined, set to ['777', '999'] (default = 'default')

- **drop_na** (*bool, int, float or 'default', optional*) – This setting sets the value for drop_na, which is used when loading data and covars only!

  If set to True, then will drop any row within the loaded data if there are any NaN! If False, the will not drop any rows for missing values.

  If an int or float, then this means some NaN entries will potentially be preserved! Missing data imputation will therefore be required later on!

If an int > 1, then will drop any row with more than drop_na NaN values. If a float, will determine the drop threshold as a percentage of the possible values, where 1 would not drop any rows as it would require the number of columns + 1 NaN, and .5 would require that more than half the column entries are NaN in order to drop that row.

if 'default', and not already defined, set to True (default = 'default')

- **drop_or_na** (*{'drop', 'na'}, optional*) – This setting sets the value for drop_na, which is used when loading data and covars only!

  filter_outlier_percent, or when loading a binary variable in load covars and more then two classes are present - are both instances where rows/subjects are by default dropped. If drop_or_na is set to 'na', then these values will instead be set to 'na' rather then the whole row dropped!

  Otherwise, if left as default value of 'drop', then rows will be dropped!

  if 'default', and not already defined, set to 'drop' (default = 'default')

20.2 Load_Name_Map

BPt_ML.**Load_Name_Map**(*name_map=None*, *loc=None*, *dataset_type='default'*, *source_name_col='NDAR name'*, *target_name_col='REDCap name/NDA alias'*, *na_values='default'*, *clear_existing=False*)

Loads a mapping dictionary for loading column names. Either a loc or name_map must be passed! Note: If both a name_map and loc are passed, the name_map will be loaded first, then updated with values from the loc.

Parameters

- **name_map** (*dict or None, optional*) – A dictionary containing the mapping to be passed directly. Set to None if using loc instead!

  (default = None)

- **loc** (*str, Path or None, optional*) – The location of the csv file which contains the mapping.

  (default = None)

- **dataset_type** (*{'basic', 'explorer', 'custom'}, optional*) – The dataset_type / file-type to load from. Dataset types are,

  - **'basic'** ABCD2p0NDA style (.txt and tab seperated). Typically the default columns, and therefore not neuroimaging data, will be dropped, also not including the eventname column.
  - **'explorer'** 2.0_ABCD_Data_Explorer style (.csv and comma seperated). The first 2 columns before self.subject_id (typically the default columns, and therefore not neuroimaging data - also not including the eventname column), will be dropped.
  - **'custom'** A user-defined custom dataset. Right now this is only. supported as a comma seperated file, with the subject names in a column called self.subject_id, and can optionally have 'eventname'. No columns will be dropped, (except eventname) or unless specific drop keys are passed.

  If loading multiple locs as a list, dataset_type can be a list with inds corresponding to which datatype for each loc.

  if 'default', and not already defined, set to 'basic'

```
default = 'default'
```

- **source_name_col** (*str, optional*) – The column name with the file which lists names to be changed.

  (default = "NDAR name")

- **target_name_col** (*str, optional*) – The column name within the file which lists the new name.

  (default = "REDCap name/NDA alias")

- **na_values** (*list, optional*) – Additional values to treat as NaN, by default ABCD specific values of '777' and '999' are treated as NaN, and those set to default by pandas 'read_csv' function. Note: if new values are passed here, it will override these default '777' and '999' NaN values, so if it desired to keep these, they should be passed explicitly, along with any new values.

  if 'default', and not already defined, set to ['777', '999'] (default = 'default')

- **clear_existing** (*bool, optional*) – If set to True, will clear the existing loaded name_map, otherwise the name_map dictionary will be updated if already loaded!

20.3 Load_Exclusions

BPt_ML.**Load_Exclusions**(*loc=None*, *subjects=None*, *clear_existing=False*)

Loads in a set of excluded subjects, from either a file or as directly passed in.

Parameters

- **loc** (*str, Path or None, optional*) – Location of a file to load in excluded subjects from. The file should be formatted as one subject per line. (default = None)
- **subjects** (*list, set, array-like or None, optional*) – An explicit list of subjects to add to exclusions. (default = None)
- **clear_existing** (*bool, optional*) – If this parameter is set to True, then any existing loaded exclusions will first be cleared before loading new exclusions!

  Warning: If any subjects have been dropped from a different place, e.g. targets or data, then simply reloading / clearing existing exclusions might result in computing a misleading overlap of final valid subjects. Reloading should therefore be best used right after loading the original exclusions, or if not possible, then reloading the notebook or re-running the script.

  (default = False)

Notes

For best/most reliable performance across all Loading cases, exclusions should be loaded before data, covars and targets.

If default subject id behavior is set to False, reading subjects from a exclusion loc might not function as expected.

20.4 Load_Inclusions

BPt_ML.**Load_Inclusions**(*loc=None*, *subjects=None*, *clear_existing=False*)

Loads in a set of subjects such that only these subjects can be loaded in, and any subject not as an inclusion is dropped, from either a file or as directly passed in.

If multiple inclusions are loaded, the final set of inclusions is computed as the union of all passed inclusions, not the intersection! In this way, inclusions acts more as an iterative whitelist.

Parameters

- **loc** (*str, Path or None, optional*) – Location of a file to load in inclusion subjects from. The file should be formatted as one subject per line. (default = None)
- **subjects** (*list, set, array-like or None, optional*) – An explicit list of subjects to add to inclusions. (default = None)
- **clear_existing** (*bool, optional*) – If this parameter is set to True, then any existing loaded inclusions will first be cleared before loading new inclusions!

  Warning: If any subjects have been dropped from a different place, e.g. targets or data, then simply reloading / clearing existing inclusions might result in computing a misleading overlap of final valid subjects. Reloading should therefore be best used right after loading the original inclusions, or if not possible, then reloading the notebook or re-running the script.

  (default = False)

Notes

For best/most reliable performance across all Loading cases, inclusions should be loaded before data, covars and targets.

If default subject id behavior is set to False, reading subjects from a inclusion loc might not function as expected.

20.5 Load_Data

BPt_ML.**Load_Data**(*loc=None*, *df=None*, *dataset_type='default'*, *drop_keys=None*, *inclusion_keys=None*, *subject_id='default'*, *eventname='default'*, *eventname_col='default'*, *overlap_subjects='default'*, *merge='default'*, *na_values='default'*, *drop_na='default'*, *drop_or_na='default'*, *filter_outlier_percent=None*, *filter_outlier_std=None*, *unique_val_drop=None*, *unique_val_warn=0.05*, *drop_col_duplicates=None*, *clear_existing=False*, *ext=None*)

Class method for loading ROI-style data, assuming all loaded columns are continuous / float datatype.

Parameters

- **loc** (*str Path, list of or None, optional*) – The location of the file to load data load from. If passed a list, then will load each loc in the list, and will assume them all to be of the same dataset_type if one dataset_type is passed, or if they differ in type, a list must be passed to dataset_type with the different types in order.

Note: some proc will be done on each loaded dataset before merging with the rest (duplicate subjects, proc for eventname ect...), but other dataset loading behavior won't occur until after the merge, e.g., dropping cols by key, filtering for outlier, ect...

(default = None)

- **df** (*pandas DataFrame or None, optional*) – This parameter represents the option for the user to pass in a raw custom dataframe. A loc and/or a df must be passed.

  When pasing a raw DataFrame, the loc and dataset_type param will be ignored, as those are for loading data from a file. Otherwise, it will be treated the same as if loading from a file, which means, there should be a column within the passed dataframe with subject_id, and e.g. if eventname params are passed, they will be applied along with any other proc. specified.

  (default = None)

- **dataset_type** (*{'basic', 'explorer', 'custom'}, optional*) – The dataset_type / file-type to load from. Dataset types are,

  – **'basic'** ABCD2p0NDA style (.txt and tab seperated). Typically the default columns, and therefore not neuroimaging data, will be dropped, also not including the eventname column.

  – **'explorer'** 2.0_ABCD_Data_Explorer style (.csv and comma seperated). The first 2 columns before self.subject_id (typically the default columns, and therefore not neuroimaging data - also not including the eventname column), will be dropped.

  – **'custom'** A user-defined custom dataset. Right now this is only. supported as a comma seperated file, with the subject names in a column called self.subject_id, and can optionally have 'eventname'. No columns will be dropped, (except eventname) or unless specific drop keys are passed.

  If loading multiple locs as a list, dataset_type can be a list with inds corresponding to which datatype for each loc.

  if 'default', and not already defined, set to 'basic'

```
default = 'default'
```

- **drop_keys** (*str, list or None, optional*) – A list of keys to drop columns by, where if any key given in a columns name, then that column will be dropped. If a str, then same behavior, just with one col. (Note: if a name mapping exists, this drop step will be conducted after renaming)

  (default = None)

- **inclusion_keys** (*str, list or None, optional*) – A list of keys in which to only keep a loaded data column if ANY of the passed inclusion_keys are present within that column name.

  If passed only with drop_keys will be proccessed second.

  (Note: if a name mapping exists, this drop step will be conducted after renaming)

  (default = None)

- **subject_id** (*str, optional*) – The name of the column with unique subject ids in different dataset, for default ABCD datasets this is 'src_subject_id', but if a user wanted to load and work with a different dataset, they just need to change this accordingly (in addition to setting eventname most likely to None and use_abcd_subject_ids to False)

  if 'default', and not already defined, set to 'src_subject_id'.

```
default = 'default'
```

- **eventname** (*value, list of values or None, optional*) – Optional value to provide, specifying to optional keep certain rows when reading data based on the eventname flag, where eventname is the value and eventname_col is the name of the value.

  If a list of values are passed, then it will be treated as keeping a row if that row's value within the eventname_col is equal to ANY of the passed eventname values.

  As ABCD is a longitudinal study, this flag lets you select only one specific time point, or if set to None, will load everything.

  For selecting only baseline imagine data one might consider setting this param to 'baseline_year_1_arm_1'.

  if 'default', and not already defined, set to None. (default = 'default')

- **eventname_col** (*str or None, optional*) – If an eventname is provided, this param refers to the column name containing the eventname. This could also be used along with eventname to be set to any arbitrary value, in order to perform selection by specific column value.

  Note: The eventname col is dropped after proc'ed!

  if 'default', and not already defined, set to 'eventname' (default = 'default')

- **overlap_subjects** (*bool, optional*) – This parameter dictates when loading data, covars, targets or strat (after initial basic proc and/or merge w/ other passed loc's), if the loaded data should be restricted to only the overlapping subjects from previously loaded data, targets, covars or strat - important when performing intermediate proc. If False, then all subjects will be kept throughout the rest of the optional processing - and only merged at the end AFTER processing has been done.

  Note: Inclusions and Exclusions are always applied regardless of this parameter.

  if 'default', and not already defined, set to False (default = 'default')

- **merge** (*{'inner' or 'outer'}*) – Simmilar to overlap subjects, this parameter controls the merge behavior between different df's. i.e., when calling Load_Data twice, a local dataframe is merged with the class self.data on the second call. There are two behaviors that make sense here, one is 'inner' which says, only take the overlapping subjects from each dataframe, and the other is 'outer' which will keep all subjects from both, and set any missing subjects values to NaN.

  if 'default', and not already defined, set to 'inner' (default = 'default')

- **na_values** (*list, optional*) – Additional values to treat as NaN, by default ABCD specific values of '777' and '999' are treated as NaN, and those set to default by pandas 'read_csv' function. Note: if new values are passed here, it will override these default '777' and '999' NaN values, so if it desired to keep these, they should be passed explicitly, along with any new values.

  if 'default', and not already defined, set to ['777', '999'] (default = 'default')

- **drop_na** (*bool, int, float or 'default', optional*) – This setting sets the value for drop_na, which is used when loading data and covars only!

  If set to True, then will drop any row within the loaded data if there are any NaN! If False, the will not drop any rows for missing values.

  If an int or float, then this means some NaN entries will potentially be preserved! Missing data imputation will therefore be required later on!

If an int > 1, then will drop any row with more than drop_na NaN values. If a float, will determine the drop threshold as a percentage of the possible values, where 1 would not drop any rows as it would require the number of columns + 1 NaN, and .5 would require that more than half the column entries are NaN in order to drop that row.

if 'default', and not already defined, set to True (default = 'default')

- **drop_or_na** (*{'drop', 'na'}, optional*) – This setting sets the value for drop_na, which is used when loading data and covars only!

  filter_outlier_percent, or when loading a binary variable in load covars and more then two classes are present - are both instances where rows/subjects are by default dropped. If drop_or_na is set to 'na', then these values will instead be set to 'na' rather then the whole row dropped!

  Otherwise, if left as default value of 'drop', then rows will be dropped!

  if 'default', and not already defined, set to 'drop' (default = 'default')

- **filter_outlier_percent** (*int, float, tuple or None, optional*) – *For float data only.* A percent of values to exclude from either end of the targets distribution, provided as either 1 number, or a tuple (% from lower, % from higher). set *filter_outlier_percent* to None for no filtering. If over 1 then treated as a percent, if under 1, then used directly.

  If drop_or_na == 'drop', then all rows/subjects with >= 1 value(s) found outside of the percent will be dropped. Otherwise, if drop_or_na = 'na', then any outside values will be set to NaN.

  (default = None)

- **filter_outlier_std** (*int, float, tuple or None, optional*) – *For float data only.* Determines outliers as data points within each column where their value is less than the mean of the column - *filter_outlier_std[0]* * the standard deviation of the column, and greater than the mean of the column + *filter_outlier_std[1]* * the standard deviation of the column.

  If a singler number is passed, that number is applied to both the lower and upper range. If a tuple with None on one side is passed, e.g. (None, 3), then nothing will be taken off that lower or upper bound.

  If drop_or_na == 'drop', then all rows/subjects with >= 1 value(s) found will be dropped. Otherwise, if drop_or_na = 'na', then any outside values will be set to NaN.

  (default = None)

- **unique_val_drop** (*int, float None, optional*) – This parameter allows you to drops columns within loaded data where there are under a certain threshold of unique values.

  The threshold is determined by the passed value as either a float for a percentage of the data, e.g., computed as unique_val_drop * len(data), or if passed a number greater then 1, then that number, where a ny column with less unique values then this threshold will be dropped.

  (default = None)

- **unique_val_warn** (*int or float, optional*) – This parameter is simmilar to unique_val_drop, but only warns about columns with under the threshold (see unique_val_drop for how the threshold is computed) unique vals.

  (default = .05)

- **drop_col_duplicates** (*float or None/False, optional*) – If set to None, will not drop any. If float, then pass a value between 0 and 1, where if two columns within data are correlated >= to *corr_thresh*, the second column is removed.

  A value of 1 will instead make a quicker direct ==’s comparison.

  Note: This param just drops duplicated within the just loaded data. You can call self.Drop_Data_Duplicates() to drop duplicates across all loaded data.

  Be advised, this functionality runs rather slow when there are ~500+ columns to compare!

  (default = None)

- **clear_existing** (*bool, optional*) – If this parameter is set to True, then any existing loaded data will first be cleared before loading new data!

  Warning: If any subjects have been dropped from a different place, e.g. targets, then simply reloading / clearing existing data might result in computing a misleading overlap of final valid subjects. Reloading should therefore be best used right after loading the original data, or if not possible, then reloading the notebook or re-running the script.

  (default = False)

- **ext** (*None or str, optional*) – Optional fixed extension to append to all loaded col names, leave as None to ignore this param. Note: applied after name mapping.

  (default = None)

20.6 Load_Data_Files

BPt_ML.**Load_Data_Files**(*loc=None*, *df=None*, *files=None*, *file_to_subject=None*, *load_func=<function load>*, *dataset_type='default'*, *drop_keys=None*, *inclusion_keys=None*, *subject_id='default'*, *eventname='default'*, *eventname_col='default'*, *overlap_subjects='default'*, *merge='default'*, *reduce_func=<function mean>*, *filter_outlier_percent=None*, *filter_outlier_std=None*, *clear_existing=False*, *ext=None*)

Class method for loading in data as file paths, where file paths correspond to some sort of raw data which should only be actually loaded / proc’ed within the actual modelling. The further assumption made is that these files represent ‘Data’ in the same sense that *Load_Data()* represents data, where once loaded / proc’ed (See *Loaders*), the outputted features should be continuous / float datatype.

Parameters

- **loc** (*str Path, list of or None, optional*) – The location of the file to load data load from. If passed a list, then will load each loc in the list, and will assume them all to be of the same dataset_type if one dataset_type is passed, or if they differ in type, a list must be passed to dataset_type with the different types in order.

  Note: some proc will be done on each loaded dataset before merging with the rest (duplicate subjects, proc for eventname ect...), but other dataset loading behavior won’t occur until after the merge, e.g., dropping cols by key, filtering for outlier, ect...

  (default = None)

- **df** (*pandas DataFrame or None, optional*) – This parameter represents the option for the user to pass in a raw custom dataframe. A loc and/or a df must be passed.

When pasing a raw DataFrame, the loc and dataset_type param will be ignored, as those are for loading data from a file. Otherwise, it will be treated the same as if loading from a file, which means, there should be a column within the passed dataframe with subject_id, and e.g. if eventname params are passed, they will be applied along with any other proc. specified.

(default = None)

- **files** (*dict, optional*) – Another alternative for specifying files to load can be done by passing a dict to this param.

  Warning: This option right now only works if all files to load are the same across each subject, e.g., no missing data for one modality. This will hopefully be fixed in the future, or atleast provide a better warning!

  Specifically, a python dictionary should be passed where each key refers to the name of that feature / column of data files to load, and the value is a python list, or array-like of str file paths.

  You must also pass a python function to the file_to_subject param, which specifies how to convert from passed file path, to a subject name.

  E.g., consider the example below, where 2 subjects files are loaded for 'feat1' and feat2':

```
files = {'feat1': ['f1/subj_0.npy', 'f1/subj_1.npy'],
         'feat2': ['f2/subj_0.npy', 'f2/subj_1.npy']}

def file_to_subject_func(file):
    subject = file.split('/')[1].replace('.npy', '')
    return subject

file_to_subject = file_to_subject_func
# or
file_to_subject = {'feat1': file_to_subject_func,
                   'feat2': file_to_subject_func}
```

  In this example, subjects are loaded as 'subj_0' and 'subj_1', and they have associated loaded data files 'feat1' and 'feat2'.

```
default = None
```

- **file_to_subject** (*python function, or dict of optional*) – If files is passed, then you also need to specify a function which takes in a file path, and returns the relevant subject for that file path. If just one function is passed, it will be used for to load all dictionary entries, alternatively you can pass a matching dictionary of funcs, allowing for different funcs for each feature to load.

  See the example in param *files*.

```
default = None
```

- **load_func** (*python function, optional*) – Data_Files represent a path to a saved file, which means you must also provide some information on how to load the saved file. This parameter is where that loading function should be passed. The passed *load_func* will be used on each Data_File individually and whatever the output of the function is will be passed to *loaders* directly in modelling.

  You might need to pass a user defined custom function in some cases, e.g., you want to use np.load, but then also np.stack. Just wrap those two functions in one, and pass the new function.

(default = np.load)

- **dataset_type** (*{'basic', 'explorer', 'custom'}, optional*) – The dataset_type / file-type to load from. Dataset types are,
  - **'basic'** ABCD2p0NDA style (.txt and tab seperated). Typically the default columns, and therefore not neuroimaging data, will be dropped, also not including the eventname column.
  - **'explorer'** 2.0_ABCD_Data_Explorer style (.csv and comma seperated). The first 2 columns before self.subject_id (typically the default columns, and therefore not neuroimaging data - also not including the eventname column), will be dropped.
  - **'custom'** A user-defined custom dataset. Right now this is only. supported as a comma seperated file, with the subject names in a column called self.subject_id, and can optionally have 'eventname'. No columns will be dropped, (except eventname) or unless specific drop keys are passed.

  If loading multiple locs as a list, dataset_type can be a list with inds corresponding to which datatype for each loc.

  if 'default', and not already defined, set to 'basic'

```
default = 'default'
```

- **drop_keys** (*str, list or None, optional*) – A list of keys to drop columns by, where if any key given in a columns name, then that column will be dropped. If a str, then same behavior, just with one col. (Note: if a name mapping exists, this drop step will be conducted after renaming)

  (default = None)

- **inclusion_keys** (*str, list or None, optional*) – A list of keys in which to only keep a loaded data column if ANY of the passed inclusion_keys are present within that column name.

  If passed only with drop_keys will be proccessed second.

  (Note: if a name mapping exists, this drop step will be conducted after renaming)

  (default = None)

- **subject_id** (*str, optional*) – The name of the column with unique subject ids in different dataset, for default ABCD datasets this is 'src_subject_id', but if a user wanted to load and work with a different dataset, they just need to change this accordingly (in addition to setting eventname most likely to None and use_abcd_subject_ids to False)

  if 'default', and not already defined, set to 'src_subject_id'.

```
default = 'default'
```

- **eventname** (*value, list of values or None, optional*) – Optional value to provide, specifying to optional keep certain rows when reading data based on the eventname flag, where eventname is the value and eventname_col is the name of the value.

  If a list of values are passed, then it will be treated as keeping a row if that row's value within the eventname_col is equal to ANY of the passed eventname values.

  As ABCD is a longitudinal study, this flag lets you select only one specific time point, or if set to None, will load everything.

  For selecting only baseline imagine data one might consider setting this param to 'baseline_year_1_arm_1'.

if 'default', and not already defined, set to None. (default = 'default')

- **eventname_col** (*str or None, optional*) – If an eventname is provided, this param refers to the column name containing the eventname. This could also be used along with eventname to be set to any arbitrary value, in order to perform selection by specific column value.

  Note: The eventname col is dropped after proc'ed!

  if 'default', and not already defined, set to 'eventname' (default = 'default')

- **overlap_subjects** (*bool, optional*) – This parameter dictates when loading data, covars, targets or strat (after initial basic proc and/or merge w/ other passed loc's), if the loaded data should be restricted to only the overlapping subjects from previously loaded data, targets, covars or strat - important when performing intermediate proc. If False, then all subjects will be kept throughout the rest of the optional processing - and only merged at the end AFTER processing has been done.

  Note: Inclusions and Exclusions are always applied regardless of this parameter.

  if 'default', and not already defined, set to False (default = 'default')

- **merge** (*{'inner' or 'outer'}*) – Simmilar to overlap subjects, this parameter controls the merge behavior between different df's. i.e., when calling Load_Data twice, a local dataframe is merged with the class self.data on the second call. There are two behaviors that make sense here, one is 'inner' which says, only take the overlapping subjects from each dataframe, and the other is 'outer' which will keep all subjects from both, and set any missing subjects values to NaN.

  if 'default', and not already defined, set to 'inner' (default = 'default')

- **reduce_func** (*python function or list of, optional*) – This function is used if either filter_outlier_percent or filter_outlier_std is requested.

  The passed python function should reduce the file, once loaded, to one number, making it comptabile with the different filtering strategies. For example, the default function is just to take the mean of each loaded file, and to compute outlier detection on the mean.

  You may also pass a list to reduce func, where each entry of the list is a single reduce func. In this case outlier filtering will be computed on each reduce_fun seperately, and the union of all subjects marked as outlier will be dropped at the end.

```
default = np.mean
```

- **filter_outlier_percent** (*int, float, tuple or None, optional*) – *For float data only.* A percent of values to exclude from either end of the targets distribution, provided as either 1 number, or a tuple (% from lower, % from higher). set *filter_outlier_percent* to None for no filtering. If over 1 then treated as a percent, if under 1, then used directly.

  If drop_or_na == 'drop', then all rows/subjects with >= 1 value(s) found outside of the percent will be dropped. Otherwise, if drop_or_na = 'na', then any outside values will be set to NaN.

  (default = None)

- **filter_outlier_std** (*int, float, tuple or None, optional*) – *For float data only.* Determines outliers as data points within each column where their value is less than the mean of the column - *filter_outlier_std[0]* * the standard deviation of the column, and greater than the mean of the column + *filter_outlier_std[1]* * the standard deviation of the column.

If a singler number is passed, that number is applied to both the lower and upper range. If a tuple with None on one side is passed, e.g. (None, 3), then nothing will be taken off that lower or upper bound.

If drop_or_na == 'drop', then all rows/subjects with >= 1 value(s) found will be dropped. Otherwise, if drop_or_na = 'na', then any outside values will be set to NaN.

(default = None)

- **clear_existing** (*bool, optional*) – If this parameter is set to True, then any existing loaded data will first be cleared before loading new data!

  Warning: If any subjects have been dropped from a different place, e.g. targets, then simply reloading / clearing existing data might result in computing a misleading overlap of final valid subjects. Reloading should therefore be best used right after loading the original data, or if not possible, then reloading the notebook or re-running the script.

  (default = False)

- **ext** (*None or str, optional*) – Optional fixed extension to append to all loaded col names, leave as None to ignore this param. Note: applied after name mapping.

  (default = None)

20.7 Drop_Data_Cols

BPt_ML.**Drop_Data_Cols**(*drop_keys=None*, *inclusion_keys=None*)

Function to drop columns within loaded data by drop_keys or inclusion_keys.

Parameters

- **drop_keys** (*str, list or None, optional*) – A list of keys to drop columns within loaded data by, where if ANY key given in a columns name, then that column will be dropped. If a str, then same behavior, just with one col.

  If passed along with inclusion_keys will be processed first.

  (Note: if a name mapping exists, this drop step will be conducted after renaming)

  (default = None)

- **inclusion_keys** (*str, list or None, optional*) – A list of keys in which to only keep a loaded data column if ANY of the passed inclusion_keys are present within that column name.

  If passed only with drop_keys will be proccessed second.

  (Note: if a name mapping exists, this drop step will be conducted after renaming)

  (default = None)

20.8 Filter_Data_Cols

BPt_ML.**Filter_Data_Cols**(*filter_outlier_percent=None*, *filter_outlier_std=None*, *overlap_subjects='default'*, *drop_or_na='default'*)

Perform filtering on all loaded data based on an outlier percent, either dropping outlier rows or setting specific

outliers to NaN.

Note, if overlap_subject is set to True here, only the overlap will be saved after proc within self.data.

Parameters

- **filter_outlier_percent** (*int, float, tuple or None*) – *For float data only.* A percent of values to exclude from either end of the targets distribution, provided as either 1 number, or a tuple (% from lower, % from higher). set *filter_outlier_percent* to None for no filtering. If over 1 then treated as a percent, if under 1, then used directly.

  If drop_or_na == 'drop', then all rows/subjects with >= 1 value(s) found outside of the percent will be dropped. Otherwise, if drop_or_na = 'na', then any outside values will be set to NaN.

  (default = None)

- **filter_outlier_std** (*int, float, tuple or None, optional*) – *For float data only.* Determines outliers as data points within each column where their value is less than the mean of the column - *filter_outlier_std[0]* * the standard deviation of the column, and greater than the mean of the column + *filter_outlier_std[1]* * the standard deviation of the column.

  If a single number is passed, that number is applied to both the lower and upper range. If a tuple with None on one side is passed, e.g. (None, 3), then nothing will be taken off that lower or upper bound.

  (default = None)

- **overlap_subjects** (*bool, optional*) – This parameter dictates when loading data, covars, targets or strat (after initial basic proc and/or merge w/ other passed loc's), if the loaded data should be restricted to only the overlapping subjects from previously loaded data, targets, covars or strat - important when performing intermediate proc. If False, then all subjects will be kept throughout the rest of the optional processing - and only merged at the end AFTER processing has been done.

  Note: Inclusions and Exclusions are always applied regardless of this parameter.

  if 'default', and not already defined, set to False (default = 'default')

- **drop_or_na** (*{'drop', 'na'}, optional*) – This setting sets the value for drop_na, which is used when loading data and covars only!

  filter_outlier_percent, or when loading a binary variable in load covars and more then two classes are present - are both instances where rows/subjects are by default dropped. If drop_or_na is set to 'na', then these values will instead be set to 'na' rather then the whole row dropped!

  Otherwise, if left as default value of 'drop', then rows will be dropped!

  if 'default', and not already defined, set to 'drop' (default = 'default')

20.9 Proc_Data_Unique_Cols

BPt_ML.**Proc_Data_Unique_Cols**(*unique_val_drop=None, unique_val_warn=0.05, overlap_subjects='default'*)

This function performs proccessing on all loaded data based on the number of unique values loaded within each column (allowing users to drop or warn!).

Note, if overlap_subjects is set to True here, only the overlap will be saved after proc within self.data.

Parameters

- **unique_val_drop** (*int, float None, optional*) – This parameter allows you to drops columns within loaded data where there are under a certain threshold of unique values.

  The threshold is determined by the passed value as either a float for a percentage of the data, e.g., computed as unique_val_drop * len(data), or if passed a number greater then 1, then that number, where a ny column with less unique values then this threshold will be dropped.

  (default = None)

- **unique_val_warn** (*int or float, optional*) – This parameter is simmilar to unique_val_drop, but only warns about columns with under the threshold (see unique_val_drop for how the threshold is computed) unique vals.

  (default = .05)

- **overlap_subjects** (*bool, optional*) – This parameter dictates when loading data, covars, targets or strat (after initial basic proc and/or merge w/ other passed loc's), if the loaded data should be restricted to only the overlapping subjects from previously loaded data, targets, covars or strat - important when performing intermediate proc. If False, then all subjects will be kept throughout the rest of the optional processing - and only merged at the end AFTER processing has been done.

  Note: Inclusions and Exclusions are always applied regardless of this parameter.

  if 'default', and not already defined, set to False (default = 'default')

20.10 Drop_Data_Duplicates

BPt_ML.**Drop_Data_Duplicates**(*corr_thresh*, *overlap_subjects='default'*)

Drop duplicates columns within self.data based on if two data columns are >= to a certain correlation threshold.

Note, if overlap_subjects is set to True here, only the overlap will be saved after proc within self.data.

Parameters

- **corr_thresh** (*float*) – A value between 0 and 1, where if two columns within self.data are correlated >= to *corr_thresh*, the second column is removed.

  A value of 1 will instead make a quicker direct ==’s comparison.

- **overlap_subjects** (*bool, optional*) – This parameter dictates when loading data, covars, targets or strat (after initial basic proc and/or merge w/ other passed loc's), if the loaded data should be restricted to only the overlapping subjects from previously loaded data, targets, covars or strat - important when performing intermediate proc. If False, then all subjects will be kept throughout the rest of the optional processing - and only merged at the end AFTER processing has been done.

  Note: Inclusions and Exclusions are always applied regardless of this parameter.

  if 'default', and not already defined, set to False (default = 'default')

20.11 Show_Data_Dist

BPt_ML.**Show_Data_Dist**(*data_subset='SHOW_ALL'*, *num_feats=20*, *feats='random'*, *reduce_func=None*, *frame_interval=500*, *plot_type='hist'*, *show_only_overlap=True*, *subjects=None*, *save=True*, *dpi='default'*, *save_name='data distribution'*, *random_state='default'*, *return_anim=False*)

This method displays some summary statistics about the loaded targets, as well as plots the distibution if possible.

Note: to display loaded data files, pass a fun to reduce_func, otherwise they will not be displayed.

Parameters

- **data_subset** (*'SHOW_ALL' or array-like, optional*) – 'SHOW_ALL' is reserved for showing the distributions of loaded data. You may also pass a list/array-like to specify specific a custom source of features to show.

  If self.all_data is already prepared, this data subset can also include any float type features loaded as covar or target.

  ```
  default = 'SHOW_ALL'
  ```

- **num_feats** (*int, optional*) – The number of features' distributions in which to view. Note: If too many are selected it may take a long time to render and/or consume a lot of memory!

  ```
  default = 20
  ```

- **feats** (*{'random', 'skew'}, optional*) – The features in which to display, if 'random' then will select *num_feats* random features to display. If 'skew', will show the top *num_feats* features by absolute skew.

  If 'skew' and subjects == 'both', will compute the top skewed features based on the training set.

  ```
  default = 'random'
  ```

- **reduce_func** (*python function or list of, optional*) – If a function is passed here, then data files will be loaded and reduced to 1 number according to the passed function. For example, the default function is just to take the mean of each loaded file, and to compute outlier detection on the mean.

  To not display data files, if any, then just keep reduce func as None

  ```
  default = None
  ```

- **frame_interval** (*int, optional*) – The number of milliseconds between each frame.

  ```
  default = 500
  ```

- **plot_type** (*{'bar', 'hist', 'kde'}*) – The type of base seaborn plot to generate for each datapoint. Either 'bar' for barplot, or 'hist' for seaborns dist plot, or 'kde' for just a kernel density estimate plot.

  ```
  default = 'hist'
  ```

- **show_only_overlap** (*bool, optional*) – If True, then displays only the distributions for valid overlapping subjects across data, covars, ect... otherwise, if False, shows the current loaded distribution as is.

  If subjects is set (anything but None), this param will be ignored.

  ```
  default = True
  ```

- **subjects** (*None, 'train', 'test', 'both' or array-like, optional*) – If not None, then plot only the subjects loaded as train_subjects, or as test subjects, of you can pass a custom list or array-like of subjects.

  If 'both', then will plot the train and test distributions seperately. Note: This only works for plot_type == 'hist' or 'kde'. Also take into account, specifying 'both' will show some different information, then the default settings.

  ```
  default = None
  ```

- **save** (*bool, optional*) – If the animation should be saved as a gif, True or False.

  ```
  default = True
  ```

- **dpi** (*int, 'default', optional*) – The dpi in which to save the distribution gif. If 'default' use the class default value.

  ```
  default = 'default'
  ```

- **save_name** (*str, optional*) – The name in which the gif should be saved under.

  ```
  default = 'data distribution'
  ```

- **random_state** (*'default', int or None*) – The random state in which to choose random features from. If 'default' use the class define value, otherwise set to the value passed. None for random.

  ```
  default = 'default'
  ```

- **return_anim** (*bool, optional*) – If True, return just the animation

  ```
  default = False
  ```

20.12 Load_Targets

BPt_ML.**Load_Targets**(*loc=None*, *df=None*, *col_name=None*, *data_type=None*, *dataset_type='default'*, *subject_id='default'*, *eventname='default'*, *eventname_col='default'*, *overlap_subjects='default'*, *merge='default'*, *na_values='default'*, *drop_na='default'*, *drop_or_na='default'*, *filter_outlier_percent=None*, *filter_outlier_std=None*, *categorical_drop_percent=None*, *float_bins=10*, *float_bin_strategy='uniform'*, *clear_existing=False*, *ext=None*)

Loads in targets, the outcome / variable(s) to predict.

Parameters

- **loc** (*str, Path or None, optional*) – The location of the file to load targets load from.

  Either loc or df must be set, but they both cannot be set!

(default = None)

- **df** (*pandas DataFrame or None, optional*) – This parameter represents the option for the user to pass in a raw custom dataframe. A loc and/or a df must be passed.

  When pasing a raw DataFrame, the loc and dataset_type param will be ignored, as those are for loading from a file. Otherwise, it will be treated the same as if loading from a file, which means, there should be a column within the passed dataframe with subject_id, and e.g. if eventname params are passed, they will be applied along with any other proc. specified.

  Either loc or df must be set, but they both cannot be set!

- **col_name** (*str, list, optional*) – The name(s) of the column(s) to load.

  Note: Must be in the same order as data types passed in. (default = None)

- **data_type** (*{'b', 'c', 'f', 'f2c', 'a'}, optional*) – The data types of the different columns to load, in the same order as the column names passed in. Shorthands for datatypes can be used as well.

  If a list is passed to col_name, then you can either supply one data_type to be applied to all passed cols, or a list with corresponding data types by index for each col_name passed.

  - **'binary' or 'b'** Binary input
  - **'categorical' or 'c'** Categorical input
  - **'float' or 'f'** Float numerical input
  - **'float_to_cat', 'f2c', 'float_to_bin' or 'f2b'** This specifies that the data should be loaded initially as float, then descritized to be a binned categorical feature.
  - **'auto' or 'a'** This specifies that the type should be automatically inferred. Current inference rules are: if 2 unique non-nan categories then binary, if pandas datatype category, then categorical, otherwise float.

  Datatypes are explained further in Notes.

  (default = None)

- **dataset_type** (*{'basic', 'explorer', 'custom'}, optional*) – The dataset_type / file-type to load from. Dataset types are,

  - **'basic'** ABCD2p0NDA style (.txt and tab seperated). Typically the default columns, and therefore not neuroimaging data, will be dropped, also not including the eventname column.
  - **'explorer'** 2.0_ABCD_Data_Explorer style (.csv and comma seperated). The first 2 columns before self.subject_id (typically the default columns, and therefore not neuroimaging data - also not including the eventname column), will be dropped.
  - **'custom'** A user-defined custom dataset. Right now this is only. supported as a comma seperated file, with the subject names in a column called self.subject_id, and can optionally have 'eventname'. No columns will be dropped, (except eventname) or unless specific drop keys are passed.

  If loading multiple locs as a list, dataset_type can be a list with inds corresponding to which datatype for each loc.

  if 'default', and not already defined, set to 'basic'

```
default = 'default'
```

- **subject_id** (*str, optional*) – The name of the column with unique subject ids in different dataset, for default ABCD datasets this is 'src_subject_id', but if a user wanted to load and work with a different dataset, they just need to change this accordingly (in addition to setting eventname most likely to None and use_abcd_subject_ids to False)

  if 'default', and not already defined, set to 'src_subject_id'.

```
default = 'default'
```

- **eventname** (*value, list of values or None, optional*) – Optional value to provide, specifying to optional keep certain rows when reading data based on the eventname flag, where eventname is the value and eventname_col is the name of the value.

  If a list of values are passed, then it will be treated as keeping a row if that row's value within the eventname_col is equal to ANY of the passed eventname values.

  As ABCD is a longitudinal study, this flag lets you select only one specific time point, or if set to None, will load everything.

  For selecting only baseline imagine data one might consider setting this param to 'baseline_year_1_arm_1'.

  if 'default', and not already defined, set to None. (default = 'default')

- **eventname_col** (*str or None, optional*) – If an eventname is provided, this param refers to the column name containing the eventname. This could also be used along with eventname to be set to any arbitrary value, in order to perform selection by specific column value.

  Note: The eventname col is dropped after proc'ed!

  if 'default', and not already defined, set to 'eventname' (default = 'default')

- **overlap_subjects** (*bool, optional*) – This parameter dictates when loading data, covars, targets or strat (after initial basic proc and/or merge w/ other passed loc's), if the loaded data should be restricted to only the overlapping subjects from previously loaded data, targets, covars or strat - important when performing intermediate proc. If False, then all subjects will be kept throughout the rest of the optional processing - and only merged at the end AFTER processing has been done.

  Note: Inclusions and Exclusions are always applied regardless of this parameter.

  if 'default', and not already defined, set to False (default = 'default')

- **merge** (*{'inner' or 'outer'}*) – Simmilar to overlap subjects, this parameter controls the merge behavior between different df's. i.e., when calling Load_Data twice, a local dataframe is merged with the class self.data on the second call. There are two behaviors that make sense here, one is 'inner' which says, only take the overlapping subjects from each dataframe, and the other is 'outer' which will keep all subjects from both, and set any missing subjects values to NaN.

  if 'default', and not already defined, set to 'inner' (default = 'default')

- **na_values** (*list, optional*) – Additional values to treat as NaN, by default ABCD specific values of '777' and '999' are treated as NaN, and those set to default by pandas 'read_csv' function. Note: if new values are passed here, it will override these default '777' and '999' NaN values, so if it desired to keep these, they should be passed explicitly, along with any new values.

  if 'default', and not already defined, set to ['777', '999'] (default = 'default')

- **drop_na** (*bool, int, float or 'default', optional*) – This setting sets the value for drop_na, which is used when loading data and covars only!

  If set to True, then will drop any row within the loaded data if there are any NaN! If False, the will not drop any rows for missing values.

  If an int or float, then this means some NaN entries will potentially be preserved! Missing data imputation will therefore be required later on!

  If an int > 1, then will drop any row with more than drop_na NaN values. If a float, will determine the drop threshold as a percentage of the possible values, where 1 would not drop any rows as it would require the number of columns + 1 NaN, and .5 would require that more than half the column entries are NaN in order to drop that row.

  if 'default', and not already defined, set to True (default = 'default')

- **drop_or_na** (*{'drop', 'na'}, optional*) – This setting sets the value for drop_na, which is used when loading data and covars only!

  filter_outlier_percent, or when loading a binary variable in load covars and more then two classes are present - are both instances where rows/subjects are by default dropped. If drop_or_na is set to 'na', then these values will instead be set to 'na' rather then the whole row dropped!

  Otherwise, if left as default value of 'drop', then rows will be dropped!

  if 'default', and not already defined, set to 'drop' (default = 'default')

- **filter_outlier_percent** (*float, tuple, list of or None, optional*) – For float datatypes only. A percent of values to exclude from either end of the target distribution, provided as either 1 number, or a tuple (% from lower, % from higher). set *filter_outlier_percent* to None for no filtering.

  For example, if passed (1, 1), then the bottom 1% and top 1% of the distribution will be dropped, the same as passing 1. Further, if passed (.1, 1), the bottom .1% and top 1% will be removed.

  A list of values can also be passed in the case that multiple col_names / targets are being loaded. In this case, the index should correspond. If a list is not passed then the same value is used for all targets.

  (default = None).

- **filter_outlier_std** (*int, float, tuple, None or list of, optional*) – For float datatypes only. Determines outliers as data points within each column (target distribution) where their value is less than the mean of the column - *filter_outlier_std[0]* * the standard deviation of the column, and greater than the mean of the column + *filter_outlier_std[1]* * the standard deviation of the column.

  If a single number is passed, that number is applied to both the lower and upper range. If a tuple with None on one side is passed, e.g. (None, 3), then nothing will be taken off that lower or upper bound.

  A list of values can also be passed in the case that multiple col_names / covars are being loaded. In this case, the index should correspond. If a list is not passed here, then the same value is used when loading all targets.

  (default = None)

- **categorical_drop_percent** (*float, list of or None, optional*) – Optional percentage threshold for dropping categories when loading categorical data. If a float is given, then a category will be dropped if it makes up less than that % of the data

points. E.g. if .01 is passed, then any datapoints with a category with less then 1% of total valid datapoints is dropped.

A list of values can also be passed in the case that multiple col_names / targets are being loaded. In this case, the index should correspond. If a list is not passed then the same value is used for all targets.

(default = None)

- **float_bins** (*int or list of, optional*) – If any columns are loaded as 'float_to_bin' or 'f2b' then input must be discretized into bins. This param controls the number of bins to create. As with other params, if one value is passed, it is applied to all columns, but if different values per column loaded are desired, a list of ints (with inds correponding) should be pased. For columns that are not specifed as 'f2b' type, anything can be passed in that list index spot as it will be igored.

  (default = 10)

- **float_bin_strategy** (*{'uniform', 'quantile', 'kmeans'}, optional*) – If any columns are loaded as 'float_to_bin' or 'f2b' then input must be discretized into bins. This param controls the strategy used to define the bins. Options are,

  – **'uniform'** All bins in each feature have identical widths.

  – **'quantile'** All bins in each feature have the same number of points.

  – **'kmeans'** Values in each bin have the same nearest center of a 1D k-means cluster.

  As with float_bins, if one value is passed, it is applied to all columns, but if different values per column loaded are desired, a list of choices (with inds correponding) should be pased.

  (default = 'uniform')

- **clear_existing** (*bool, optional*) – If this parameter is set to True, then any existing loaded targets will first be cleared before loading new targets!

  Warning: If any subjects have been dropped from a different place, e.g. covars or data, then simply reloading / clearing existing covars might result in computing a misleading overlap of final valid subjects. Reloading should therefore be best used right after loading the original data, or if not possible, then reloading the notebook or re-running the script.

  (default = False)

- **ext** (*None or str, optional*) – Optional fixed extension to append to all loaded col names, leave as None to ignore this param. Note: applied after name mapping.

  (default = None)

Notes

Targets can be either 'binary', 'categorical', or 'float',

- **binary** Targets are read in and label encoded to be 0 or 1, Will also work if passed column of unique string also, e.g. 'M' and 'F'.
- **categorical** Targets are treated as taking on one fixed value from a limited set of possible values.
- **float** Targets are read in as a floating point number, and optionally then filtered.

20.13 Binarize_Target

BPt_ML.**Binarize_Target**(*threshold=None*, *lower=None*, *upper=None*, *target=0*, *replace=True*, *merge='outer'*)

This function binarizes a loaded target variable, assuming that a float type target is loaded, otherwise this function will break!

Parameters

- **threshold** (*float or None, optional*) – Single binary threshold, where any value less than the threshold will be set to 0 and any value greater than or equal to the threshold will be set to 1. Leave threshold as None, and use lower and upper instead to 'cut' out a chunk of values in the middle.

  (default = None)

- **lower** (*float or None, optional*) – Any value that is greater than lower will be set to 1, and any value <= upper and >= lower will be dropped.

  If a value is set for lower, one cannot be set for threshold, and one must bet set for upper.

  (default = None)

- **upper** (*float or None, optional*) – Any value that is less than upper will be set to 0, and any value <= upper and >= lower will be dropped.

  If a value is set for upper, one cannot be set for threshold, and one must bet set for lower.

  (default = None)

- **target** (*int or str, optional*) – The loaded target in which to Binarize. This can be the int index, or the name of the target column. If only one target is loaded, just leave as default.

  (default = 0)

- **replace** (*bool, optional*) – If True, then replace the target to be binarized in place, otherwise if False, add the binarized version as a new target.

  (default = True)

- **merge** (*{'inner' or 'outer'}*) – This argument is used only when replace is False, and is further relevant only when upper and lower arguments are passed. If 'inner', then drop from the loaded target dataframe any subjects which do not overlap, if 'outer', then set any non-overlapping subjects data to NaN's.

  (default = 'outer')

20.14 Show_Targets_Dist

BPt_ML.**Show_Targets_Dist**(*targets='SHOW_ALL'*, *cat_show_original_name=True*, *show_only_overlap=True*, *subjects=None*, *show=True*, *cat_type='Counts'*, *return_display_dfs=False*)

This method displays some summary statistics about the loaded targets, as well as plots the distibution if possible.

Parameters

- **targets** (*str, int or list, optional*) – The single (str) or multiple targets (list), in which to display the distributions of. The str input 'SHOW_ALL' is reserved, and set to default, for showing the distributions of loaded targets.

  You can also pass the int index of the loaded target to show!

  (default = 'SHOW_ALL')

- **cat_show_original_name** (*bool, optional*) – If True, then when showing a categorical distribution (or binary) make the distr plot using the original names. Otherwise, use the internally used names.

  (default = True)

- **show_only_overlap** (*bool, optional*) – If True, then displays only the distributions for valid overlapping subjects across data, covars, ect... otherwise, if False, shows the current loaded distribution as is.

  (default = True)

- **subjects** (*None, 'train', 'test' or array-like, optional*) – If None, plot all subjects. If not None, then plot only the subjects loaded as train_subjects, or as test subjects, or you can pass a custom list or array-like of subjects.

  (default = None)

- **show** (*bool, optional*) – If True, then plt.show(), the matplotlib command will be called, and the figure displayed. On the other hand, if set to False, then the user can customize the plot as they desire. You can think of plt.show() as clearing all of the loaded settings, so in order to make changes, you can't call this until you are done.

  (default = True)

- **cat_type** (*{'Counts', 'Frequency'}, optional*) – If plotting a categorical variable (binary or categorical), plot the X axis as either by raw count or frequency.

  (default = 'Counts')

- **return_display_dfs** (*bool, optional*) – Optionally return the display df as a pandas df

  (default = False)

20.15 Load_Covars

BPt_ML.**Load_Covars**(*loc=None*, *df=None*, *col_name=None*, *data_type=None*, *dataset_type='default'*, *subject_id='default'*, *eventname='default'*, *eventname_col='default'*, *overlap_subjects='default'*, *merge='default'*, *na_values='default'*, *drop_na='default'*, *drop_or_na='default'*, *nan_as_class=False*, *code_categorical_as='depreciated'*, *filter_outlier_percent=None*, *filter_outlier_std=None*, *categorical_drop_percent=None*, *float_bins=10*, *float_bin_strategy='uniform'*, *clear_existing=False*, *ext=None*)

Load a covariate or covariates, type data.

Parameters

- **loc** (*str, Path or None, optional*) – The location of the file to load co-variates load from.

  Either loc or df must be set, but they both cannot be set!

(default = None)

- **df** (*pandas DataFrame or None, optional*) – This parameter represents the option for the user to pass in a raw custom dataframe. A loc and/or a df must be passed.

  When pasing a raw DataFrame, the loc and dataset_type param will be ignored, as those are for loading from a file. Otherwise, it will be treated the same as if loading from a file, which means, there should be a column within the passed dataframe with subject_id, and e.g. if eventname params are passed, they will be applied along with any other proc. specified.

  Either loc or df must be set, but they both cannot be set!

- **col_name** (*str or list, optional*) – The name(s) of the column(s) to load.

  Note: Must be in the same order as data types passed in.

  (default = None)

- **data_type** (*{'b', 'c', 'f', 'm', 'f2c'} or None, optional*) – The data types of the different columns to load, in the same order as the column names passed in. Shorthands for datatypes can be used as well.

  If a list is passed to col_name, then you can either supply one data_type to be applied to all passed cols, or a list with corresponding data types by index for each col_name passed.

  - **'binary' or 'b'** Binary input
  - **'categorical' or 'c'** Categorical input
  - **'float' or 'f'** Float numerical input
  - **'float_to_cat', 'f2c', 'float_to_bin' or 'f2b'** This specifies that the data should be loaded initially as float, then descritized to be a binned categorical feature.
  - **'multilabel' or 'm'** Multilabel categorical input

  Warning: If 'multilabel' datatype is specified, then the associated col name should be a list of columns, and will be assumed to be. For example, if loading multiple targets and one is multilabel, a nested list should be passed to col_name.

  (default = None)

- **dataset_type** (*{'basic', 'explorer', 'custom'}, optional*) – The dataset_type / file-type to load from. Dataset types are,

  - **'basic'** ABCD2p0NDA style (.txt and tab seperated). Typically the default columns, and therefore not neuroimaging data, will be dropped, also not including the eventname column.
  - **'explorer'** 2.0_ABCD_Data_Explorer style (.csv and comma seperated). The first 2 columns before self.subject_id (typically the default columns, and therefore not neuroimaging data - also not including the eventname column), will be dropped.
  - **'custom'** A user-defined custom dataset. Right now this is only. supported as a comma seperated file, with the subject names in a column called self.subject_id, and can optionally have 'eventname'. No columns will be dropped, (except eventname) or unless specific drop keys are passed.

  If loading multiple locs as a list, dataset_type can be a list with inds corresponding to which datatype for each loc.

  if 'default', and not already defined, set to 'basic'

```
default = 'default'
```

- **subject_id** (*str, optional*) – The name of the column with unique subject ids in different dataset, for default ABCD datasets this is 'src_subject_id', but if a user wanted to load and work with a different dataset, they just need to change this accordingly (in addition to setting eventname most likely to None and use_abcd_subject_ids to False)

  if 'default', and not already defined, set to 'src_subject_id'.

```
default = 'default'
```

- **eventname** (*value, list of values or None, optional*) – Optional value to provide, specifying to optional keep certain rows when reading data based on the eventname flag, where eventname is the value and eventname_col is the name of the value.

  If a list of values are passed, then it will be treated as keeping a row if that row's value within the eventname_col is equal to ANY of the passed eventname values.

  As ABCD is a longitudinal study, this flag lets you select only one specific time point, or if set to None, will load everything.

  For selecting only baseline imagine data one might consider setting this param to 'baseline_year_1_arm_1'.

  if 'default', and not already defined, set to None. (default = 'default')

- **eventname_col** (*str or None, optional*) – If an eventname is provided, this param refers to the column name containing the eventname. This could also be used along with eventname to be set to any arbitrary value, in order to perform selection by specific column value.

  Note: The eventname col is dropped after proc'ed!

  if 'default', and not already defined, set to 'eventname' (default = 'default')

- **overlap_subjects** (*bool, optional*) – This parameter dictates when loading data, covars, targets or strat (after initial basic proc and/or merge w/ other passed loc's), if the loaded data should be restricted to only the overlapping subjects from previously loaded data, targets, covars or strat - important when performing intermediate proc. If False, then all subjects will be kept throughout the rest of the optional processing - and only merged at the end AFTER processing has been done.

  Note: Inclusions and Exclusions are always applied regardless of this parameter.

  if 'default', and not already defined, set to False (default = 'default')

- **merge** (*{'inner' or 'outer'}*) – Simmilar to overlap subjects, this parameter controls the merge behavior between different df's. i.e., when calling Load_Data twice, a local dataframe is merged with the class self.data on the second call. There are two behaviors that make sense here, one is 'inner' which says, only take the overlapping subjects from each dataframe, and the other is 'outer' which will keep all subjects from both, and set any missing subjects values to NaN.

  if 'default', and not already defined, set to 'inner' (default = 'default')

- **na_values** (*list, optional*) – Additional values to treat as NaN, by default ABCD specific values of '777' and '999' are treated as NaN, and those set to default by pandas 'read_csv' function. Note: if new values are passed here, it will override these default '777' and '999' NaN values, so if it desired to keep these, they should be passed explicitly, along with any new values.

if 'default', and not already defined, set to ['777', '999'] (default = 'default')

- **drop_na** (*bool, int, float or 'default', optional*) – This setting sets the value for drop_na, which is used when loading data and covars only!

  If set to True, then will drop any row within the loaded data if there are any NaN! If False, the will not drop any rows for missing values.

  If an int or float, then this means some NaN entries will potentially be preserved! Missing data imputation will therefore be required later on!

  If an int > 1, then will drop any row with more than drop_na NaN values. If a float, will determine the drop threshold as a percentage of the possible values, where 1 would not drop any rows as it would require the number of columns + 1 NaN, and .5 would require that more than half the column entries are NaN in order to drop that row.

  if 'default', and not already defined, set to True (default = 'default')

- **drop_or_na** (*{'drop', 'na'}, optional*) – This setting sets the value for drop_na, which is used when loading data and covars only!

  filter_outlier_percent, or when loading a binary variable in load covars and more then two classes are present - are both instances where rows/subjects are by default dropped. If drop_or_na is set to 'na', then these values will instead be set to 'na' rather then the whole row dropped!

  Otherwise, if left as default value of 'drop', then rows will be dropped!

  if 'default', and not already defined, set to 'drop' (default = 'default')

- **nan_as_class** (*bool, or list of, optional*) – If True, then when data_type is categorical, instead of keeping rows with NaN (explicitly this parameter does not override drop_na, so to use this, drop_na must be set to not True). the NaN values will be treated as a unique category.

  A list of values can also be passed in the case that multiple col_names / covars are being loaded. In this case, the index should correspond. If a list is not passed here, then the same value is used when loading all covars.

```
default = False
```

- **code_categorical_as** (*'depreciated', optional*) – This parameter has been removed, please use transformers within the actual modelling to accomplish something simillar.

```
default = 'depreciated'
```

- **filter_outlier_percent** (*int, float, tuple, None or list of, optional*) – For float datatypes only. A percent of values to exclude from either end of the covars distribution, provided as either 1 number, or a tuple (% from lower, % from higher). set *filter_outlier_percent* to None for no filtering.

  For example, if passed (1, 1), then the bottom 1% and top 1% of the distribution will be dropped, the same as passing 1. Further, if passed (.1, 1), the bottom .1% and top 1% will be removed.

  A list of values can also be passed in the case that multiple col_names / covars are being loaded. In this case, the index should correspond. If a list is not passed here, then the same value is used when loading all covars.

  Note: If loading a variable with type 'float_to_cat' / 'float_to_bin', the outlier filtering will be performed before kbin encoding.

(default = None)

- **filter_outlier_std** (*int, float, tuple, None or list of, optional*) – For float datatypes only. Determines outliers as data points within each column where their value is less than the mean of the column - *filter_outlier_std[0]* * the standard deviation of the column, and greater than the mean of the column + *filter_outlier_std[1]* * the standard deviation of the column.

  If a single number is passed, that number is applied to both the lower and upper range. If a tuple with None on one side is passed, e.g. (None, 3), then nothing will be taken off that lower or upper bound.

  A list of values can also be passed in the case that multiple col_names / covars are being loaded. In this case, the index should correspond. If a list is not passed here, then the same value is used when loading all covars.

  Note: If loading a variable with type 'float_to_cat' / 'float_to_bin', the outlier filtering will be performed before kbin encoding.

  (default = None)

- **categorical_drop_percent** (*float, None or list of, optional*) – Optional percentage threshold for dropping categories when loading categorical data. If a float is given, then a category will be dropped if it makes up less than that % of the data points. E.g. if .01 is passed, then any datapoints with a category with less then 1% of total valid datapoints is dropped.

  A list of values can also be passed in the case that multiple col_names / covars are being loaded. In this case, the index should correspond. If a list is not passed here, then the same value is used when loading all covars.

  Note: percent in the name might be a bit misleading. For 1%, you should pass .01, for 10%, you should pass .1.

  If loading a categorical variable, this filtering will be applied before ordinally encoding that variable. If instead loading a variable with type 'float_to_cat' / 'float_to_bin', the outlier filtering will be performed after kbin encoding (as before then it is not categorical). This can yield gaps in the oridinal outputted values.

  (default = None)

- **float_bins** (*int or list of, optional*) – If any columns are loaded as 'float_to_bin' or 'f2b' then input must be discretized into bins. This param controls the number of bins to create. As with other params, if one value is passed, it is applied to all columns, but if different values per column loaded are desired, a list of ints (with inds corresponding) should be pased. For columns that are not specifed as 'f2b' type, anything can be passed in that list index spot as it will be igored.

  (default = 10)

- **float_bin_strategy** (*{'uniform', 'quantile', 'kmeans'}, optional*) – If any columns are loaded as 'float_to_bin' or 'f2b' then input must be discretized into bins. This param controls the strategy used to define the bins. Options are,

  – **'uniform'** All bins in each feature have identical widths.

  – **'quantile'** All bins in each feature have the same number of points.

  – **'kmeans'** Values in each bin have the same nearest center of a 1D k-means cluster.

As with float_bins, if one value is passed, it is applied to all columns, but if different values per column loaded are desired, a list of choices (with inds correponding) should be pased.

(default = 'uniform')

- **clear_existing** (*bool, optional*) – If this parameter is set to True, then any existing loaded covars will first be cleared before loading new covars!

  Warning: If any subjects have been dropped from a different place, e.g. targets or data, then simply reloading / clearing existing covars might result in computing a misleading overlap of final valid subjects. Reloading should therefore be best used right after loading the original data, or if not possible, then reloading the notebook or re-running the script.

  (default = False)

- **ext** (*None or str, optional*) – Optional fixed extension to append to all loaded col names, leave as None to ignore this param. Note: applied after name mapping.

  (default = None)

20.16 Show_Covars_Dist

BPt_ML.**Show_Covars_Dist**(*covars='SHOW_ALL'*, *cat_show_original_name=True*, *show_only_overlap=True*, *subjects=None*, *show=True*, *cat_type='Counts'*, *return_display_dfs=False*)

Plot a single or multiple covar distributions, along with outputting useful summary statistics.

Parameters

- **covars** (*str or list, optional*) – The single covar (str) or multiple covars (list), in which to display the distributions of. The str input 'SHOW_ALL' is reserved, and set to default, for showing the distributions of all loaded covars.

  (default = 'SHOW_ALL')

- **cat_show_original_name** (*bool, optional*) – If True, then when showing a categorical distribution (or binary) make the distr plot using the original names. Otherwise, use the internally used names.

  (default = True)

- **show_only_overlap** (*bool, optional*) – If True, then displays only the distributions for valid overlapping subjects across data, covars, ect... otherwise, shows the current loaded distribution as is.

  (default = True)

- **subjects** (*None, 'train', 'test' or array-like, optional*) – If not None, then plot only the subjects loaded as train_subjects, or as test subjects, of you can pass a custom list or array-like of subjects.

  (default = None)

- **show** (*bool, optional*) – If True, then plt.show(), the matplotlib command will be called, and the figure displayed. On the other hand, if set to False, then the user can customize the plot as they desire. You can think of plt.show() as clearing all of the loaded settings, so in order to make changes, you can't call this until you are done.

(default = True)

- **cat_type** (*{'Counts', 'Frequency'}, optional*) – If plotting a categorical variable (binary or categorical), plot the X axis as either by raw count or frequency.

  (default = 'Counts')

- **return_display_dfs** (*bool, optional*) – Optionally return the display df as a pandas df

  (default = False)

20.17 Load_Strat

BPt_ML.**Load_Strat**(*loc=None*, *df=None*, *col_name=None*, *dataset_type='default'*, *subject_id='default'*, *eventname='default'*, *eventname_col='default'*, *overlap_subjects='default'*, *binary_col=False*, *float_to_binary=False*, *float_col=False*, *float_bins=10*, *float_bin_strategy='uniform'*, *filter_outlier_percent=None*, *filter_outlier_std=None*, *categorical_drop_percent=None*, *na_values='default'*, *clear_existing=False*, *ext=None*)

Load stratification values from a file. See Notes for more details on what stratification values are.

Parameters

- **loc** (*str, Path or None, optional*) – The location of the file to load stratification vals load from.

  Either loc or df must be set, but they both cannot be set!

  (default = None)

- **df** (*pandas DataFrame or None, optional*) – This parameter represents the option for the user to pass in a raw custom dataframe. A loc and/or a df must be passed.

  When pasing a raw DataFrame, the loc and dataset_type param will be ignored, as those are for loading from a file. Otherwise, it will be treated the same as if loading from a file, which means, there should be a column within the passed dataframe with subject_id, and e.g. if eventname params are passed, they will be applied along with any other proc. specified.

  Either loc or df must be set, but they both cannot be set!

- **col_name** (*str or list, optional*) – The name(s) of the column(s) to load. Any datatype can be loaded with the exception of multilabel, but for float variables in particular, they should be specified with the *float_col* and corresponding *float_bins* and *float_bin_strategy* params. Noisy binary cols can also be specified with the *binary_col* param.

  (default = None)

- **dataset_type** (*{'basic', 'explorer', 'custom'}, optional*) – The dataset_type / file-type to load from. Dataset types are,

  - **'basic'** ABCD2p0NDA style (.txt and tab seperated). Typically the default columns, and therefore not neuroimaging data, will be dropped, also not including the eventname column.
  - **'explorer'** 2.0_ABCD_Data_Explorer style (.csv and comma seperated). The first 2 columns before self.subject_id (typically the default columns, and therefore not neuroimaging data - also not including the eventname column), will be dropped.

– **'custom'** A user-defined custom dataset. Right now this is only. supported as a comma seperated file, with the subject names in a column called self.subject_id, and can optionally have 'eventname'. No columns will be dropped, (except eventname) or unless specific drop keys are passed.

If loading multiple locs as a list, dataset_type can be a list with inds corresponding to which datatype for each loc.

if 'default', and not already defined, set to 'basic'

```
default = 'default'
```

- **subject_id** (*str, optional*) – The name of the column with unique subject ids in different dataset, for default ABCD datasets this is 'src_subject_id', but if a user wanted to load and work with a different dataset, they just need to change this accordingly (in addition to setting eventname most likely to None and use_abcd_subject_ids to False)

  if 'default', and not already defined, set to 'src_subject_id'.

```
default = 'default'
```

- **eventname** (*value, list of values or None, optional*) – Optional value to provide, specifying to optional keep certain rows when reading data based on the eventname flag, where eventname is the value and eventname_col is the name of the value.

  If a list of values are passed, then it will be treated as keeping a row if that row's value within the eventname_col is equal to ANY of the passed eventname values.

  As ABCD is a longitudinal study, this flag lets you select only one specific time point, or if set to None, will load everything.

  For selecting only baseline imagine data one might consider setting this param to 'baseline_year_1_arm_1'.

  if 'default', and not already defined, set to None. (default = 'default')

- **eventname_col** (*str or None, optional*) – If an eventname is provided, this param refers to the column name containing the eventname. This could also be used along with eventname to be set to any arbitrary value, in order to perform selection by specific column value.

  Note: The eventname col is dropped after proc'ed!

  if 'default', and not already defined, set to 'eventname' (default = 'default')

- **overlap_subjects** (*bool, optional*) – This parameter dictates when loading data, covars, targets or strat (after initial basic proc and/or merge w/ other passed loc's), if the loaded data should be restricted to only the overlapping subjects from previously loaded data, targets, covars or strat - important when performing intermediate proc. If False, then all subjects will be kept throughout the rest of the optional processing - and only merged at the end AFTER processing has been done.

  Note: Inclusions and Exclusions are always applied regardless of this parameter.

  if 'default', and not already defined, set to False (default = 'default')

- **binary_col** (*bool or list of, optional*) – Strat values are loaded as ordinal categorical, but there still exists the case where the user would like to load a binary set of values, and would like to ensure they are binary (filtering out all values but the top 2 most frequent).

This input should either be one boolean True False value, or a list of values corresponding the the length of col_name if col_name is a list.

If col_name is a list and only one value for binary_col is passed, then that value is applied to all loaded cols.

(default = False)

- **float_to_binary** (*False, int, (int, int), or list of*) – Strat values are loaded as ordinal categorical, but one could also want to load a float value, and force it to be binary via thresholding.

  If False is passed, or False within a list of values, this will be ignored. Otherwise, a single int can be passed in the case of one threshold when values lower than or equal should be converted to 0, and values > to 1. If a tuple of ints passed, that corresponds to the case of passing a lower and upper binary threshold.

  (default = False)

- **float_col** (*bool, or list or None, optional*) – Strat values are loaded as ordinal categorical, but one could also want to load a float value, and bin it into according to some strategy into ordinal categorical.

  This input should either be one boolean True False value, or a list of values corresponding the the length of col_name if col_name is a list.

  If col_name is a list and only one value for binary_col is passed, then that value is applied to all loaded cols.

  (default = None)

- **float_bins** (*int or list of, optional*) – If any float_col are set to True, then the float input must be discretized into bins. This param controls the number of bins to create. As with float_col, if one value is passed, it is applied to all columns, but if different values per column loaded are desired, a list of ints (with inds correponding) should be pased.

  (default = 10)

- **float_bin_strategy** (*{'uniform', 'quantile', 'kmeans'}, optional*) – If any float_col are set to True, then the float input must be discretized into bins. This param controls the strategy used to define the bins. Options are,

  – **'uniform'** All bins in each feature have identical widths.

  – **'quantile'** All bins in each feature have the same number of points.

  – **'kmeans'** Values in each bin have the same nearest center of a 1D k-means cluster.

  As with float_col and float_bins, if one value is passed, it is applied to all columns, but if different values per column loaded are desired, a list of choices (with inds correponding) should be pased.

  (default = 'uniform')

- **filter_outlier_percent** (*int, float, tuple, None or list of, optional*) – If any float_col are set to True, then you may perform float based outlier removal.

  A percent of values to exclude from either end of the covars distribution, provided as either 1 number, or a tuple (% from lower, % from higher). set *filter_outlier_percent* to None for no filtering.

For example, if passed (1, 1), then the bottom 1% and top 1% of the distribution will be dropped, the same as passing 1. Further, if passed (.1, 1), the bottom .1% and top 1% will be removed.

As with float_col and float_bins, if one value is passed, it is applied to all columns, but if different values per column loaded are desired, a list of choices (with inds correponding) should be pased.

Note: this filtering will be applied before binning.

(default = None)

- **filter_outlier_std** (*int, float, tuple, None or list of, optional*) – If any float_col are set to True, then you may perform float based outlier removal.

  Determines outliers as data points within each column where their value is less than the mean of the column - *filter_outlier_std[0]* * the standard deviation of the column, and greater than the mean of the column + *filter_outlier_std[1]* * the standard deviation of the column.

  If a single number is passed, that number is applied to both the lower and upper range. If a tuple with None on one side is passed, e.g. (None, 3), then nothing will be taken off that lower or upper bound.

  As with float_col and float_bins, if one value is passed, it is applied to all columns, but if different values per column loaded are desired, a list of choices (with inds correponding) should be pased.

  Note: this filtering will be applied before binning.

  (default = None)

- **categorical_drop_percent** (*float, None or list of, optional*) – Optional percentage threshold for dropping categories when loading categorical data (so for strat these are any column that are not specified as float or binary). If a float is given, then a category will be dropped if it makes up less than that % of the data points. E.g. if .01 is passed, then any datapoints with a category with less then 1% of total valid datapoints is dropped.

  A list of values can also be passed in the case that multiple col_names / strat vals are being loaded. In this case, the indices should correspond. If a list is not passed here, then the same value is used when loading all non float non binary strat cols.

  Note: if this is used with float col, then the outlier removal will be performed after the k-binning. If also provided filter_outlier_percent or std, that will be applied before binning.

  (default = None)

- **na_values** (*list, optional*) – Additional values to treat as NaN, by default ABCD specific values of '777' and '999' are treated as NaN, and those set to default by pandas 'read_csv' function. Note: if new values are passed here, it will override these default '777' and '999' NaN values, so if it desired to keep these, they should be passed explicitly, along with any new values.

  if 'default', and not already defined, set to ['777', '999'] (default = 'default')

- **clear_existing** (*bool, optional*) – If this parameter is set to True, then any existing loaded strat will first be cleared before loading new strat!

> **Warning:** If any subjects have been dropped from a different place, e.g. targets or data, then simply reloading / clearing existing strat might result in computing a misleading overlap of final valid subjects. Reloading should therefore be best used right after loading the original strat, or if not possible, then reloading the notebook or re-running the script.

(default = False)

- **ext** (*None or str, optional*) – Optional fixed extension to append to all loaded col names, leave as None to ignore this param. Note: applied after name mapping.

  (default = None)

Notes

Stratification values are categorical variables which are loaded for the purpose of defining custom validation behavior.

For example: Sex might be loaded here, and used later to ensure that any validation splits retain the same distribution of each sex. See *Define_Validation_Strategy()*, and some arguments within *Evaluate()* (sample_on and subjects_to_use).

For most relaible split behavior based off strat values, make sure to load strat values after data, targets and covars.

20.18 Show_Strat_Dist

BPt_ML.**Show_Strat_Dist**(*strat='SHOW_ALL'*, *cat_show_original_name=True*, *show_only_overlap=True*, *subjects=None*, *show=True*, *cat_type='Counts'*, *return_display_dfs=False*)

Plot a single or multiple strat distributions, along with outputting useful summary statistics.

Parameters

- **strat** (*str or list, optional*) – The single strat (str) or multiple strats (list), in which to display the distributions of. The str input 'SHOW_ALL' is reserved, and set to default, for showing the distributions of all loaded strat cols.

  (default = 'SHOW_ALL')

- **cat_show_original_name** (*bool, optional*) – If True, then make the distr. plot using the original names. Otherwise, use the internally used names.

  (default = True)

- **show_only_overlap** (*bool, optional*) – If True, then displays only the distributions for valid overlapping subjects across data, covars, ect... otherwise, shows the current loaded distribution as is.

  (default = True)

- **subjects** (*None, 'train', 'test' or array-like, optional*) – If not None, then plot only the subjects loaded as train_subjects, or as test subjects, of you can pass a custom list or array-like of subjects.

  (default = None)

- **show** (*bool, optional*) – If True, then plt.show(), the matplotlib command will be called, and the figure displayed. On the other hand, if set to False, then the user can customize the plot as they desire. You can think of plt.show() as clearing all of the loaded settings, so in order to make changes, you can't call this until you are done.

  (default = True)

- **cat_type** (*{'Counts', 'Frequency'}, optional*) – If plotting a categorical variable (binary or categorical), plot the X axis as either by raw count or frequency.

  (default = 'Counts')

- **return_display_dfs** (*bool, optional*) – Optionally return the display df as a pandas df

  (default = False)

20.19 Get_Overlapping_Subjects

BPt_ML.**Get_Overlapping_Subjects**()

This function will return the set of valid overlapping subjects currently loaded across data, targets, covars, strat ect... respecting any inclusions and exclusions.

Returns The set of valid overlapping subjects.

Return type set

20.20 Clear_Name_Map

BPt_ML.**Clear_Name_Map**()

Reset name mapping

20.21 Clear_Exclusions

BPt_ML.**Clear_Exclusions**()

Resets exclusions to be an empty set.

> **Warning:** If any subjects have been dropped from a different place, e.g. targets or data, then simply reloading / clearing existing exclusions might result in computing a misleading overlap of final valid subjects. Reloading should therefore be best used right after loading the original exclusions, or if not possible, then reloading the notebook or re-running the script.

20.22 Clear_Data

BPt_ML.**Clear_Data**()

Resets any loaded data.

> **Warning:** If any subjects have been dropped from a different place, e.g. targets, then simply clearing data might result in computing a misleading overlap of final valid subjects. Reloading should therefore be best used right after loading the original data, or if not possible, then reloading the notebook or re-running the script.

20.23 Clear_Targets

BPt_ML.**Clear_Targets**()
Resets targets

20.24 Clear_Covars

BPt_ML.**Clear_Covars**()
Reset any loaded covars.

> **Warning:** If any subjects have been dropped from a different place, e.g. targets or data, then simply reloading / clearing existing covars might result in computing a misleading overlap of final valid subjects. Reloading should therefore be best used right after loading the original covars, or if not possible, then reloading the notebook or re-running the script.

20.25 Clear_Strat

BPt_ML.**Clear_Strat**()
Reset any loaded strat

> **Warning:** If any subjects have been dropped from a different place, e.g. targets or data, then simply reloading / clearing existing strat might result in computing a misleading overlap of final valid subjects. Reloading should therefore be best used right after loading the original strat, or if not possible, then reloading the notebook or re-running the script.

20.26 Get_Nan_Subjects

BPt_ML.**Get_Nan_Subjects**()
Retrieves all subjects with any loaded NaN data, returns their pandas index.

CHAPTER

TWENTYONE

VALIDATION PHASE

21.1 Define_Validation_Strategy

BPt_ML.**Define_Validation_Strategy**(*cv=None*, *groups=None*, *stratify=None*, *train_only_loc=None*, *train_only_subjects=None*, *show=True*, *show_original=True*, *return_df=False*)

Define a validation strategy to be used during different train/test splits, in addition to model selection and model hyperparameter cross validation. See Notes for more info.

Note, can also pass a cv params objects here.

Parameters

- **cv** (*CV* or None, optional) – If None, then skip, otherwise can pass a *CV* object here, and the rest of the parameters will be skipped.

  ```
  default = None
  ```

- **groups** (*str, list or None, optional*) – In the case of str input, will assume the str to refer to a column key within the loaded strat data, and will assign it as a value to preserve groups by during any train/test or K-fold splits. If a list is passed, then each element should be a str, and they will be combined into all unique combinations of the elements of the list.

  ```
  default = None
  ```

- **stratify** (*str, list or None, optional*) – In the case of str input, will assume the str to refer to a column key within the loaded strat data, or a loaded target col., and will assign it as a value to preserve distribution of groups by during any train/test or K-fold splits. If a list is passed, then each element should be a str, and they will be combined into all unique combinations of the elements of the list.

  Any target_cols passed must be categorical or binary, and cannot be float. Though you can consider loading in a float target as a strat, which will apply a specific k_bins, and then be valid here.

  In the case that you have a loaded strat val with the same name as your target, you can distinguish between the two by passing either the raw name, e.g., if they are both loaded as 'Sex', passing just 'Sex', will try to use the loaded target. If instead you want to use your loaded strat val with the same name - you have to pass 'Sex' + self.self.strat_u_name (by default this is '_Strat').

  ```
  default = None
  ```

- **train_only_loc** (*str, Path or None, optional*) – Location of a file to load in train_only subjects, where any subject loaded as train_only will be assigned to every training fold, and never to a testing fold. This file should be formatted as one subject per line.

  You can load from a loc and pass subjects, the subjects from each source will be merged.

  This parameter is compatible with groups / stratify.

  ```
  default = None
  ```

- **train_only_subjects** (*set, array-like, 'nan', or None, optional*) – An explicit list or array-like of train_only subjects, where any subject loaded as train_only will be assigned to every training fold, and never to a testing fold.

  You can also optionally specify 'nan' as input, which will add all subjects with any NaN data to train only.

  If you want to add both all the NaN subjects and custom subjects, call *Get_Nan_Subjects()* to get all NaN subjects, and then merge them yourself with any you want to pass.

  You can load from a loc and pass subjects, the subjects from each source will be merged.

  This parameter is compatible with groups / stratify.

  ```
  default = None
  ```

- **show** (*bool, optional*) – By default, if True, information about the defined validation strategy will be shown, including a dataframe if stratify is defined.

  ```
  default = True
  ```

- **show_original** (*bool, optional*) – By default when you define stratifying behavior, a dataframe will be displayed. This param controls if that dataframe shows original names, or if False, then it shows the internally used names.

  ```
  default = True
  ```

- **return_df** (*bool, optional*) – If set to true, then will return as dataframe version of the defined validation strategy. Note: this will return None in all cases execpt for when stratifying by a variable is requested!

  ```
  default = False
  ```

Notes

Validation strategy choices are explained in more detail:

- **Random** Just make validation splits randomly.
- **Group Preserving** Make splits that ensure subjects that are part of specific group are all within the same fold e.g., split by family, so that people with the same family id are always a part of the same fold.
- **Stratifying** Make splits such that the distribution of a given group is as equally split between two folds as possible, so simmilar to matched halves or e.g., in a binary or categorical predictive context, splits could be done to ensure roughly equal distribution of the dependent class.

For now, it is possible to define only one overarching strategy (One could imagine combining group preserving splits while also trying to stratify for class, but the logistics become more complicated). Though, within one strategy it is certainly possible to provide multiple values e.g., for stratification you can stratify by target (the dependent variable to be predicted) as well as say sex, though with addition of unique value, the size of the smallest unique group decreases.

21.2 Train_Test_Split

BPt_ML.**Train_Test_Split**(*test_size=None*, *test_subjects=None*, *cv='default'*, *random_state='default'*, *test_loc='depreciated'*, *CV='depreciated'*)

Define the overarching train / test split, *highly reccomended*.

Parameters

- **test_size** (*float, int or None, optional*) – If float, should be between 0.0 and 1.0 and represent the proportion of the dataset to be included in the test split. If int, represents the absolute number (or target number) to include in the testing group. Keep as None if using test_subjects.

```
default = None
```

- **test_subjects** (*Subjects*, optional) – Pass in a *Subjects* (see for more info) formatted input. This will define an explicit set of subjects to use as a test set. If anything but None is passed here, nothing should be passed to the test_size parameter.

```
default = None
```

- **cv** ('default' or *CV*, optional) – If left as default 'default', use the class defined CV for the train test split, otherwise can pass custom behavior

```
default = 'default'
```

- **random_state** (*int None or 'default', optional*) –

  If using test_size, then can optionally provide a random state, in order to be able to recreate an exact test set.

  If set to default, will use the value saved in self.random_state, (as set in *BPt.BPt_ML* upon class init).

```
default = 'default'
```

  test_loc [depreciated] Pass a single str with the test loc to test_subjects instead.

```
default = 'depreciated'
```

  CV ['depreciated'] Switching to passing cv parameter as cv instead of CV. For now if CV is passed it will still work as if it were passed as cv.

```
default = 'depreciated'
```

CHAPTER

TWENTYTWO

MODELING PHASE

22.1 Set_Default_ML_Verbosity

BPt_ML.**Set_Default_ML_Verbosity**(*save_results='default'*, *progress_bar='default'*, *progress_loc='default'*, *pipeline_verbose='default'*, *best_params_score='default'*, *compute_train_score='default'*, *show_init_params='default'*, *fold_name='default'*, *time_per_fold='default'*, *score_per_fold='default'*, *fold_sizes='default'*, *best_params='default'*, *save_to_logs='default'*, *flush='default'*)

This function allows setting various verbosity options that effect output during *Evaluate()* and *Test()*.

Parameters

- **save_results** (*bool, optional*) – If True, all results returned by Evaluate will be saved within the log dr (if one exists!), under run_name + .eval, and simmilarly for results returned by Test, but as run_name + .test.

  if 'default', and not already defined, set to False.

```
default = 'default'
```

- **progress_bar** (*bool, optional*) – If True, a progress bar, implemented in the python library tqdm, is used to show progress during use of *Evaluate()* , If False, then no progress bar is shown. This bar should work both in a notebook env and outside one, assuming self.notebook has been set correctly.

  if 'default', and not already defined, set to True.

```
default = 'default'
```

- **progress_loc** (*str, Path or None, optional*) – If not None, then this will record the progress of each Evaluate / Test call in this location.

  if 'default', and not already defined, set to False.

```
default = 'default'
```

- **pipeline_verbose** (*bool, optional*) – This controls the verbose parameter for the pipeline object itself. If set to True, then time elapsed while fitting each step will be printed.

  if 'default', and not already defined, set to False.

```
default = 'default'
```

- **compute_train_score** (*bool, optional*) – If True, then metrics/scorers and raw preds will also be computed on the training set in addition to just the eval or testing set.

  if 'default', and not already defined, set to False.

```
default = 'default'
```

- **show_init_params** (*bool, optional*) – If True, then print/show the parameters used before running Evaluate / Test. If False, then don't print the params used.

  if 'default', and not already defined, set to True.

```
default = 'default'
```

- **fold_name** (*bool, optional*) – If True, prints a rough measure of progress via printing out the current fold (somewhat redundant with the progress bar if used, except if used with other params, e.g. time per fold, then it is helpful to have the time printed with each fold). If False, nothing is shown.

  if 'default', and not already defined, set to False.

```
default = 'default'
```

- **time_per_fold** (*bool, optional*) – If True, prints the full time that a fold took to complete.

  if 'default', and not already defined, set to False.

```
default = 'default'
```

- **score_per_fold** (*bool, optional*) – If True, displays the score for each fold, though slightly less formatted then in the final display.

  if 'default', and not already defined, set to False.

```
default = 'default'
```

- **fold_sizes** (*bool, optional*) – If True, will show the number of subjects within each train and val/test fold.

  if 'default', and not already defined, set to False.

```
default = 'default'
```

- **best_params** (*bool, optional*) – If True, print the best search params found after every param search.

  if 'default', and not already defined, set to False.

```
default = 'default'
```

- **save_to_logs** (*bool, optional*) – If True, then when possible, and with the selected model verbosity options, verbosity ouput will be saved to the log file.

  if 'default', and not already defined, set to False.

```
default = 'default'
```

- **flush** (*bool, optional*) – If True, then add flush=True to all ML prints, which adds a call to flush the std output.

  if 'default', and not already defined, set to False.

```
default = False
```

22.2 Evaluate

BPt_ML.**Evaluate**(*model_pipeline*, *problem_spec='default'*, *splits=3*, *n_repeats=2*, *cv='default'*, *train_subjects='train'*, *feat_importances=None*, *return_raw_preds=False*, *return_models=False*, *run_name='default'*, *only_fold=None*, *base_dtype='float32'*, *CV='depreciated'*)

The Evaluate function is one of the main interfaces for building and evaluating *Model_Pipeline* on the loaded data. Specifically, Evaluate is designed to try and estimate the out of sample performance of a passed *Model_Pipeline* on a specific ML task (as specified by *Problem_Spec*). This estimate is done through a defined CV strategy (*splits* and *n_repeats*). While Evaluate's ideal usage is an expirimental context for exploring different choices of *Model_Pipeline* and then ultimately with *Test* - if used carefully (i.e., dont try 50 Pipelines's and only report the one that does best), it can be used on a full dataset.

Parameters

- **model_pipeline** (*Model_Pipeline*) – The passed *model_pipeline* should be an instance of the BPt params class *Model_Pipeline*. This object defines the underlying model pipeline to be evaluated.

  See *Model_Pipeline* for more information / how to create a the model pipeline.

- **problem_spec** (*Problem_Spec* or 'default', optional) – *problem_spec* accepts an instance of the BPt.BPt_ML params class *Problem_Spec*. This object is essentially a wrapper around commonly used parameters needs to define the context the model pipeline should be evaluated in. It includes parameters like problem_type, scorer, n_jobs, random_state, etc... See *Problem_Spec* explicitly for more information and for how to create an instance of this object.

  If left as 'default', then will just initialize a Problem_Spec with default params.

```
default = 'default'
```

- **splits** (*int, float, str or list of str, optional*) – In every fold of the defined CV strategy, the passed *model_pipeline* will be fitted on a train fold, and evaluated on a validation fold. This parameter controls the type of CV, i.e., specifies what the train and validation folds should be. These splits are further determined by the subjects passed to *train_subjects*. Notably, the splits defined will respect any special split behavior as defined in *Define_Validation_Strategy*.

  Specifically, options for split are:

  – **int** The number of k-fold splits to conduct. (E.g., 3 for a 3-fold CV).

  – **float** Must be 0 < *splits* < 1, and defines a single train-test like split, with *splits* as the % of the current training data size used as a validation/test set.

  – **str** If a str is passed, then it must correspond to a loaded Strat variable. In this case, a leave-out-group CV will be used according to the value of the indicated Strat variable (E.g., a leave-out-site CV scheme).

– **list of str** If multiple str passed, first determine the overlapping unique values from their corresponing loaded Strat variables, and then use this overlapped value to define the leave-out-group CV as described above.

Note that this defines only the base CV strategy, and that the following param *n_repeats* is optionally used to replicate this base strategy, e.g., for a twice repeated train-test split evaluation. Note further that *n_repeats* will work with any of these options, but say in the case of a leave out group CV, it would be awfully redundant, versus, with a passed float value, very reasonable.

```
default = 3
```

- **n_repeats** (*int, optional*) – Given the base CV defined / described in the *splits* param, this parameter further controls if the defined train/val splits should be repeated (w/ different random splits in all cases but the leave-out-group passed str option).

  For example, if *n_repeats* is set to 2, and *splits* is 3, then a twice repeated 3-fold CV will be performed, and results returned with respect to this strategy.

  It can be a good idea to set multiple *n_repeats* (assuming enough computation power), as it can help you spot cases where you may not have enough training subjects to get stable behavior, e.g., say you run a three times repeated 3 fold CV, if the mean validation scores from each 3-fold are all very close to each other, then you know that 1 repeat is likely enough. If instead the macro std in score (the std from in this case those 3 scores) is high, then it indicates you may not have enough subjects to get stable results from just one 3-fold CV, and that you might want to consider changing some settings.

  ```
  default = 2
  ```

- **cv** (*'default' or CV params object, optional*) – If left as default 'default', use the class defined CV behavior for the splits, otherwise can pass custom behavior

  ```
  default = 'default'
  ```

- **train_subjects** (*Subjects*, optional) – This parameter determines the set of training subjects which are used in this call to *Evaluate*. Note, this parameter is distinct to the *subjects* parameter within *Problem_Spec*, which is applied after selecting the subset of *train_subjects* specified here. These subjects are used as the input to *Evaluate*, i.e., so typically any subjects data you want to remain untouched (say your global test subjects) are considered within *Evaluate*, and only those explicitly passed here are.

  By default, this value will be set to the special str indiciator 'train', which specifies that the full set of globally defined training subjects (See: `Define_Train_Test_Split()`), should be used. Other special str indicators include 'all' to select all subjects, and 'test' to select the test set subjects.

  If *subjects* is passed a str, and that str is not one of the str indicators listed above, then it will be interpretted as the location of file in which to read subjects from (assuming one subjects per line).

  subjects may also be a custom array-like of subjects to use.

  See *Subjects* for how to correctly format input and for other special options.

  ```
  default = 'train'
  ```

- **feat_importances** (*Feat_Importance* list of, str or None, optional) – If passed None, by default, no feature importances will be saved.

Alternatively, one may pass the keyword 'base', to indicate that the base feature importances - those automatically calculated by base objects (e.g., beta weights from linear models) be saved. In this case, the object Feat_Importance('base') will be made.

Otherwise, for more detailed control provide here either a single, or list of *Feat_Importance* param objects in which to specify what importance values, and with what settings should be computed. See the base *Feat_Importance* object for more information on how to specify these objects.

See *Feat Importances* to learn more about feature importances generally.

In this case of a passed list, all passed Feat_Importances will attempt to be computed.

```
default = None
```

- **return_raw_preds** (*bool, optional*) – If True, return the raw predictions from each fold.

```
default = False
```

- **return_models** (*bool, optional*) – If True, return the trained models from each evaluation.

```
default = False
```

- **run_name** (*str or 'default', optional*) – Each run of Evaluate can be optionally associated with a specific *run_name*. This name is used if *save_results* in *Set_Default_ML_Verbosity* is set to True, then will be used as the name output from Evaluate as saved as in the specific log_dr (if any, and as set when Init'ing the *BPt_ML* class object), with '.eval' appended to the name.

  If left as 'default', will come up with a kind of terrible name passed on the underlying model used in the passed *model_pipeline*.

```
default = 'default'
```

- **only_fold** (*int or None, optional*) – This is a special parameter used to only Evaluate a specific fold of the specified runs to evaluate. Keep as None to ignore.

```
default = None
```

- **base_dtype** (*numpy dtype*) – The dataset is cast to a numpy array of float. This parameter can be used to change the default behavior, e.g., if more resolution or less is needed.

```
default = 'float32'
```

- **CV** (*'depreciated'*) – Switching to passing cv parameter as cv instead of CV. For now if CV is passed it will still work as if it were passed as cv.

```
default = 'depreciated'
```

Returns **results** – Dictionary containing: 'summary_scores', A list representation of the printed summary scores, where the 0 index is the mean, 1 index is the macro std, then second index is the micro std. 'train_summary_scores', Same as summary scores, but only exists if train scores are computed. 'raw_scores', a numpy array of numpy arrays, where each internal array contains the raw scores as computed for all passed in scorers, computed for each fold within each repeat. e.g., array will have a length of *n_repeats* * number of folds, and each internal array will have

the same length as the number of scorers. Optionally, this could instead return a list containing as the first element the raw training score in this same format, and then the raw testing scores. 'raw_preds', A pandas dataframe containing the raw predictions for each subject, in the test set, and 'FIs' a list where each element corresponds to a passed feature importance.

Return type dict

Notes

Prints by default the following for each scorer,

float The mean macro score (as set by input scorer) across each repeated K-fold.

float The standard deviation of the macro score (as set by input scorer) across each repeated K-fold.

float The standard deviation of the micro score (as set by input scorer) across each fold with the repeated K-fold.

22.3 Plot_Global_Feat_Importances

BPt_ML.**Plot_Global_Feat_Importances**(*feat_importances='most recent', top_n=10, show_abs=False, multiclass=False, ci=95, palette='default', figsize=(10, 10), title='default', titles='default', xlabel='default', n_cols=1, ax=None, show=True*)

Plots any global feature importance, e.g. base or shap, values per feature not per prediction.

Parameters

- **feat_importances** (*'most recent' or Feat_Importances object*) – Input should be either a Feat_Importances object as output from a call to Evaluate, or Test, or if left as default 'most recent', the passed params will be used to plot any valid calculated feature importances from the last call to Evaluate or Test.

  Note, if there exist multiple valid feature importances in the last call, passing custom ax will most likely break things.

  (default = 'most recent')

- **top_n** (*int, optional*) – The number of top features to display. In the case where show_abs is set to True, or the feature importance being plotted is only positive, then top_n features will be shown. On the other hand, when show_abs is set to False and the feature importances being plotted contain negative numbers, then the top_n highest and top_n lowest features will be shown.

  (default = 10)

- **show_abs** (*bool, optional*) – In the case where the underlying feature importances contain negative numbers, you can either plot the top_n by absolute value, with show_abs set to True, or plot the top_n highest and lowest with show_abs set to False.

  (default = False)

- **multiclass** (*bool, optional*) – If multiclass is set to True, and the underlying feature importances were derived from a categorical problem type, then a seperate feature importance plot will be made for each class. Alternatively, if multiclass is set to False, then feature importances will be averaged over all classes.

  (default = False)

- **ci** (*float, 'sd' or None, optional*) – Size of confidence intervals to draw around estimated values. If 'sd', skip bootstrapping and draw the standard deviation of the feat importances. If None, no bootstrapping will be performed, and error bars will not be drawn.

  (default = 95)

- **palette** (*Seaborn palette name, optional*) – Color scheme to use. Search seaborn palettes for more information. Default for absolute is 'Reds', and default for both pos and neg is 'coolwarm'.

  (default = 'default')

- **title** (*str, optional*) – The title used during plotting, and also used to save a version of the figure (with spaces in title replaced by _, and as a png).

  When multiclass is True, this is the full figure title.

  (default = 'default')

- **titles** (*list, optional*) – This parameter is only used when multiclass is True. titles should be a list with the name for each classes plot. If left as default, it will just be named the original loaded name for that class.

  (default = 'default')

- **xlabel** (*str, optional*) – The xlabel, descriping the measure of feature importance. If left as 'default' it will change depend on what feature importance is being plotted.

  (default = 'default')

- **n_cols** (*int, optional*) – If multiclass, then the number of class plots to plot on each row.

  (default = 1)

- **ax** (*matplotlib axis, or axes, optional*) – A custom ax to plot to for an individual plot, or if using multiclass, then a list of axes can be passed here.

  (default = None)

- **show** (*bool, optional*) – If True, then plt.show(), the matplotlib command will be called, and the figure displayed. On the other hand, if set to False, then the user can customize the plot as they desire. You can think of plt.show() as clearing all of the loaded settings, so in order to make changes, you can't call this until you are done.

  (default = True)

22.4 Plot_Local_Feat_Importances

BPt_ML.**Plot_Local_Feat_Importances**(*feat_importances='most recent', top_n=10, title='default', titles='default', xlabel='default', one_class=None, show=True*)

Plots any local feature importance, e.g. shap, values per per prediction.

Parameters

- **feat_importances** (*'most recent' or Feat_Importances object*) – Input should be either a Feat_Importances object as output from a call to Evaluate, or Test, or if left as default 'most recent', the passed params will be used to plot any valid calculated feature importances from the last call to Evaluate or Test.

(default = 'most recent')

- **top_n** (*int, optional*) – The number of top features to display. In the case where show_abs is set to True, or the feature importance being plotted is only positive, then top_n features will be shown. On the other hand, when show_abs is set to False and the feature importances being plotted contain negative numbers, then the top_n highest and top_n lowest features will be shown.

  (default = 10)

- **title** (*str, optional*) – The title used during plotting, and also used to save a version of the figure (with spaces in title replaced by _, and as a png).

  With a multiclass / categorical problem type, this is only used if one_class is set. Otherwise, titles are used.

  (default = 'default')

- **titles** (*list, optional*) – This parameter is only used with a multiclass problem type. titles should be a list with the name for each class to plot. If left as default, it will use originally loaded class names. for that class.

  (default = 'default')

- **xlabel** (*str, optional*) – The xlabel, descriping the measure of feature importance. If left as 'default' it will change depend on what feature importance is being plotted.

  (default = 'default')

- **one_class** (*int or None, optional*) – If an underlying multiclass or categorical type, optionally provide an int here, corresponding to the single class to plot. If left as None, with make plots for all classes.

  (default = None)

- **show** (*bool, optional*) – If True, then plt.show(), the matplotlib command will be called, and the figure displayed. On the other hand, if set to False, then the user can customize the plot as they desire. You can think of plt.show() as clearing all of the loaded settings, so in order to make changes, you can't call this until you are done.

  (default = True)

CHAPTER

TWENTYTHREE

TESTING PHASE

23.1 Test

BPt_ML.**Test**(*model_pipeline*, *problem_spec='default'*, *train_subjects='train'*, *test_subjects='test'*, *feat_importances=None*, *return_raw_preds=False*, *return_models=False*, *run_name='default'*, *base_dtype='float32'*)

The test function is one of the main interfaces for testing a specific *Model_Pipeline*. Test is conceptually different from *Evaluate* in that it is designed to contrust / train a *Model_Pipeline* on one discrete set of *train_subjects* and evaluate it on a further discrete set of *test_subjects*. Otherwise, these functions are very simmilar as they both evaluate a *Model_Pipeline* as defined in the context of a *Problem_Spec*, and return similar output.

Parameters

- **model_pipeline** (*Model_Pipeline*) – The passed *model_pipeline* should be an instance of the BPt params class *Model_Pipeline*. This object defines the underlying model pipeline to be evaluated.

  See *Model_Pipeline* for more information / how to create a the model pipeline.

- **problem_spec** (*Problem_Spec* or 'default', optional) – *problem_spec* accepts an instance of the BPt.BPt_ML params class *Problem_Spec*. This object is essentially a wrapper around commonly used parameters needs to define the context the model pipeline should be evaluated in. It includes parameters like problem_type, scorer, n_jobs, random_state, etc... See *Problem_Spec* explicitly for more information and for how to create an instance of this object.

  If left as 'default', then will just initialize a Problem_Spec with default params.

  ```
  default = 'default'
  ```

- **train_subjects** (*str, array-like or Value_Subset, optional*) – This parameter determines the set of training subjects which are used to train the passed instance of *Model_Pipeline*.

  Note, this parameter and *test_subjects* are distinct, but complementary to the *subjects* parameter within *Problem_Spec*, which is applied after selecting the subset of *train_subjects* specified here.

  By default, this value will be set to the special str indiciator 'train', which specifies that the full set of globally defined training subjects (See: `Define_Train_Test_Split()`), should be used. Other special str indicators include 'all' to select all subjects, and 'test' to select the test set subjects.

If *subjects* is passed a str, and that str is not one of the str indicators listed above, then it will be interpretted as the location of file in which to read subjects from (assuming one subjects per line).

subjects may also be a custom array-like of subjects to use.

Lastly, a special wrapper, Value_Subset, can also be used to specify more specific, specifically value specific, subsets of subjects to use. See *Value_Subset* for how this input wrapper can be used.

If passing custom input here, be warned that you NEVER want to pass an overlap of subjects between *train_subjects* and *test_subjects*

```
default = 'train'
```

- **test_subjects** (*str, array-like or Value_Subset, optional*) – This parameter determines the set of testing subjects which are used to evaluate the passed instance of *Model_Pipeline*, after it has been trained on the passed *train_subjects*.

  Note, this parameter and *train_subjects* are distinct, but complementary to the *subjects* parameter within *Problem_Spec*, which is applied after selecting the subset of *test_subjects* specified here.

  By default, this value will be set to the special str indiciator 'test', which specifies that the full set of globally defined test subjects (See: `Define_Train_Test_Split()`), should be used. Other special str indicators include 'all' to select all subjects, and 'train' to select the train set subjects.

  If *subjects* is passed a str, and that str is not one of the str indicators listed above, then it will be interpretted as the location of file in which to read subjects from (assuming one subjects per line).

  subjects may also be a custom array-like of subjects to use.

  Lastly, a special wrapper, Value_Subset, can also be used to specify more specific, specifically value specific, subsets of subjects to use. See *Value_Subset* for how this input wrapper can be used.

  If passing custom input here, be warned that you NEVER want to pass an overlap of subjects between *train_subjects* and *test_subjects*

  ```
  default = 'test'
  ```

- **feat_importances** (*Feat_Importance* list of, str or None, optional) – If passed None, by default, no feature importances will be saved.

  Alternatively, one may pass the keyword 'base', to indicate that the base feature importances - those automatically calculated by base objects (e.g., beta weights from linear models) be saved. In this case, the object Feat_Importance('base') will be made.

  Otherwise, for more detailed control provide here either a single, or list of *Feat_Importance* param objects in which to specify what importance values, and with what settings should be computed. See the base *Feat_Importance* object for more information on how to specify these objects.

  See *Feat Importances* to learn more about feature importances generally.

  In this case of a passed list, all passed Feat_Importances will attempt to be computed.

  ```
  default = None
  ```

- **return_raw_preds** (*bool, optional*) – If True, return the raw predictions from each fold.

```
default = False
```

- **return_models** (*bool, optional*) – If True, return the trained models from each evaluation.

```
default = False
```

- **run_name** (*str or 'default', optional*) – Each run of test can be optionally associated with a specific *run_name*. This name is used if *save_results* in *Set_Default_ML_Verbosity* is set to True, then will be used as the name output from Test as saved as in the specific log_dr (if any, and as set when Init'ing the *BPt_ML* class object), with .test appended to the name.

  If left as 'default', will come up with a kind of terrible name passed on the underlying model used in the passed *model_pipeline*.

```
default = 'default'
```

- **base_dtype** (*numpy dtype*) – The dataset is cast to a numpy array of float. This parameter can be used to change the default behavior, e.g., if more resolution or less is needed.

```
default = 'float32'
```

Returns results – Dictionary containing: 'scores', the score on the test set by each scorer, 'raw_preds', A pandas dataframe containing the raw predictions for each subject, in the test set, and 'FIs' a list where each element corresponds to a passed feature importance.

Return type dict

23.2 Plot_Global_Feat_Importances

BPt_ML.**Plot_Global_Feat_Importances**(*feat_importances='most recent', top_n=10, show_abs=False, multiclass=False, ci=95, palette='default', figsize=(10, 10), title='default', titles='default', xlabel='default', n_cols=1, ax=None, show=True*)

Plots any global feature importance, e.g. base or shap, values per feature not per prediction.

Parameters

- **feat_importances** (*'most recent' or Feat_Importances object*) – Input should be either a Feat_Importances object as output from a call to Evaluate, or Test, or if left as default 'most recent', the passed params will be used to plot any valid calculated feature importances from the last call to Evaluate or Test.

  Note, if there exist multiple valid feature importances in the last call, passing custom ax will most likely break things.

  (default = 'most recent')

- **top_n** (*int, optional*) – The number of top features to display. In the case where show_abs is set to True, or the feature importance being plotted is only positive, then top_n

features will be shown. On the other hand, when show_abs is set to False and the feature importances being plotted contain negative numbers, then the top_n highest and top_n lowest features will be shown.

(default = 10)

- **show_abs** (*bool, optional*) – In the case where the underlying feature importances contain negative numbers, you can either plot the top_n by absolute value, with show_abs set to True, or plot the top_n highest and lowest with show_abs set to False.

  (default = False)

- **multiclass** (*bool, optional*) – If multiclass is set to True, and the underlying feature importances were derived from a categorical problem type, then a seperate feature importance plot will be made for each class. Alternatively, if multiclass is set to False, then feature importances will be averaged over all classes.

  (default = False)

- **ci** (*float, 'sd' or None, optional*) – Size of confidence intervals to draw around estimated values. If 'sd', skip bootstrapping and draw the standard deviation of the feat importances. If None, no bootstrapping will be performed, and error bars will not be drawn.

  (default = 95)

- **palette** (*Seaborn palette name, optional*) – Color scheme to use. Search seaborn palettes for more information. Default for absolute is 'Reds', and default for both pos and neg is 'coolwarm'.

  (default = 'default')

- **title** (*str, optional*) – The title used during plotting, and also used to save a version of the figure (with spaces in title replaced by _, and as a png).

  When multiclass is True, this is the full figure title.

  (default = 'default')

- **titles** (*list, optional*) – This parameter is only used when multiclass is True. titles should be a list with the name for each classes plot. If left as default, it will just be named the original loaded name for that class.

  (default = 'default')

- **xlabel** (*str, optional*) – The xlabel, descriping the measure of feature importance. If left as 'default' it will change depend on what feature importance is being plotted.

  (default = 'default')

- **n_cols** (*int, optional*) – If multiclass, then the number of class plots to plot on each row.

  (default = 1)

- **ax** (*matplotlib axis, or axes, optional*) – A custom ax to plot to for an individual plot, or if using multiclass, then a list of axes can be passed here.

  (default = None)

- **show** (*bool, optional*) – If True, then plt.show(), the matplotlib command will be called, and the figure displayed. On the other hand, if set to False, then the user can customize the plot as they desire. You can think of plt.show() as clearing all of the loaded settings, so in order to make changes, you can't call this until you are done.

(default = True)

23.3 Plot_Local_Feat_Importances

BPt_ML.**Plot_Local_Feat_Importances**(*feat_importances='most recent', top_n=10, title='default', titles='default', xlabel='default', one_class=None, show=True*)

Plots any local feature importance, e.g. shap, values per per prediction.

Parameters

- **feat_importances** (*'most recent' or Feat_Importances object*) – Input should be either a Feat_Importances object as output from a call to Evaluate, or Test, or if left as default 'most recent', the passed params will be used to plot any valid calculated feature importances from the last call to Evaluate or Test.

  (default = 'most recent')

- **top_n** (*int, optional*) – The number of top features to display. In the case where show_abs is set to True, or the feature importance being plotted is only positive, then top_n features will be shown. On the other hand, when show_abs is set to False and the feature importances being plotted contain negative numbers, then the top_n highest and top_n lowest features will be shown.

  (default = 10)

- **title** (*str, optional*) – The title used during plotting, and also used to save a version of the figure (with spaces in title replaced by _, and as a png).

  With a multiclass / categorical problem type, this is only used if one_class is set. Otherwise, titles are used.

  (default = 'default')

- **titles** (*list, optional*) – This parameter is only used with a multiclass problem type. titles should be a list with the name for each class to plot. If left as default, it will use originally loaded class names. for that class.

  (default = 'default')

- **xlabel** (*str, optional*) – The xlabel, descriping the measure of feature importance. If left as 'default' it will change depend on what feature importance is being plotted.

  (default = 'default')

- **one_class** (*int or None, optional*) – If an underlying multiclass or categorical type, optionally provide an int here, corresponding to the single class to plot. If left as None, with make plots for all classes.

  (default = None)

- **show** (*bool, optional*) – If True, then plt.show(), the matplotlib command will be called, and the figure displayed. On the other hand, if set to False, then the user can customize the plot as they desire. You can think of plt.show() as clearing all of the loaded settings, so in order to make changes, you can't call this until you are done.

  (default = True)

CHAPTER

TWENTYFOUR

EXTRAS

24.1 Save

BPt_ML.**Save**(*loc*, *low_memory=False*)

This class method is used to save an existing BPt_ML object for further use.

Parameters

- **loc** (*str or Path*) – The location in which the pickle of the BPt_ML object should be saved! This is the same loc which should be passed to *Load* in order to re-load the object.
- **low_memory** (*bool, optional*) – If this parameter is set to True, then self.data, self.targets, self.covars, self.strat will be deleted before saving. The assumption for the param to be used is that self.all_data has already been created, and therefore the individual dataframes with data, covars ect... can safely be deleted as the user will not need to work with them directly any more.

```
default = False
```

24.2 Save_Table

BPt_ML.**Save_Table**(*save_loc*, *targets='SHOW_ALL'*, *covars='SHOW_ALL'*, *strat='SHOW_ALL'*, *group_by=None*, *split=True*, *include_all=True*, *subjects=None*, *cat_show_original_name=True*, *shape='long'*, *heading=None*, *center=True*, *rnd_to=2*, *style=None*)

This method is used to save a table with summary statistics in docx format.

Warning: if there is any NaN data kept in any of the selected covars, then those subject's data will not be outputted to the table! Likewise, only overlapped subjects present in any loaded data, covars, strat, targets, ect... will be outputted to the table!.

Note: you must have the optional library python-docx installed to use this function.

Parameters

- **save_loc** (*str*) – The location where the .docx file with the table should be saved. You should include .docx in this save_loc.
- **targets** (*str, int or list, optional*) – The single (str) or multiple targets (list), in which to add to the outputed table. The str input 'SHOW_ALL' is reserved, and set to default, for displaying all loaded targets.

  You can also pass the int index, (or indices).

(default = 'SHOW_ALL')

- **covars** (*str or list, optional*) – The single (str) or multiple covars (list), in which to add to the outputed table. The str input 'SHOW_ALL' is reserved, and set to default, for displaying all loaded covars.

  Warning: If there are any NaN's in selected covars, these subjects will be disregarded from all summary measures.

  (default = 'SHOW_ALL')

- **strat** (*str or list, optional*) – The single (str) or multiple strat (list), in which to add to the outputed table. The str input 'SHOW_ALL' is reserved, and set to default, for displaying all loaded strat.

  Note, if strat is passed, then self.strat_u_name will be removed from every title before plotting, so if a same variables is loaded as a covar and a strat it will be put in the table twice, and if in a rarer case strat_u_name has been changed to a common value present in a covar name, then this common key will be removed.

  (default = 'SHOW_ALL')

- **group_by** (*str or list, optional*) – This parameter, by default None, controls if the table statistics should be further broken down by different binary / categorical (or multilabel) groups. For example, by passing 'split', (assuming the split param has been left as True), will output a table with statistics for each column as seperated by global train test split.

  Notably, 'split' is a special keyword, to split on any other group, the name of that feature/column should be passed. E.g., to split on a loaded covar 'sex', then 'sex' would be passed.

  If a list of values is passed, then each element will be used for its own seperate split. Further, if the *include_all* parameter is left as its default value of True, then in addition to a single or multiple group_by splits, the values over all subjects will also be displayed. (E.g. in the train test split case, statistics would be shown for the full sample, only the train subjects and only the test subjects).

  (default = None)

- **split** (*bool, optional*) – If True, then information about the global train test split will be added as a binary feature of the table. If False, or if a global split has not yet been defined, this split will not be added to the table. Note that 'split' can also be passed to *group_by*.

  Note: If it is desired to create a table with just the train subjects (or test) for example, then the *subjects* parameter should be used. If subjects is set to 'train' or 'test', this *split* param will switch to False regardless of user passed input.

  (default = True)

- **include_all** (*bool, optional*) – If True, as default, then in addition to passed group_by param(s), statistics will be displayed over all subjects as well. If no group_by param(s) are passed, then *include_all* will be True regardless of user passed value.

  (default = True)

- **subjects** (*None, 'train', 'test' or array-like, optional*) – If left as None, display a table with all overlapping subjects. Alternatively this parameter can be used to pass just the train_subjects with 'train' or just the test subjects with 'test' or even a custom list or array-like set of subjects

(default = None)

- **cat_show_original_name** (*bool, optional*) – If True, then when showing a categorical distribution (or binary) use the original loaded name in the table. Otherwise, if False, the internally used variable names will be used to determine table headers.

  (default = True)

- **shape** (*{'long', 'wide'}, optional*) – There are two options for shape. First 'long', which is selected by default, will create a table where each row is a summary statistic for a passed variable, and each column represents each *group_by* group if any. This is a good choice when the number of variable to plot is more than the number of groups to display values by.

  Alternatively, you may pass 'wide', for the opposite behavior of 'long', where in this case the variables to summarize will each be a column in the table, and the group_by groups will represent rows.

  If your table ends up being too squished in either direction, you can try the opposite shape. If both are squished, you'll need to reduce the number of group_by variables or targets, covars/ strat.

  (default = 'long')

- **heading** (*str, optional*) – You may optionally pass a heading for the table as a str. By default no heading will be added.

  (default = None)

- **center** (*bool, optional*) – This parameter optionally determines if the values in the table along with the headers should be centered within each cell. If False, the values will be left alligned on the bottom.

  (default = True)

- **rnd_to** (*int, optional*) – This parameter determines how many decimal places each value to be added to the table should be rounded to. E.g., the default value of 2 will round each table entry to 2 decimal points, but if rnd_to 0 was passed, that would be no decimal points and -1, would be rounded to the nearest 10.

  (default = 2)

- **style** (*str or None, optional*) – The default .docx table style in which to use, which contrals things like table color and fonts, ect... Keep style as None by default to just use the default style, or feel free to try passing any of a huge number of preset styles. These styles can be found at the bottom of https://python-docx.readthedocs.io/en/latest/user/styles-understanding.html under "Table styles in default template".

  Some examples are: 'Colorful Grid', 'Dark List', 'Light List', 'Medium List 2', 'Medium Shading 2 Accent 6', ect...

  (default = None)

CHAPTER

TWENTYFIVE

MODELS

Different base obj choices for the *Model* are shown below The exact str indicator, as passed to the *obj* param is represented by the sub-heading (within “”) The avaliable models are further broken down by which can workwith different problem_types. Additionally, a link to the original models documentation as well as the implemented parameter distributions are shown.

25.1 binary

25.1.1 “dt classifier”

Base Class Documenation: `sklearn.tree.DecisionTreeClassifier`

Param Distributions

0. “default”

```
defaults only
```

1. “dt classifier dist”

```
max_depth: ng.p.Scalar(lower=1, upper=30).set_integer_casting()
min_samples_split: ng.p.Scalar(lower=2, upper=50).set_integer_casting()
class_weight: ng.p.TransitionChoice([None, 'balanced'])
```

25.1.2 “elastic net logistic”

Base Class Documenation: `sklearn.linear_model.LogisticRegression`

Param Distributions

0. “base elastic”

```
max_iter: 1000
multi_class: 'auto'
penalty: 'elasticnet'
class_weight: None
solver: 'saga'
l1_ratio: .5
```

1. “elastic classifier”

```
max_iter: 1000
multi_class: 'auto'
penalty: 'elasticnet'
class_weight: ng.p.TransitionChoice([None, 'balanced'])
solver: 'saga'
l1_ratio: ng.p.Scalar(lower=.01, upper=1)
C: ng.p.Log(lower=1e-5, upper=1e5)
```

2. “elastic clf v2”

```
max_iter: 1000
multi_class: 'auto'
penalty: 'elasticnet'
class_weight: ng.p.TransitionChoice([None, 'balanced'])
solver: 'saga'
l1_ratio: ng.p.Scalar(lower=.01, upper=1)
C: ng.p.Log(lower=1e-2, upper=1e5)
```

3. “elastic classifier extra”

```
max_iter: ng.p.Scalar(lower=1000, upper=10000).set_integer_casting()
multi_class: 'auto'
penalty: 'elasticnet'
class_weight: ng.p.TransitionChoice([None, 'balanced'])
solver: 'saga'
l1_ratio: ng.p.Scalar(lower=.01, upper=1)
C: ng.p.Log(lower=1e-5, upper=1e5)
tol: ng.p.Log(lower=1e-6, upper=.01)
```

25.1.3 “et classifier”

Base Class Documenation: `sklearn.ensemble.ExtraTreesClassifier`

Param Distributions

0. “default”

```
defaults only
```

25.1.4 “gaussian nb”

Base Class Documenation: `sklearn.naive_bayes.GaussianNB`

Param Distributions

0. “base gnb”

```
var_smoothing: 1e-9
```

25.1.5 “gb classifier”

Base Class Documenation: `sklearn.ensemble.GradientBoostingClassifier`

Param Distributions

0. “default”

```
defaults only
```

25.1.6 “gp classifier”

Base Class Documenation: sklearn.gaussian_process.GaussianProcessClassifier

Param Distributions

0. “base gp classifier”

```
n_restarts_optimizer: 5
```

25.1.7 “hgb classifier”

Base Class Documenation: sklearn.ensemble.gradient_boosting.HistGradientBoostingClassifier

Param Distributions

0. “default”

```
defaults only
```

25.1.8 “knn classifier”

Base Class Documenation: sklearn.neighbors.KNeighborsClassifier

Param Distributions

0. “base knn”

```
n_neighbors: 5
```

1. “knn dist”

```
weights: ng.p.TransitionChoice(['uniform', 'distance'])
n_neighbors: ng.p.Scalar(lower=2, upper=25).set_integer_casting()
```

25.1.9 “lasso logistic”

Base Class Documenation: sklearn.linear_model.LogisticRegression

Param Distributions

0. “base lasso”

```
max_iter: 1000
multi_class: 'auto'
penalty: 'l1'
class_weight: None
solver: 'liblinear'
```

1. “lasso C”

```
max_iter: 1000
multi_class: 'auto'
penalty: 'l1'
class_weight: ng.p.TransitionChoice([None, 'balanced'])
solver: 'liblinear'
C: ng.p.Log(lower=1e-5, upper=1e3)
```

2. "lasso C extra"

```
max_iter: ng.p.Scalar(lower=1000, upper=10000).set_integer_casting()
multi_class: 'auto'
penalty: 'l1'
class_weight: ng.p.TransitionChoice([None, 'balanced'])
solver: 'liblinear'
C: ng.p.Log(lower=1e-5, upper=1e3)
tol: ng.p.Log(lower=1e-6, upper=.01)
```

25.1.10 "light gbm classifier"

Base Class Documenation: `lightgbm.LGBMClassifier`

Param Distributions

0. "base lgbm"

```
silent: True
```

1. "lgbm classifier dist1"

```
silent: True
boosting_type: ng.p.TransitionChoice(['gbdt', 'dart', 'goss'])
n_estimators: ng.p.Scalar(init=100, lower=3, upper=500).set_integer_
↪casting()
num_leaves: ng.p.Scalar(init=20, lower=6, upper=80).set_integer_casting()
min_child_samples: ng.p.Scalar(lower=10, upper=500).set_integer_casting()
min_child_weight: ng.p.Log(lower=1e-5, upper=1e4)
subsample: ng.p.Scalar(lower=.3, upper=.95)
colsample_bytree: ng.p.Scalar(lower=.3, upper=.95)
reg_alpha: ng.p.TransitionChoice([0, ng.p.Log(lower=1e-5, upper=1)])
reg_lambda: ng.p.TransitionChoice([0, ng.p.Log(lower=1e-5, upper=1)])
class_weight: ng.p.TransitionChoice([None, 'balanced'])
```

2. "lgbm classifier dist2"

```
silent: True
lambda_l2: 0.001
boosting_type: ng.p.TransitionChoice(['gbdt', 'dart'])
min_child_samples: ng.p.TransitionChoice([1, 5, 7, 10, 15, 20, 35, 50,␣
↪100, 200, 500, 1000])
num_leaves: ng.p.TransitionChoice([2, 4, 7, 10, 15, 20, 25, 30, 35, 40,␣
↪50, 65, 80, 100, 125, 150, 200, 250])
colsample_bytree: ng.p.TransitionChoice([0.7, 0.9, 1.0])
subsample: ng.p.Scalar(lower=.3, upper=1)
learning_rate: ng.p.TransitionChoice([0.01, 0.05, 0.1])
n_estimators: ng.p.TransitionChoice([5, 20, 35, 50, 75, 100, 150, 200,␣
↪350, 500, 750, 1000])
class_weight: ng.p.TransitionChoice([None, 'balanced'])
```

25.1.11 "linear svm classifier"

Base Class Documenation: `sklearn.svm.LinearSVC`

Param Distributions

0. "base linear svc"

```
max_iter: 1000
```

1. "linear svc dist"

```
max_iter: 1000
C: ng.p.Log(lower=1e-4, upper=1e4)
class_weight: ng.p.TransitionChoice([None, 'balanced'])
```

25.1.12 "logistic"

Base Class Documenation: `sklearn.linear_model.LogisticRegression`

Param Distributions

0. "base logistic"

```
max_iter: 1000
multi_class: 'auto'
penalty: 'none'
class_weight: None
solver: 'lbfgs'
```

25.1.13 "mlp classifier"

Base Class Documenation: `BPt.extensions.MLP.MLPClassifier_Wrapper`

Param Distributions

0. "default"

```
defaults only
```

1. "mlp dist 3 layer"

```
hidden_layer_sizes: ng.p.Array(init=(100, 100, 100)).set_
↪mutation(sigma=50).set_bounds(lower=1, upper=300).set_integer_casting()
activation: ng.p.TransitionChoice(['identity', 'logistic', 'tanh', 'relu
↪'])
alpha: ng.p.Log(lower=1e-5, upper=1e2)
batch_size: ng.p.TransitionChoice(['auto', ng.p.Scalar(init=200,␣
↪lower=50, upper=400).set_integer_casting()])
learning_rate: ng.p.TransitionChoice(['constant', 'invscaling', 'adaptive
↪'])
learning_rate_init: ng.p.Log(lower=1e-5, upper=1e2)
max_iter: ng.p.Scalar(init=200, lower=100, upper=1000).set_integer_
↪casting()
beta_1: ng.p.Scalar(init=.9, lower=.1, upper=.99)
beta_2: ng.p.Scalar(init=.999, lower=.1, upper=.9999)
```

2. "mlp dist es 3 layer"

```
hidden_layer_sizes: ng.p.Scalar(init=100, lower=2, upper=300).set_
↪integer_casting()
activation: ng.p.TransitionChoice(['identity', 'logistic', 'tanh', 'relu
↪'])
alpha: ng.p.Log(lower=1e-5, upper=1e2)
batch_size: ng.p.TransitionChoice(['auto', ng.p.Scalar(init=200,␣
↪lower=50, upper=400).set_integer_casting()])
learning_rate: ng.p.TransitionChoice(['constant', 'invscaling', 'adaptive
↪'])
learning_rate_init: ng.p.Log(lower=1e-5, upper=1e2)
max_iter: ng.p.Scalar(init=200, lower=100, upper=1000).set_integer_
↪casting()
beta_1: ng.p.Scalar(init=.9, lower=.1, upper=.99)
beta_2: ng.p.Scalar(init=.999, lower=.1, upper=.9999)
early_stopping: True
n_iter_no_change: ng.p.Scalar(lower=5, upper=50)
```

3. "mlp dist 2 layer"

```
hidden_layer_sizes: ng.p.Array(init=(100, 100)).set_mutation(sigma=50).
↪set_bounds(lower=1, upper=300).set_integer_casting()
activation: ng.p.TransitionChoice(['identity', 'logistic', 'tanh', 'relu
↪'])
alpha: ng.p.Log(lower=1e-5, upper=1e2)
batch_size: ng.p.TransitionChoice(['auto', ng.p.Scalar(init=200,␣
↪lower=50, upper=400).set_integer_casting()])
learning_rate: ng.p.TransitionChoice(['constant', 'invscaling', 'adaptive
↪'])
learning_rate_init: ng.p.Log(lower=1e-5, upper=1e2)
max_iter: ng.p.Scalar(init=200, lower=100, upper=1000).set_integer_
↪casting()
beta_1: ng.p.Scalar(init=.9, lower=.1, upper=.99)
beta_2: ng.p.Scalar(init=.999, lower=.1, upper=.9999)
```

4. "mlp dist es 2 layer"

```
hidden_layer_sizes: ng.p.Scalar(init=100, lower=2, upper=300).set_
↪integer_casting()
activation: ng.p.TransitionChoice(['identity', 'logistic', 'tanh', 'relu
↪'])
alpha: ng.p.Log(lower=1e-5, upper=1e2)
batch_size: ng.p.TransitionChoice(['auto', ng.p.Scalar(init=200,␣
↪lower=50, upper=400).set_integer_casting()])
learning_rate: ng.p.TransitionChoice(['constant', 'invscaling', 'adaptive
↪'])
learning_rate_init: ng.p.Log(lower=1e-5, upper=1e2)
max_iter: ng.p.Scalar(init=200, lower=100, upper=1000).set_integer_
↪casting()
beta_1: ng.p.Scalar(init=.9, lower=.1, upper=.99)
beta_2: ng.p.Scalar(init=.999, lower=.1, upper=.9999)
early_stopping: True
n_iter_no_change: ng.p.Scalar(lower=5, upper=50)
```

5. "mlp dist 1 layer"

```
hidden_layer_sizes: ng.p.Scalar(init=100, lower=2, upper=300).set_
↪integer_casting()
activation: ng.p.TransitionChoice(['identity', 'logistic', 'tanh', 'relu
↪'])
alpha: ng.p.Log(lower=1e-5, upper=1e2)
batch_size: ng.p.TransitionChoice(['auto', ng.p.Scalar(init=200,␣
↪lower=50, upper=400).set_integer_casting()])
learning_rate: ng.p.TransitionChoice(['constant', 'invscaling', 'adaptive
↪'])
learning_rate_init: ng.p.Log(lower=1e-5, upper=1e2)
max_iter: ng.p.Scalar(init=200, lower=100, upper=1000).set_integer_
↪casting()
beta_1: ng.p.Scalar(init=.9, lower=.1, upper=.99)
beta_2: ng.p.Scalar(init=.999, lower=.1, upper=.9999)
```

6. “mlp dist es 1 layer”

```
hidden_layer_sizes: ng.p.Scalar(init=100, lower=2, upper=300).set_
↪integer_casting()
activation: ng.p.TransitionChoice(['identity', 'logistic', 'tanh', 'relu
↪'])
alpha: ng.p.Log(lower=1e-5, upper=1e2)
batch_size: ng.p.TransitionChoice(['auto', ng.p.Scalar(init=200,␣
↪lower=50, upper=400).set_integer_casting()])
learning_rate: ng.p.TransitionChoice(['constant', 'invscaling', 'adaptive
↪'])
learning_rate_init: ng.p.Log(lower=1e-5, upper=1e2)
max_iter: ng.p.Scalar(init=200, lower=100, upper=1000).set_integer_
↪casting()
beta_1: ng.p.Scalar(init=.9, lower=.1, upper=.99)
beta_2: ng.p.Scalar(init=.999, lower=.1, upper=.9999)
early_stopping: True
n_iter_no_change: ng.p.Scalar(lower=5, upper=50)
```

25.1.14 “pa classifier”

Base Class Documenation: `sklearn.linear_model.PassiveAggressiveClassifier`

Param Distributions

0. “default”

```
defaults only
```

25.1.15 “random forest classifier”

Base Class Documenation: `sklearn.ensemble.RandomForestClassifier`

Param Distributions

0. “base rf regressor”

```
n_estimators: 100
```

1. “rf classifier dist”

```
n_estimators: ng.p.Scalar(init=100, lower=3, upper=500).set_integer_
↪casting()
max_depth: ng.p.TransitionChoice([None, ng.p.Scalar(init=25, lower=2,␣
↪upper=200).set_integer_casting()])
max_features: ng.p.Scalar(lower=.1, upper=1.0)
min_samples_split: ng.p.Scalar(lower=.1, upper=1.0)
bootstrap: True
class_weight: ng.p.TransitionChoice([None, 'balanced'])
```

25.1.16 "ridge logistic"

Base Class Documenation: `sklearn.linear_model.LogisticRegression`

Param Distributions

0. "base ridge"

```
max_iter: 1000
penalty: 'l2'
solver: 'saga'
```

1. "ridge C"

```
max_iter: 1000
solver: 'saga'
C: ng.p.Log(lower=1e-5, upper=1e3)
class_weight: ng.p.TransitionChoice([None, 'balanced'])
```

2. "ridge C extra"

```
max_iter: ng.p.Scalar(lower=1000, upper=10000).set_integer_casting()
solver: 'saga'
C: ng.p.Log(lower=1e-5, upper=1e3)
class_weight: ng.p.TransitionChoice([None, 'balanced'])
tol: ng.p.Log(lower=1e-6, upper=.01)
```

25.1.17 "sgd classifier"

Base Class Documenation: `sklearn.linear_model.SGDClassifier`

Param Distributions

0. "base sgd"

```
loss: 'hinge'
```

1. "sgd classifier"

```
loss: ng.p.TransitionChoice(['hinge', 'log', 'modified_huber', 'squared_
↪hinge', 'perceptron'])
penalty: ng.p.TransitionChoice(['l2', 'l1', 'elasticnet'])
alpha: ng.p.Log(lower=1e-5, upper=1e2)
l1_ratio: ng.p.Scalar(lower=0, upper=1)
max_iter: 1000
learning_rate: ng.p.TransitionChoice(['optimal', 'invscaling', 'adaptive
↪', 'constant'])
```

(continues on next page)

(continued from previous page)

```
eta0: ng.p.Log(lower=1e-6, upper=1e3)
power_t: ng.p.Scalar(lower=.1, upper=.9)
early_stopping: ng.p.TransitionChoice([False, True])
validation_fraction: ng.p.Scalar(lower=.05, upper=.5)
n_iter_no_change: ng.p.TransitionChoice(np.arange(2, 20))
class_weight: ng.p.TransitionChoice([None, 'balanced'])
```

25.1.18 "svm classifier"

Base Class Documenation: `sklearn.svm.SVC`

Param Distributions

0. "base svm classifier"

```
kernel: 'rbf'
gamma: 'scale'
probability: True
```

1. "svm classifier dist"

```
kernel: 'rbf'
gamma: ng.p.Log(lower=1e-6, upper=1)
C: ng.p.Log(lower=1e-4, upper=1e4)
probability: True
class_weight: ng.p.TransitionChoice([None, 'balanced'])
```

25.1.19 "xgb classifier"

Base Class Documenation: `xgboost.XGBClassifier`

Param Distributions

0. "base xgb classifier"

```
verbosity: 0
objective: 'binary:logistic'
```

1. "xgb classifier dist1"

```
verbosity: 0
objective: 'binary:logistic'
n_estimators: ng.p.Scalar(init=100, lower=3, upper=500).set_integer_
↪casting()
min_child_weight: ng.p.Log(lower=1e-5, upper=1e4)
subsample: ng.p.Scalar(lower=.3, upper=.95)
colsample_bytree: ng.p.Scalar(lower=.3, upper=.95)
reg_alpha: ng.p.TransitionChoice([0, ng.p.Log(lower=1e-5, upper=1)])
reg_lambda: ng.p.TransitionChoice([0, ng.p.Log(lower=1e-5, upper=1)])
```

2. "xgb classifier dist2"

```
verbosity: 0
objective: 'binary:logistic'
```

(continues on next page)

(continued from previous page)

```
max_depth: ng.p.TransitionChoice([None, ng.p.Scalar(init=25, lower=2,␣
↪upper=200).set_integer_casting()])
learning_rate: ng.p.Scalar(lower=.01, upper=.5)
n_estimators: ng.p.Scalar(lower=3, upper=500).set_integer_casting()
min_child_weight: ng.p.TransitionChoice([1, 5, 10, 50])
subsample: ng.p.Scalar(lower=.5, upper=1)
colsample_bytree: ng.p.Scalar(lower=.4, upper=.95)
```

3. “xgb classifier dist3”

```
verbosity: 0
objective: 'binary:logistic'
learning_rare: ng.p.Scalar(lower=.005, upper=.3)
min_child_weight: ng.p.Scalar(lower=.5, upper=10)
max_depth: ng.p.TransitionChoice(np.arange(3, 10))
subsample: ng.p.Scalar(lower=.5, upper=1)
colsample_bytree: ng.p.Scalar(lower=.5, upper=1)
reg_alpha: ng.p.Log(lower=.00001, upper=1)
```

25.2 regression

25.2.1 “ard regressor”

Base Class Documenation: `sklearn.linear_model.ARDRegression`

Param Distributions

0. “default”

```
defaults only
```

25.2.2 “bayesian ridge regressor”

Base Class Documenation: `sklearn.linear_model.BayesianRidge`

Param Distributions

0. “default”

```
defaults only
```

25.2.3 “dt regressor”

Base Class Documenation: `sklearn.tree.DecisionTreeRegressor`

Param Distributions

0. “default”

```
defaults only
```

1. “dt dist”

```
max_depth: ng.p.Scalar(lower=1, upper=30).set_integer_casting()
min_samples_split: ng.p.Scalar(lower=2, upper=50).set_integer_casting()
```

25.2.4 "elastic net regressor"

Base Class Documenation: sklearn.linear_model.ElasticNet

Param Distributions

0. "base elastic net"

```
max_iter: 1000
```

1. "elastic regression"

```
max_iter: 1000
alpha: ng.p.Log(lower=1e-5, upper=1e5)
l1_ratio: ng.p.Scalar(lower=.01, upper=1)
```

2. "elastic regression extra"

```
max_iter: ng.p.Scalar(lower=1000, upper=10000).set_integer_casting()
alpha: ng.p.Log(lower=1e-5, upper=1e5)
l1_ratio: ng.p.Scalar(lower=.01, upper=1)
tol: ng.p.Log(lower=1e-6, upper=.01)
```

25.2.5 "et regressor"

Base Class Documenation: sklearn.ensemble.ExtraTreesRegressor

Param Distributions

0. "default"

```
defaults only
```

25.2.6 "gb regressor"

Base Class Documenation: sklearn.ensemble.GradientBoostingRegressor

Param Distributions

0. "default"

```
defaults only
```

25.2.7 "gp regressor"

Base Class Documenation: sklearn.gaussian_process.GaussianProcessRegressor

Param Distributions

0. "base gp regressor"

```
n_restarts_optimizer: 5
normalize_y: True
```

25.2.8 "hgb regressor"

Base Class Documenation: `sklearn.ensemble.gradient_boosting.HistGradientBoostingRegressor`

Param Distributions

0. "default"

```
defaults only
```

25.2.9 "knn regressor"

Base Class Documenation: `sklearn.neighbors.KNeighborsRegressor`

Param Distributions

0. "base knn regression"

```
n_neighbors: 5
```

1. "knn dist regression"

```
weights: ng.p.TransitionChoice(['uniform', 'distance'])
n_neighbors: ng.p.Scalar(lower=2, upper=25).set_integer_casting()
```

25.2.10 "lasso regressor"

Base Class Documenation: `sklearn.linear_model.Lasso`

Param Distributions

0. "base lasso regressor"

```
max_iter: 1000
```

1. "lasso regressor dist"

```
max_iter: 1000
alpha: ng.p.Log(lower=1e-5, upper=1e5)
```

25.2.11 "light gbm regressor"

Base Class Documenation: `lightgbm.LGBMRegressor`

Param Distributions

0. "base lgbm"

```
silent: True
```

1. "lgbm dist1"

```
silent: True
boosting_type: ng.p.TransitionChoice(['gbdt', 'dart', 'goss'])
n_estimators: ng.p.Scalar(init=100, lower=3, upper=500).set_integer_
↪casting()
num_leaves: ng.p.Scalar(init=20, lower=6, upper=80).set_integer_casting()
min_child_samples: ng.p.Scalar(lower=10, upper=500).set_integer_casting()
min_child_weight: ng.p.Log(lower=1e-5, upper=1e4)
subsample: ng.p.Scalar(lower=.3, upper=.95)
colsample_bytree: ng.p.Scalar(lower=.3, upper=.95)
reg_alpha: ng.p.TransitionChoice([0, ng.p.Log(lower=1e-5, upper=1)])
reg_lambda: ng.p.TransitionChoice([0, ng.p.Log(lower=1e-5, upper=1)])
```

2. "lgbm dist2"

```
silent: True
lambda_l2: 0.001
boosting_type: ng.p.TransitionChoice(['gbdt', 'dart'])
min_child_samples: ng.p.TransitionChoice([1, 5, 7, 10, 15, 20, 35, 50,␣
↪100, 200, 500, 1000])
num_leaves: ng.p.TransitionChoice([2, 4, 7, 10, 15, 20, 25, 30, 35, 40,␣
↪50, 65, 80, 100, 125, 150, 200, 250])
colsample_bytree: ng.p.TransitionChoice([0.7, 0.9, 1.0])
subsample: ng.p.Scalar(lower=.3, upper=1)
learning_rate: ng.p.TransitionChoice([0.01, 0.05, 0.1])
n_estimators: ng.p.TransitionChoice([5, 20, 35, 50, 75, 100, 150, 200,␣
↪350, 500, 750, 1000])
```

25.2.12 "linear regressor"

Base Class Documenation: `sklearn.linear_model.LinearRegression`

Param Distributions

0. "base linear"

```
fit_intercept: True
```

25.2.13 "linear svm regressor"

Base Class Documenation: `sklearn.svm.LinearSVR`

Param Distributions

0. "base linear svr"

```
loss: 'epsilon_insensitive'
max_iter: 1000
```

1. "linear svr dist"

```
loss: 'epsilon_insensitive'
max_iter: 1000
C: ng.p.Log(lower=1e-4, upper=1e4)
```

25.2.14 "mlp regressor"

Base Class Documenation: `BPt.extensions.MLP.MLPRegressor_Wrapper`

Param Distributions

0. "default"

```
defaults only
```

1. "mlp dist 3 layer"

```
hidden_layer_sizes: ng.p.Array(init=(100, 100, 100)).set_
↪mutation(sigma=50).set_bounds(lower=1, upper=300).set_integer_casting()
activation: ng.p.TransitionChoice(['identity', 'logistic', 'tanh', 'relu
↪'])
alpha: ng.p.Log(lower=1e-5, upper=1e2)
batch_size: ng.p.TransitionChoice(['auto', ng.p.Scalar(init=200,␣
↪lower=50, upper=400).set_integer_casting()])
learning_rate: ng.p.TransitionChoice(['constant', 'invscaling', 'adaptive
↪'])
learning_rate_init: ng.p.Log(lower=1e-5, upper=1e2)
max_iter: ng.p.Scalar(init=200, lower=100, upper=1000).set_integer_
↪casting()
beta_1: ng.p.Scalar(init=.9, lower=.1, upper=.99)
beta_2: ng.p.Scalar(init=.999, lower=.1, upper=.9999)
```

2. "mlp dist es 3 layer"

```
hidden_layer_sizes: ng.p.Scalar(init=100, lower=2, upper=300).set_
↪integer_casting()
activation: ng.p.TransitionChoice(['identity', 'logistic', 'tanh', 'relu
↪'])
alpha: ng.p.Log(lower=1e-5, upper=1e2)
batch_size: ng.p.TransitionChoice(['auto', ng.p.Scalar(init=200,␣
↪lower=50, upper=400).set_integer_casting()])
learning_rate: ng.p.TransitionChoice(['constant', 'invscaling', 'adaptive
↪'])
learning_rate_init: ng.p.Log(lower=1e-5, upper=1e2)
max_iter: ng.p.Scalar(init=200, lower=100, upper=1000).set_integer_
↪casting()
beta_1: ng.p.Scalar(init=.9, lower=.1, upper=.99)
beta_2: ng.p.Scalar(init=.999, lower=.1, upper=.9999)
early_stopping: True
n_iter_no_change: ng.p.Scalar(lower=5, upper=50)
```

3. "mlp dist 2 layer"

```
hidden_layer_sizes: ng.p.Array(init=(100, 100)).set_mutation(sigma=50).
↪set_bounds(lower=1, upper=300).set_integer_casting()
activation: ng.p.TransitionChoice(['identity', 'logistic', 'tanh', 'relu
↪'])
alpha: ng.p.Log(lower=1e-5, upper=1e2)
batch_size: ng.p.TransitionChoice(['auto', ng.p.Scalar(init=200,␣
↪lower=50, upper=400).set_integer_casting()])
learning_rate: ng.p.TransitionChoice(['constant', 'invscaling', 'adaptive
↪'])
learning_rate_init: ng.p.Log(lower=1e-5, upper=1e2)
```

(continues on next page)

(continued from previous page)

```
max_iter: ng.p.Scalar(init=200, lower=100, upper=1000).set_integer_
↪casting()
beta_1: ng.p.Scalar(init=.9, lower=.1, upper=.99)
beta_2: ng.p.Scalar(init=.999, lower=.1, upper=.9999)
```

4. “mlp dist es 2 layer”

```
hidden_layer_sizes: ng.p.Scalar(init=100, lower=2, upper=300).set_
↪integer_casting()
activation: ng.p.TransitionChoice(['identity', 'logistic', 'tanh', 'relu
↪'])
alpha: ng.p.Log(lower=1e-5, upper=1e2)
batch_size: ng.p.TransitionChoice(['auto', ng.p.Scalar(init=200,␣
↪lower=50, upper=400).set_integer_casting()])
learning_rate: ng.p.TransitionChoice(['constant', 'invscaling', 'adaptive
↪'])
learning_rate_init: ng.p.Log(lower=1e-5, upper=1e2)
max_iter: ng.p.Scalar(init=200, lower=100, upper=1000).set_integer_
↪casting()
beta_1: ng.p.Scalar(init=.9, lower=.1, upper=.99)
beta_2: ng.p.Scalar(init=.999, lower=.1, upper=.9999)
early_stopping: True
n_iter_no_change: ng.p.Scalar(lower=5, upper=50)
```

5. “mlp dist 1 layer”

```
hidden_layer_sizes: ng.p.Scalar(init=100, lower=2, upper=300).set_
↪integer_casting()
activation: ng.p.TransitionChoice(['identity', 'logistic', 'tanh', 'relu
↪'])
alpha: ng.p.Log(lower=1e-5, upper=1e2)
batch_size: ng.p.TransitionChoice(['auto', ng.p.Scalar(init=200,␣
↪lower=50, upper=400).set_integer_casting()])
learning_rate: ng.p.TransitionChoice(['constant', 'invscaling', 'adaptive
↪'])
learning_rate_init: ng.p.Log(lower=1e-5, upper=1e2)
max_iter: ng.p.Scalar(init=200, lower=100, upper=1000).set_integer_
↪casting()
beta_1: ng.p.Scalar(init=.9, lower=.1, upper=.99)
beta_2: ng.p.Scalar(init=.999, lower=.1, upper=.9999)
```

6. “mlp dist es 1 layer”

```
hidden_layer_sizes: ng.p.Scalar(init=100, lower=2, upper=300).set_
↪integer_casting()
activation: ng.p.TransitionChoice(['identity', 'logistic', 'tanh', 'relu
↪'])
alpha: ng.p.Log(lower=1e-5, upper=1e2)
batch_size: ng.p.TransitionChoice(['auto', ng.p.Scalar(init=200,␣
↪lower=50, upper=400).set_integer_casting()])
learning_rate: ng.p.TransitionChoice(['constant', 'invscaling', 'adaptive
↪'])
learning_rate_init: ng.p.Log(lower=1e-5, upper=1e2)
max_iter: ng.p.Scalar(init=200, lower=100, upper=1000).set_integer_
↪casting()
beta_1: ng.p.Scalar(init=.9, lower=.1, upper=.99)
beta_2: ng.p.Scalar(init=.999, lower=.1, upper=.9999)
```

(continues on next page)

(continued from previous page)

```
early_stopping: True
n_iter_no_change: ng.p.Scalar(lower=5, upper=50)
```

25.2.15 "random forest regressor"

Base Class Documenation: sklearn.ensemble.RandomForestRegressor

Param Distributions

0. "base rf"

```
n_estimators: 100
```

1. "rf dist"

```
n_estimators: ng.p.Scalar(init=100, lower=3, upper=500).set_integer_
↪casting()
max_depth: ng.p.TransitionChoice([None, ng.p.Scalar(init=25, lower=2,␣
↪upper=200).set_integer_casting()])
max_features: ng.p.Scalar(lower=.1, upper=1.0)
min_samples_split: ng.p.Scalar(lower=.1, upper=1.0)
bootstrap: True
```

25.2.16 "ridge regressor"

Base Class Documenation: sklearn.linear_model.Ridge

Param Distributions

0. "base ridge regressor"

```
max_iter: 1000
solver: 'lsqr'
```

1. "ridge regressor dist"

```
max_iter: 1000
solver: 'lsqr'
alpha: ng.p.Log(lower=1e-3, upper=1e5)
```

25.2.17 "svm regressor"

Base Class Documenation: sklearn.svm.SVR

Param Distributions

0. "base svm"

```
kernel: 'rbf'
gamma: 'scale'
```

1. "svm dist"

```
kernel: 'rbf'
gamma: ng.p.Log(lower=1e-6, upper=1)
C: ng.p.Log(lower=1e-4, upper=1e4)
```

25.2.18 "tweedie regressor"

Base Class Documenation: `sklearn.linear_model.glm.TweedieRegressor`

Param Distributions

0. "default"

```
defaults only
```

25.2.19 "xgb regressor"

Base Class Documenation: `xgboost.XGBRegressor`

Param Distributions

0. "base xgb"

```
verbosity: 0
objective: 'reg:squarederror'
```

1. "xgb dist1"

```
verbosity: 0
objective: 'reg:squarederror'
n_estimators: ng.p.Scalar(init=100, lower=3, upper=500).set_integer_
↪casting()
min_child_weight: ng.p.Log(lower=1e-5, upper=1e4)
subsample: ng.p.Scalar(lower=.3, upper=.95)
colsample_bytree: ng.p.Scalar(lower=.3, upper=.95)
reg_alpha: ng.p.TransitionChoice([0, ng.p.Log(lower=1e-5, upper=1)])
reg_lambda: ng.p.TransitionChoice([0, ng.p.Log(lower=1e-5, upper=1)])
```

2. "xgb dist2"

```
verbosity: 0
objective: 'reg:squarederror'
max_depth: ng.p.TransitionChoice([None, ng.p.Scalar(init=25, lower=2,␣
↪upper=200).set_integer_casting()])
learning_rate: ng.p.Scalar(lower=.01, upper=.5)
n_estimators: ng.p.Scalar(lower=3, upper=500).set_integer_casting()
min_child_weight: ng.p.TransitionChoice([1, 5, 10, 50])
subsample: ng.p.Scalar(lower=.5, upper=1)
colsample_bytree: ng.p.Scalar(lower=.4, upper=.95)
```

3. "xgb dist3"

```
verbosity: 0
objective: 'reg:squarederror'
learning_rare: ng.p.Scalar(lower=.005, upper=.3)
min_child_weight: ng.p.Scalar(lower=.5, upper=10)
```

(continues on next page)

(continued from previous page)

```
max_depth: ng.p.TransitionChoice(np.arange(3, 10))
subsample: ng.p.Scalar(lower=.5, upper=1)
colsample_bytree: ng.p.Scalar(lower=.5, upper=1)
reg_alpha: ng.p.Log(lower=.00001, upper=1)
```

25.3 categorical

25.3.1 "dt classifier"

Base Class Documenation: `sklearn.tree.DecisionTreeClassifier`

Param Distributions

0. "default"

```
defaults only
```

1. "dt classifier dist"

```
max_depth: ng.p.Scalar(lower=1, upper=30).set_integer_casting()
min_samples_split: ng.p.Scalar(lower=2, upper=50).set_integer_casting()
class_weight: ng.p.TransitionChoice([None, 'balanced'])
```

25.3.2 "elastic net logistic"

Base Class Documenation: `sklearn.linear_model.LogisticRegression`

Param Distributions

0. "base elastic"

```
max_iter: 1000
multi_class: 'auto'
penalty: 'elasticnet'
class_weight: None
solver: 'saga'
l1_ratio: .5
```

1. "elastic classifier"

```
max_iter: 1000
multi_class: 'auto'
penalty: 'elasticnet'
class_weight: ng.p.TransitionChoice([None, 'balanced'])
solver: 'saga'
l1_ratio: ng.p.Scalar(lower=.01, upper=1)
C: ng.p.Log(lower=1e-5, upper=1e5)
```

2. "elastic clf v2"

```
max_iter: 1000
multi_class: 'auto'
penalty: 'elasticnet'
```

(continues on next page)

(continued from previous page)

```
class_weight: ng.p.TransitionChoice([None, 'balanced'])
solver: 'saga'
l1_ratio: ng.p.Scalar(lower=.01, upper=1)
C: ng.p.Log(lower=1e-2, upper=1e5)
```

3. "elastic classifier extra"

```
max_iter: ng.p.Scalar(lower=1000, upper=10000).set_integer_casting()
multi_class: 'auto'
penalty: 'elasticnet'
class_weight: ng.p.TransitionChoice([None, 'balanced'])
solver: 'saga'
l1_ratio: ng.p.Scalar(lower=.01, upper=1)
C: ng.p.Log(lower=1e-5, upper=1e5)
tol: ng.p.Log(lower=1e-6, upper=.01)
```

25.3.3 "et classifier"

Base Class Documenation: `sklearn.ensemble.ExtraTreesClassifier`

Param Distributions

0. "default"

```
defaults only
```

25.3.4 "gaussian nb"

Base Class Documenation: `sklearn.naive_bayes.GaussianNB`

Param Distributions

0. "base gnb"

```
var_smoothing: 1e-9
```

25.3.5 "gb classifier"

Base Class Documenation: `sklearn.ensemble.GradientBoostingClassifier`

Param Distributions

0. "default"

```
defaults only
```

25.3.6 "gp classifier"

Base Class Documenation: `sklearn.gaussian_process.GaussianProcessClassifier`

Param Distributions

0. "base gp classifier"

```
n_restarts_optimizer: 5
```

25.3.7 "hgb classifier"

Base Class Documenation: `sklearn.ensemble.gradient_boosting.HistGradientBoostingClassifier`

Param Distributions

0. "default"

```
defaults only
```

25.3.8 "knn classifier"

Base Class Documenation: `sklearn.neighbors.KNeighborsClassifier`

Param Distributions

0. "base knn"

```
n_neighbors: 5
```

1. "knn dist"

```
weights: ng.p.TransitionChoice(['uniform', 'distance'])
n_neighbors: ng.p.Scalar(lower=2, upper=25).set_integer_casting()
```

25.3.9 "lasso logistic"

Base Class Documenation: `sklearn.linear_model.LogisticRegression`

Param Distributions

0. "base lasso"

```
max_iter: 1000
multi_class: 'auto'
penalty: 'l1'
class_weight: None
solver: 'liblinear'
```

1. "lasso C"

```
max_iter: 1000
multi_class: 'auto'
penalty: 'l1'
class_weight: ng.p.TransitionChoice([None, 'balanced'])
solver: 'liblinear'
C: ng.p.Log(lower=1e-5, upper=1e3)
```

2. "lasso C extra"

```
max_iter: ng.p.Scalar(lower=1000, upper=10000).set_integer_casting()
multi_class: 'auto'
penalty: 'l1'
class_weight: ng.p.TransitionChoice([None, 'balanced'])
solver: 'liblinear'
C: ng.p.Log(lower=1e-5, upper=1e3)
tol: ng.p.Log(lower=1e-6, upper=.01)
```

25.3.10 "light gbm classifier"

Base Class Documenation: `lightgbm.LGBMClassifier`

Param Distributions

0. "base lgbm"

```
silent: True
```

1. "lgbm classifier dist1"

```
silent: True
boosting_type: ng.p.TransitionChoice(['gbdt', 'dart', 'goss'])
n_estimators: ng.p.Scalar(init=100, lower=3, upper=500).set_integer_
↪casting()
num_leaves: ng.p.Scalar(init=20, lower=6, upper=80).set_integer_casting()
min_child_samples: ng.p.Scalar(lower=10, upper=500).set_integer_casting()
min_child_weight: ng.p.Log(lower=1e-5, upper=1e4)
subsample: ng.p.Scalar(lower=.3, upper=.95)
colsample_bytree: ng.p.Scalar(lower=.3, upper=.95)
reg_alpha: ng.p.TransitionChoice([0, ng.p.Log(lower=1e-5, upper=1)])
reg_lambda: ng.p.TransitionChoice([0, ng.p.Log(lower=1e-5, upper=1)])
class_weight: ng.p.TransitionChoice([None, 'balanced'])
```

2. "lgbm classifier dist2"

```
silent: True
lambda_l2: 0.001
boosting_type: ng.p.TransitionChoice(['gbdt', 'dart'])
min_child_samples: ng.p.TransitionChoice([1, 5, 7, 10, 15, 20, 35, 50,␣
↪100, 200, 500, 1000])
num_leaves: ng.p.TransitionChoice([2, 4, 7, 10, 15, 20, 25, 30, 35, 40,␣
↪50, 65, 80, 100, 125, 150, 200, 250])
colsample_bytree: ng.p.TransitionChoice([0.7, 0.9, 1.0])
subsample: ng.p.Scalar(lower=.3, upper=1)
learning_rate: ng.p.TransitionChoice([0.01, 0.05, 0.1])
n_estimators: ng.p.TransitionChoice([5, 20, 35, 50, 75, 100, 150, 200,␣
↪350, 500, 750, 1000])
class_weight: ng.p.TransitionChoice([None, 'balanced'])
```

25.3.11 "linear svm classifier"

Base Class Documenation: `sklearn.svm.LinearSVC`

Param Distributions

0. "base linear svc"

```
max_iter: 1000
```

1. “linear svc dist”

```
max_iter: 1000
C: ng.p.Log(lower=1e-4, upper=1e4)
class_weight: ng.p.TransitionChoice([None, 'balanced'])
```

25.3.12 “logistic”

Base Class Documenation: `sklearn.linear_model.LogisticRegression`

Param Distributions

0. “base logistic”

```
max_iter: 1000
multi_class: 'auto'
penalty: 'none'
class_weight: None
solver: 'lbfgs'
```

25.3.13 “mlp classifier”

Base Class Documenation: `BPt.extensions.MLP.MLPClassifier_Wrapper`

Param Distributions

0. “default”

```
defaults only
```

1. “mlp dist 3 layer”

```
hidden_layer_sizes: ng.p.Array(init=(100, 100, 100)).set_
↪mutation(sigma=50).set_bounds(lower=1, upper=300).set_integer_casting()
activation: ng.p.TransitionChoice(['identity', 'logistic', 'tanh', 'relu
↪'])
alpha: ng.p.Log(lower=1e-5, upper=1e2)
batch_size: ng.p.TransitionChoice(['auto', ng.p.Scalar(init=200,␣
↪lower=50, upper=400).set_integer_casting()])
learning_rate: ng.p.TransitionChoice(['constant', 'invscaling', 'adaptive
↪'])
learning_rate_init: ng.p.Log(lower=1e-5, upper=1e2)
max_iter: ng.p.Scalar(init=200, lower=100, upper=1000).set_integer_
↪casting()
beta_1: ng.p.Scalar(init=.9, lower=.1, upper=.99)
beta_2: ng.p.Scalar(init=.999, lower=.1, upper=.9999)
```

2. “mlp dist es 3 layer”

```
hidden_layer_sizes: ng.p.Scalar(init=100, lower=2, upper=300).set_
↪integer_casting()
activation: ng.p.TransitionChoice(['identity', 'logistic', 'tanh', 'relu
↪'])
```

(continues on next page)

(continued from previous page)

```
alpha: ng.p.Log(lower=1e-5, upper=1e2)
batch_size: ng.p.TransitionChoice(['auto', ng.p.Scalar(init=200,␣
↪lower=50, upper=400).set_integer_casting()])
learning_rate: ng.p.TransitionChoice(['constant', 'invscaling', 'adaptive
↪'])
learning_rate_init: ng.p.Log(lower=1e-5, upper=1e2)
max_iter: ng.p.Scalar(init=200, lower=100, upper=1000).set_integer_
↪casting()
beta_1: ng.p.Scalar(init=.9, lower=.1, upper=.99)
beta_2: ng.p.Scalar(init=.999, lower=.1, upper=.9999)
early_stopping: True
n_iter_no_change: ng.p.Scalar(lower=5, upper=50)
```

3. “mlp dist 2 layer”

```
hidden_layer_sizes: ng.p.Array(init=(100, 100)).set_mutation(sigma=50).
↪set_bounds(lower=1, upper=300).set_integer_casting()
activation: ng.p.TransitionChoice(['identity', 'logistic', 'tanh', 'relu
↪'])
alpha: ng.p.Log(lower=1e-5, upper=1e2)
batch_size: ng.p.TransitionChoice(['auto', ng.p.Scalar(init=200,␣
↪lower=50, upper=400).set_integer_casting()])
learning_rate: ng.p.TransitionChoice(['constant', 'invscaling', 'adaptive
↪'])
learning_rate_init: ng.p.Log(lower=1e-5, upper=1e2)
max_iter: ng.p.Scalar(init=200, lower=100, upper=1000).set_integer_
↪casting()
beta_1: ng.p.Scalar(init=.9, lower=.1, upper=.99)
beta_2: ng.p.Scalar(init=.999, lower=.1, upper=.9999)
```

4. “mlp dist es 2 layer”

```
hidden_layer_sizes: ng.p.Scalar(init=100, lower=2, upper=300).set_
↪integer_casting()
activation: ng.p.TransitionChoice(['identity', 'logistic', 'tanh', 'relu
↪'])
alpha: ng.p.Log(lower=1e-5, upper=1e2)
batch_size: ng.p.TransitionChoice(['auto', ng.p.Scalar(init=200,␣
↪lower=50, upper=400).set_integer_casting()])
learning_rate: ng.p.TransitionChoice(['constant', 'invscaling', 'adaptive
↪'])
learning_rate_init: ng.p.Log(lower=1e-5, upper=1e2)
max_iter: ng.p.Scalar(init=200, lower=100, upper=1000).set_integer_
↪casting()
beta_1: ng.p.Scalar(init=.9, lower=.1, upper=.99)
beta_2: ng.p.Scalar(init=.999, lower=.1, upper=.9999)
early_stopping: True
n_iter_no_change: ng.p.Scalar(lower=5, upper=50)
```

5. “mlp dist 1 layer”

```
hidden_layer_sizes: ng.p.Scalar(init=100, lower=2, upper=300).set_
↪integer_casting()
activation: ng.p.TransitionChoice(['identity', 'logistic', 'tanh', 'relu
↪'])
alpha: ng.p.Log(lower=1e-5, upper=1e2)
batch_size: ng.p.TransitionChoice(['auto', ng.p.Scalar(init=200,␣
↪lower=50, upper=400).set_integer_casting()])
```

(continues on next page)

(continued from previous page)

```
learning_rate: ng.p.TransitionChoice(['constant', 'invscaling', 'adaptive
↪'])
learning_rate_init: ng.p.Log(lower=1e-5, upper=1e2)
max_iter: ng.p.Scalar(init=200, lower=100, upper=1000).set_integer_
↪casting()
beta_1: ng.p.Scalar(init=.9, lower=.1, upper=.99)
beta_2: ng.p.Scalar(init=.999, lower=.1, upper=.9999)
```

6. “mlp dist es 1 layer”

```
hidden_layer_sizes: ng.p.Scalar(init=100, lower=2, upper=300).set_
↪integer_casting()
activation: ng.p.TransitionChoice(['identity', 'logistic', 'tanh', 'relu
↪'])
alpha: ng.p.Log(lower=1e-5, upper=1e2)
batch_size: ng.p.TransitionChoice(['auto', ng.p.Scalar(init=200,␣
↪lower=50, upper=400).set_integer_casting()])
learning_rate: ng.p.TransitionChoice(['constant', 'invscaling', 'adaptive
↪'])
learning_rate_init: ng.p.Log(lower=1e-5, upper=1e2)
max_iter: ng.p.Scalar(init=200, lower=100, upper=1000).set_integer_
↪casting()
beta_1: ng.p.Scalar(init=.9, lower=.1, upper=.99)
beta_2: ng.p.Scalar(init=.999, lower=.1, upper=.9999)
early_stopping: True
n_iter_no_change: ng.p.Scalar(lower=5, upper=50)
```

25.3.14 “pa classifier”

Base Class Documenation: `sklearn.linear_model.PassiveAggressiveClassifier`

Param Distributions

0. “default”

```
defaults only
```

25.3.15 “random forest classifier”

Base Class Documenation: `sklearn.ensemble.RandomForestClassifier`

Param Distributions

0. “base rf regressor”

```
n_estimators: 100
```

1. “rf classifier dist”

```
n_estimators: ng.p.Scalar(init=100, lower=3, upper=500).set_integer_
↪casting()
max_depth: ng.p.TransitionChoice([None, ng.p.Scalar(init=25, lower=2,␣
↪upper=200).set_integer_casting()])
max_features: ng.p.Scalar(lower=.1, upper=1.0)
```

(continues on next page)

(continued from previous page)

```
min_samples_split: ng.p.Scalar(lower=.1, upper=1.0)
bootstrap: True
class_weight: ng.p.TransitionChoice([None, 'balanced'])
```

25.3.16 "ridge logistic"

Base Class Documenation: sklearn.linear_model.LogisticRegression

Param Distributions

0. "base ridge"

```
max_iter: 1000
penalty: 'l2'
solver: 'saga'
```

1. "ridge C"

```
max_iter: 1000
solver: 'saga'
C: ng.p.Log(lower=1e-5, upper=1e3)
class_weight: ng.p.TransitionChoice([None, 'balanced'])
```

2. "ridge C extra"

```
max_iter: ng.p.Scalar(lower=1000, upper=10000).set_integer_casting()
solver: 'saga'
C: ng.p.Log(lower=1e-5, upper=1e3)
class_weight: ng.p.TransitionChoice([None, 'balanced'])
tol: ng.p.Log(lower=1e-6, upper=.01)
```

25.3.17 "sgd classifier"

Base Class Documenation: sklearn.linear_model.SGDClassifier

Param Distributions

0. "base sgd"

```
loss: 'hinge'
```

1. "sgd classifier"

```
loss: ng.p.TransitionChoice(['hinge', 'log', 'modified_huber', 'squared_
↪hinge', 'perceptron'])
penalty: ng.p.TransitionChoice(['l2', 'l1', 'elasticnet'])
alpha: ng.p.Log(lower=1e-5, upper=1e2)
l1_ratio: ng.p.Scalar(lower=0, upper=1)
max_iter: 1000
learning_rate: ng.p.TransitionChoice(['optimal', 'invscaling', 'adaptive
↪', 'constant'])
eta0: ng.p.Log(lower=1e-6, upper=1e3)
power_t: ng.p.Scalar(lower=.1, upper=.9)
early_stopping: ng.p.TransitionChoice([False, True])
```

(continues on next page)

(continued from previous page)

```
validation_fraction: ng.p.Scalar(lower=.05, upper=.5)
n_iter_no_change: ng.p.TransitionChoice(np.arange(2, 20))
class_weight: ng.p.TransitionChoice([None, 'balanced'])
```

25.3.18 "svm classifier"

Base Class Documenation: `sklearn.svm.SVC`

Param Distributions

0. "base svm classifier"

```
kernel: 'rbf'
gamma: 'scale'
probability: True
```

1. "svm classifier dist"

```
kernel: 'rbf'
gamma: ng.p.Log(lower=1e-6, upper=1)
C: ng.p.Log(lower=1e-4, upper=1e4)
probability: True
class_weight: ng.p.TransitionChoice([None, 'balanced'])
```

25.3.19 "xgb classifier"

Base Class Documenation: `xgboost.XGBClassifier`

Param Distributions

0. "base xgb classifier"

```
verbosity: 0
objective: 'binary:logistic'
```

1. "xgb classifier dist1"

```
verbosity: 0
objective: 'binary:logistic'
n_estimators: ng.p.Scalar(init=100, lower=3, upper=500).set_integer_
↪casting()
min_child_weight: ng.p.Log(lower=1e-5, upper=1e4)
subsample: ng.p.Scalar(lower=.3, upper=.95)
colsample_bytree: ng.p.Scalar(lower=.3, upper=.95)
reg_alpha: ng.p.TransitionChoice([0, ng.p.Log(lower=1e-5, upper=1)])
reg_lambda: ng.p.TransitionChoice([0, ng.p.Log(lower=1e-5, upper=1)])
```

2. "xgb classifier dist2"

```
verbosity: 0
objective: 'binary:logistic'
max_depth: ng.p.TransitionChoice([None, ng.p.Scalar(init=25, lower=2,␣
↪upper=200).set_integer_casting()])
learning_rate: ng.p.Scalar(lower=.01, upper=.5)
```

(continues on next page)

(continued from previous page)

```
n_estimators: ng.p.Scalar(lower=3, upper=500).set_integer_casting()
min_child_weight: ng.p.TransitionChoice([1, 5, 10, 50])
subsample: ng.p.Scalar(lower=.5, upper=1)
colsample_bytree: ng.p.Scalar(lower=.4, upper=.95)
```

3. "xgb classifier dist3"

```
verbosity: 0
objective: 'binary:logistic'
learning_rare: ng.p.Scalar(lower=.005, upper=.3)
min_child_weight: ng.p.Scalar(lower=.5, upper=10)
max_depth: ng.p.TransitionChoice(np.arange(3, 10))
subsample: ng.p.Scalar(lower=.5, upper=1)
colsample_bytree: ng.p.Scalar(lower=.5, upper=1)
reg_alpha: ng.p.Log(lower=.00001, upper=1)
```

CHAPTER

TWENTYSIX

SCORERS

Different availible choices for the *scorer* parameter are shown below. *scorer* is accepted by *Problem_Spec*, *Param_Search* and *Feat_Importance* The str indicator for each *scorer* is represented bythe sub-heading (within “”) The avaliable scorers are further broken down by which can work with different problem_types. Additionally, a link to the original models documentation is shown.

26.1 binary

26.1.1 “accuracy”

Base Func Documenation: sklearn.metrics.accuracy_score()

26.1.2 “roc_auc”

Base Func Documenation: sklearn.metrics.roc_auc_score()

26.1.3 “roc_auc_ovr”

Base Func Documenation: sklearn.metrics.roc_auc_score()

26.1.4 “roc_auc_ovo”

Base Func Documenation: sklearn.metrics.roc_auc_score()

26.1.5 “roc_auc_ovr_weighted”

Base Func Documenation: sklearn.metrics.roc_auc_score()

26.1.6 “roc_auc_ovo_weighted”

Base Func Documenation: sklearn.metrics.roc_auc_score()

26.1.7 “balanced_accuracy”

Base Func Documenation: sklearn.metrics.balanced_accuracy_score()

26.1.8 "average_precision"

Base Func Documenation: sklearn.metrics.average_precision_score()

26.1.9 "neg_log_loss"

Base Func Documenation: sklearn.metrics.log_loss()

26.1.10 "neg_brier_score"

Base Func Documenation: sklearn.metrics.brier_score_loss()

26.1.11 "precision"

Base Func Documenation: sklearn.metrics.precision_score()

26.1.12 "precision_macro"

Base Func Documenation: sklearn.metrics.precision_score()

26.1.13 "precision_micro"

Base Func Documenation: sklearn.metrics.precision_score()

26.1.14 "precision_samples"

Base Func Documenation: sklearn.metrics.precision_score()

26.1.15 "precision_weighted"

Base Func Documenation: sklearn.metrics.precision_score()

26.1.16 "recall"

Base Func Documenation: sklearn.metrics.recall_score()

26.1.17 "recall_macro"

Base Func Documenation: sklearn.metrics.recall_score()

26.1.18 "recall_micro"

Base Func Documenation: sklearn.metrics.recall_score()

26.1.19 "recall_samples"

Base Func Documenation: `sklearn.metrics.recall_score()`

26.1.20 "recall_weighted"

Base Func Documenation: `sklearn.metrics.recall_score()`

26.1.21 "f1"

Base Func Documenation: `sklearn.metrics.f1_score()`

26.1.22 "f1_macro"

Base Func Documenation: `sklearn.metrics.f1_score()`

26.1.23 "f1_micro"

Base Func Documenation: `sklearn.metrics.f1_score()`

26.1.24 "f1_samples"

Base Func Documenation: `sklearn.metrics.f1_score()`

26.1.25 "f1_weighted"

Base Func Documenation: `sklearn.metrics.f1_score()`

26.1.26 "jaccard"

Base Func Documenation: `sklearn.metrics.jaccard_score()`

26.1.27 "jaccard_macro"

Base Func Documenation: `sklearn.metrics.jaccard_score()`

26.1.28 "jaccard_micro"

Base Func Documenation: `sklearn.metrics.jaccard_score()`

26.1.29 "jaccard_samples"

Base Func Documenation: `sklearn.metrics.jaccard_score()`

26.1.30 "jaccard_weighted"

Base Func Documenation: sklearn.metrics.jaccard_score()

26.1.31 "neg_hamming"

Base Func Documenation: sklearn.metrics.hamming_loss()

26.1.32 "matthews"

Base Func Documenation: sklearn.metrics.matthews_corrcoef()

26.1.33 "default"

Base Func Documenation: sklearn.metrics.roc_auc_score()

26.2 regression

26.2.1 "explained_variance"

Base Func Documenation: sklearn.metrics.explained_variance_score()

26.2.2 "explained_variance score"

Base Func Documenation: sklearn.metrics.explained_variance_score()

26.2.3 "r2"

Base Func Documenation: sklearn.metrics.r2_score()

26.2.4 "max_error"

Base Func Documenation: sklearn.metrics.max_error()

26.2.5 "neg_median_absolute_error"

Base Func Documenation: sklearn.metrics.median_absolute_error()

26.2.6 "median_absolute_error"

Base Func Documenation: sklearn.metrics.median_absolute_error()

26.2.7 "neg_mean_absolute_error"

Base Func Documenation: sklearn.metrics.mean_absolute_error()

26.2.8 "mean_absolute_error"

Base Func Documenation: sklearn.metrics.mean_absolute_error()

26.2.9 "neg_mean_squared_error"

Base Func Documenation: sklearn.metrics.mean_squared_error()

26.2.10 "mean_squared_error"

Base Func Documenation: sklearn.metrics.mean_squared_error()

26.2.11 "neg_mean_squared_log_error"

Base Func Documenation: sklearn.metrics.mean_squared_log_error()

26.2.12 "mean_squared_log_error"

Base Func Documenation: sklearn.metrics.mean_squared_log_error()

26.2.13 "neg_root_mean_squared_error"

Base Func Documenation: sklearn.metrics.mean_squared_error()

26.2.14 "root_mean_squared_error"

Base Func Documenation: sklearn.metrics.mean_squared_error()

26.2.15 "neg_mean_poisson_deviance"

Base Func Documenation: sklearn.metrics.mean_poisson_deviance()

26.2.16 "mean_poisson_deviance"

Base Func Documenation: sklearn.metrics.mean_poisson_deviance()

26.2.17 "neg_mean_gamma_deviance"

Base Func Documenation: sklearn.metrics.mean_gamma_deviance()

26.2.18 "mean_gamma_deviance"

Base Func Documenation: sklearn.metrics.mean_gamma_deviance()

26.2.19 "default"

Base Func Documenation: sklearn.metrics.r2_score()

26.3 categorical

26.3.1 "accuracy"

Base Func Documenation: sklearn.metrics.accuracy_score()

26.3.2 "roc_auc"

Base Func Documenation: sklearn.metrics.roc_auc_score()

26.3.3 "roc_auc_ovr"

Base Func Documenation: sklearn.metrics.roc_auc_score()

26.3.4 "roc_auc_ovo"

Base Func Documenation: sklearn.metrics.roc_auc_score()

26.3.5 "roc_auc_ovr_weighted"

Base Func Documenation: sklearn.metrics.roc_auc_score()

26.3.6 "roc_auc_ovo_weighted"

Base Func Documenation: sklearn.metrics.roc_auc_score()

26.3.7 "balanced_accuracy"

Base Func Documenation: sklearn.metrics.balanced_accuracy_score()

26.3.8 "average_precision"

Base Func Documenation: sklearn.metrics.average_precision_score()

26.3.9 "neg_log_loss"

Base Func Documenation: `sklearn.metrics.log_loss()`

26.3.10 "neg_brier_score"

Base Func Documenation: `sklearn.metrics.brier_score_loss()`

26.3.11 "precision"

Base Func Documenation: `sklearn.metrics.precision_score()`

26.3.12 "precision_macro"

Base Func Documenation: `sklearn.metrics.precision_score()`

26.3.13 "precision_micro"

Base Func Documenation: `sklearn.metrics.precision_score()`

26.3.14 "precision_samples"

Base Func Documenation: `sklearn.metrics.precision_score()`

26.3.15 "precision_weighted"

Base Func Documenation: `sklearn.metrics.precision_score()`

26.3.16 "recall"

Base Func Documenation: `sklearn.metrics.recall_score()`

26.3.17 "recall_macro"

Base Func Documenation: `sklearn.metrics.recall_score()`

26.3.18 "recall_micro"

Base Func Documenation: `sklearn.metrics.recall_score()`

26.3.19 "recall_samples"

Base Func Documenation: `sklearn.metrics.recall_score()`

26.3.20 "recall_weighted"

Base Func Documenation: sklearn.metrics.recall_score()

26.3.21 "f1"

Base Func Documenation: sklearn.metrics.f1_score()

26.3.22 "f1_macro"

Base Func Documenation: sklearn.metrics.f1_score()

26.3.23 "f1_micro"

Base Func Documenation: sklearn.metrics.f1_score()

26.3.24 "f1_samples"

Base Func Documenation: sklearn.metrics.f1_score()

26.3.25 "f1_weighted"

Base Func Documenation: sklearn.metrics.f1_score()

26.3.26 "jaccard"

Base Func Documenation: sklearn.metrics.jaccard_score()

26.3.27 "jaccard_macro"

Base Func Documenation: sklearn.metrics.jaccard_score()

26.3.28 "jaccard_micro"

Base Func Documenation: sklearn.metrics.jaccard_score()

26.3.29 "jaccard_samples"

Base Func Documenation: sklearn.metrics.jaccard_score()

26.3.30 "jaccard_weighted"

Base Func Documenation: sklearn.metrics.jaccard_score()

26.3.31 "neg_hamming"

Base Func Documenation: `sklearn.metrics.hamming_loss()`

26.3.32 "matthews"

Base Func Documenation: `sklearn.metrics.matthews_corrcoef()`

26.3.33 "default"

Base Func Documenation: `sklearn.metrics.roc_auc_score()`

CHAPTER

TWENTYSEVEN

LOADERS

Different base obj choices for the *Loader* are shown below The exact str indicator, as passed to the *obj* param is represented by the sub-heading (within “”) Additionally, a link to the original models documentation as well as the implemented parameter distributions are shown.

27.1 All Problem Types

27.1.1 “identity”

Base Class Documenation: `BPt.extensions.Loaders.Identity`

Param Distributions

0. “default”

```
defaults only
```

27.1.2 “surface rois”

Base Class Documenation: *BPt.extensions.Loaders.SurfLabels*

Param Distributions

0. “default”

```
defaults only
```

27.1.3 “volume rois”

Base Class Documenation: `nilearn.input_data.nifti_labels_masker.NiftiLabelsMasker`

Param Distributions

0. “default”

```
defaults only
```

27.1.4 “connectivity”

Base Class Documenation: `BPt.extensions.Loaders.Connectivity`

Param Distributions

0. “default”

```
defaults only
```

CHAPTER

TWENTYEIGHT

IMPUTERS

Different base obj choices for the *Imputer* are shown below The exact str indicator, as passed to the *obj* param is represented by the sub-heading (within “”) Additionally, a link to the original models documentation as well as the implemented parameter distributions are shown. Note that if the iterative imputer is requested, base_model must also be passed.

28.1 All Problem Types

28.1.1 “mean”

Base Class Documenation: `sklearn.impute.SimpleImputer`

Param Distributions

0. “mean imp”

```
strategy: 'mean'
```

28.1.2 “median”

Base Class Documenation: `sklearn.impute.SimpleImputer`

Param Distributions

0. “median imp”

```
strategy: 'median'
```

28.1.3 “most frequent”

Base Class Documenation: `sklearn.impute.SimpleImputer`

Param Distributions

0. “most freq imp”

```
strategy: 'most_frequent'
```

28.1.4 “constant”

Base Class Documenation: `sklearn.impute.SimpleImputer`

Param Distributions

0. “constant imp”

```
strategy: 'constant'
```

28.1.5 “iterative”

Base Class Documenation: `sklearn.impute.IterativeImputer`

Param Distributions

0. “iterative imp”

```
initial_strategy: 'mean'
skip_complete: True
```

CHAPTER

TWENTYNINE

SCALERS

Different base obj choices for the *Scaler* are shown below The exact str indicator, as passed to the *obj* param is represented by the sub-heading (within “”) Additionally, a link to the original models documentation as well as the implemented parameter distributions are shown.

29.1 All Problem Types

29.1.1 “standard”

Base Class Documenation: sklearn.preprocessing.StandardScaler

Param Distributions

0. “base standard”

```
with_mean: True
with_std: True
```

29.1.2 “minmax”

Base Class Documenation: sklearn.preprocessing.MinMaxScaler

Param Distributions

0. “base minmax”

```
feature_range: (0, 1)
```

29.1.3 “maxabs”

Base Class Documenation: sklearn.preprocessing.MaxAbsScaler

Param Distributions

0. “default”

```
defaults only
```

29.1.4 “robust”

Base Class Documenation: sklearn.preprocessing.RobustScaler

Param Distributions

0. “base robust”

```
quantile_range: (5, 95)
```

1. “robust gs”

```
quantile_range: ng.p.TransitionChoice([(x, 100-x) for x in np.arange(1,␣
↪40)])
```

29.1.5 “yeo”

Base Class Documenation: sklearn.preprocessing.PowerTransformer

Param Distributions

0. “base yeo”

```
method: 'yeo-johnson'
standardize: True
```

29.1.6 “boxcox”

Base Class Documenation: sklearn.preprocessing.PowerTransformer

Param Distributions

0. “base boxcox”

```
method: 'box-cox'
standardize: True
```

29.1.7 “winsorize”

Base Class Documenation: *BPt.extensions.Scalers.Winsorizer*

Param Distributions

0. “base winsorize”

```
quantile_range: (1, 99)
```

1. “winsorize gs”

```
quantile_range: ng.p.TransitionChoice([(x, 100-x) for x in np.arange(1,␣
↪40)])
```

29.1.8 “quantile norm”

Base Class Documenation: sklearn.preprocessing.QuantileTransformer

Param Distributions

0. “base quant norm”

```
output_distribution: 'normal'
```

29.1.9 “quantile uniform”

Base Class Documenation: sklearn.preprocessing.QuantileTransformer

Param Distributions

0. “base quant uniform”

```
output_distribution: 'uniform'
```

29.1.10 “normalize”

Base Class Documenation: sklearn.preprocessing.Normalizer

Param Distributions

0. “default”

```
defaults only
```

CHAPTER

THIRTY

TRANSFORMERS

Different base obj choices for the *Transformer* are shown below The exact str indicator, as passed to the *obj* param is represented by the sub-heading (within “”) Additionally, a link to the original models documentation as well as the implemented parameter distributions are shown.

30.1 All Problem Types

30.1.1 “pca”

Base Class Documenation: sklearn.decomposition.PCA

Param Distributions

0. “default”

```
defaults only
```

1. “pca var search”

```
n_components: ng.p.Scalar(init=.75, lower=.1, upper=.99)
svd_solver: 'full'
```

30.1.2 “sparse pca”

Base Class Documenation: sklearn.decomposition.SparsePCA

Param Distributions

0. “default”

```
defaults only
```

30.1.3 “mini batch sparse pca”

Base Class Documenation: sklearn.decomposition.MiniBatchSparsePCA

Param Distributions

0. “default”

```
defaults only
```

30.1.4 "factor analysis"

Base Class Documenation: sklearn.decomposition.FactorAnalysis

Param Distributions

0. "default"

```
defaults only
```

30.1.5 "dictionary learning"

Base Class Documenation: sklearn.decomposition.DictionaryLearning

Param Distributions

0. "default"

```
defaults only
```

30.1.6 "mini batch dictionary learning"

Base Class Documenation: sklearn.decomposition.MiniBatchDictionaryLearning

Param Distributions

0. "default"

```
defaults only
```

30.1.7 "fast ica"

Base Class Documenation: sklearn.decomposition.FastICA

Param Distributions

0. "default"

```
defaults only
```

30.1.8 "incremental pca"

Base Class Documenation: sklearn.decomposition.IncrementalPCA

Param Distributions

0. "default"

```
defaults only
```

30.1.9 "kernel pca"

Base Class Documenation: sklearn.decomposition.KernelPCA

Param Distributions

0. "default"

```
defaults only
```

30.1.10 "nmf"

Base Class Documenation: sklearn.decomposition.NMF

Param Distributions

0. "default"

```
defaults only
```

30.1.11 "truncated svd"

Base Class Documenation: sklearn.decomposition.TruncatedSVD

Param Distributions

0. "default"

```
defaults only
```

30.1.12 "one hot encoder"

Base Class Documenation: sklearn.preprocessing.OneHotEncoder

Param Distributions

0. "ohe"

```
sparse: False
handle_unknown: 'ignore'
```

30.1.13 "backward difference encoder"

Base Class Documenation: category_encoders.backward_difference.BackwardDifferenceEncoder

Param Distributions

0. "default"

```
defaults only
```

30.1.14 "binary encoder"

Base Class Documenation: category_encoders.binary.BinaryEncoder

Param Distributions

0. "default"

```
defaults only
```

30.1.15 "cat boost encoder"

Base Class Documenation: category_encoders.cat_boost.CatBoostEncoder

Param Distributions

0. "default"

```
defaults only
```

30.1.16 "helmert encoder"

Base Class Documenation: category_encoders.helmert.HelmertEncoder

Param Distributions

0. "default"

```
defaults only
```

30.1.17 "james stein encoder"

Base Class Documenation: category_encoders.james_stein.JamesSteinEncoder

Param Distributions

0. "default"

```
defaults only
```

30.1.18 "leave one out encoder"

Base Class Documenation: category_encoders.leave_one_out.LeaveOneOutEncoder

Param Distributions

0. "default"

```
defaults only
```

30.1.19 "m estimate encoder"

Base Class Documenation: `category_encoders.m_estimate.MEstimateEncoder`

Param Distributions

0. "default"

```
defaults only
```

30.1.20 "polynomial encoder"

Base Class Documenation: `category_encoders.polynomial.PolynomialEncoder`

Param Distributions

0. "default"

```
defaults only
```

30.1.21 "sum encoder"

Base Class Documenation: `category_encoders.sum_coding.SumEncoder`

Param Distributions

0. "default"

```
defaults only
```

30.1.22 "target encoder"

Base Class Documenation: `category_encoders.target_encoder.TargetEncoder`

Param Distributions

0. "default"

```
defaults only
```

30.1.23 "woe encoder"

Base Class Documenation: `category_encoders.woe.WOEEncoder`

Param Distributions

0. "default"

```
defaults only
```

CHAPTER

THIRTYONE

FEAT SELECTORS

Different base obj choices for the *Feat_Selector* are shown below The exact str indicator, as passed to the *obj* param is represented by the sub-heading (within “”) The avaliable feat selectors are further broken down by which can workwith different problem_types. Additionally, a link to the original models documentation as well as the implemented parameter distributions are shown.

31.1 binary

31.1.1 “rfe”

Base Class Documenation: sklearn.feature_selection.RFE

Param Distributions

0. “base rfe”

```
n_features_to_select: None
```

1. “rfe num feats dist”

```
n_features_to_select: ng.p.Scalar(init=.5, lower=.1, upper=.99)
```

31.1.2 “selector”

Base Class Documenation: *BPt.extensions.Feat_Selectors.FeatureSelector*

Param Distributions

0. “random”

```
mask: 'sets as random features'
```

1. “searchable”

```
mask: 'sets as hyperparameters'
```

31.1.3 “univariate selection c”

Base Class Documenation: sklearn.feature_selection.SelectPercentile

Param Distributions

0. “base univar fs classifier”

```
score_func: f_classif
percentile: 50
```

1. “univar fs classifier dist”

```
score_func: f_classif
percentile: ng.p.Scalar(init=50, lower=1, upper=99)
```

2. “univar fs classifier dist2”

```
score_func: f_classif
percentile: ng.p.Scalar(init=75, lower=50, upper=99)
```

31.1.4 “variance threshold”

Base Class Documenation: `sklearn.feature_selection.VarianceThreshold`

Param Distributions

0. “default”

```
defaults only
```

31.2 regression

31.2.1 “rfe”

Base Class Documenation: `sklearn.feature_selection.RFE`

Param Distributions

0. “base rfe”

```
n_features_to_select: None
```

1. “rfe num feats dist”

```
n_features_to_select: ng.p.Scalar(init=.5, lower=.1, upper=.99)
```

31.2.2 “selector”

Base Class Documenation: *BPt.extensions.Feat_Selectors.FeatureSelector*

Param Distributions

0. “random”

```
mask: 'sets as random features'
```

1. “searchable”

```
mask: 'sets as hyperparameters'
```

31.2.3 "univariate selection r"

Base Class Documenation: sklearn.feature_selection.SelectPercentile

Param Distributions

0. "base univar fs regression"

```
score_func: f_regression
percentile: 50
```

1. "univar fs regression dist"

```
score_func: f_regression
percentile: ng.p.Scalar(init=50, lower=1, upper=99)
```

2. "univar fs regression dist2"

```
score_func: f_regression
percentile: ng.p.Scalar(init=75, lower=50, upper=99)
```

31.2.4 "variance threshold"

Base Class Documenation: sklearn.feature_selection.VarianceThreshold

Param Distributions

0. "default"

```
defaults only
```

31.3 categorical

31.3.1 "rfe"

Base Class Documenation: sklearn.feature_selection.RFE

Param Distributions

0. "base rfe"

```
n_features_to_select: None
```

1. "rfe num feats dist"

```
n_features_to_select: ng.p.Scalar(init=.5, lower=.1, upper=.99)
```

31.3.2 “selector”

Base Class Documenation: *BPt.extensions.Feat_Selectors.FeatureSelector*

Param Distributions

0. “random”

```
mask: 'sets as random features'
```

1. “searchable”

```
mask: 'sets as hyperparameters'
```

31.3.3 “univariate selection c”

Base Class Documenation: sklearn.feature_selection.SelectPercentile

Param Distributions

0. “base univar fs classifier”

```
score_func: f_classif
percentile: 50
```

1. “univar fs classifier dist”

```
score_func: f_classif
percentile: ng.p.Scalar(init=50, lower=1, upper=99)
```

2. “univar fs classifier dist2”

```
score_func: f_classif
percentile: ng.p.Scalar(init=75, lower=50, upper=99)
```

31.3.4 “variance threshold”

Base Class Documenation: sklearn.feature_selection.VarianceThreshold

Param Distributions

0. “default”

```
defaults only
```

CHAPTER

THIRTYTWO

ENSEMBLE TYPES

Different base obj choices for the *Ensemble* are shown below The exact str indicator, as passed to the *obj* param is represented by the sub-heading (within “”) The avaliable ensembles are further broken down by which can workwith different problem_types. Additionally, a link to the original models documentation as well as the implemented parameter distributions are shown. Also note that ensemble require a few extra params! I.e., in general, all DESlib based ensemble need needs_split = True

32.1 binary

32.1.1 “adaboost classifier”

Base Class Documenation: `sklearn.ensemble.AdaBoostClassifier`

Param Distributions

0. “default”

```
defaults only
```

32.1.2 “aposteriori”

Base Class Documenation: `deslib.dcs.a_posteriori.APosteriori`

Param Distributions

0. “default”

```
defaults only
```

32.1.3 “apriori”

Base Class Documenation: `deslib.dcs.a_priori.APriori`

Param Distributions

0. “default”

```
defaults only
```

32.1.4 "bagging classifier"

Base Class Documenation: `sklearn.ensemble.BaggingClassifier`

Param Distributions

0. "default"

```
defaults only
```

32.1.5 "balanced bagging classifier"

Base Class Documenation: `imblearn.ensemble.BalancedBaggingClassifier`

Param Distributions

0. "default"

```
defaults only
```

32.1.6 "des clustering"

Base Class Documenation: `deslib.des.des_clustering.DESClustering`

Param Distributions

0. "default"

```
defaults only
```

32.1.7 "des knn"

Base Class Documenation: `deslib.des.des_knn.DESKNN`

Param Distributions

0. "default"

```
defaults only
```

32.1.8 "deskl"

Base Class Documenation: `deslib.des.probabilistic.DESKL`

Param Distributions

0. "default"

```
defaults only
```

32.1.9 "desmi"

Base Class Documenation: deslib.des.des_mi.DESMI

Param Distributions

0. "default"

```
defaults only
```

32.1.10 "desp"

Base Class Documenation: deslib.des.des_p.DESP

Param Distributions

0. "default"

```
defaults only
```

32.1.11 "exponential"

Base Class Documenation: deslib.des.probabilistic.Exponential

Param Distributions

0. "default"

```
defaults only
```

32.1.12 "knop"

Base Class Documenation: deslib.des.knop.KNOP

Param Distributions

0. "default"

```
defaults only
```

32.1.13 "knorae"

Base Class Documenation: deslib.des.knora_e.KNORAE

Param Distributions

0. "default"

```
defaults only
```

32.1.14 "knrau"

Base Class Documenation: deslib.des.knora_u.KNORAU

Param Distributions

0. "default"

```
defaults only
```

32.1.15 "lca"

Base Class Documenation: deslib.dcs.lca.LCA

Param Distributions

0. "default"

```
defaults only
```

32.1.16 "logarithmic"

Base Class Documenation: deslib.des.probabilistic.Logarithmic

Param Distributions

0. "default"

```
defaults only
```

32.1.17 "mcb"

Base Class Documenation: deslib.dcs.mcb.MCB

Param Distributions

0. "default"

```
defaults only
```

32.1.18 "metades"

Base Class Documenation: deslib.des.meta_des.METADES

Param Distributions

0. "default"

```
defaults only
```

32.1.19 "min dif"

Base Class Documenation: deslib.des.probabilistic.MinimumDifference

Param Distributions

0. "default"

```
defaults only
```

32.1.20 "mla"

Base Class Documenation: deslib.dcs.mla.MLA

Param Distributions

0. "default"

```
defaults only
```

32.1.21 "ola"

Base Class Documenation: deslib.dcs.ola.OLA

Param Distributions

0. "default"

```
defaults only
```

32.1.22 "rank"

Base Class Documenation: deslib.dcs.rank.Rank

Param Distributions

0. "default"

```
defaults only
```

32.1.23 "rrc"

Base Class Documenation: deslib.des.probabilistic.RRC

Param Distributions

0. "default"

```
defaults only
```

32.1.24 "single best"

Base Class Documenation: deslib.static.single_best.SingleBest

Param Distributions

0. "default"

```
defaults only
```

32.1.25 "stacked"

Base Class Documenation: deslib.static.stacked.StackedClassifier

Param Distributions

0. "default"

```
defaults only
```

32.1.26 "stacking classifier"

Base Class Documenation: BPt.pipeline.Ensembles.BPtStackingClassifier

Param Distributions

0. "default"

```
defaults only
```

32.1.27 "voting classifier"

Base Class Documenation: BPt.pipeline.Ensembles.BPtVotingClassifier

Param Distributions

0. "voting classifier"

```
voting: 'soft'
```

32.2 regression

32.2.1 "adaboost regressor"

Base Class Documenation: sklearn.ensemble.AdaBoostRegressor

Param Distributions

0. "default"

```
defaults only
```

32.2.2 "bagging regressor"

Base Class Documenation: sklearn.ensemble.BaggingRegressor

Param Distributions

0. "default"

```
defaults only
```

32.2.3 "stacking regressor"

Base Class Documenation: BPt.pipeline.Ensembles.BPtStackingRegressor

Param Distributions

0. "default"

```
defaults only
```

32.2.4 "voting regressor"

Base Class Documenation: BPt.pipeline.Ensembles.BPtVotingRegressor

Param Distributions

0. "default"

```
defaults only
```

32.3 categorical

32.3.1 "adaboost classifier"

Base Class Documenation: sklearn.ensemble.AdaBoostClassifier

Param Distributions

0. "default"

```
defaults only
```

32.3.2 "aposteriori"

Base Class Documenation: deslib.dcs.a_posteriori.APosteriori

Param Distributions

0. "default"

```
defaults only
```

32.3.3 "apriori"

Base Class Documenation: deslib.dcs.a_priori.APriori

Param Distributions

0. "default"

```
defaults only
```

32.3.4 "bagging classifier"

Base Class Documenation: sklearn.ensemble.BaggingClassifier

Param Distributions

0. "default"

```
defaults only
```

32.3.5 "balanced bagging classifier"

Base Class Documenation: imblearn.ensemble.BalancedBaggingClassifier

Param Distributions

0. "default"

```
defaults only
```

32.3.6 "des clustering"

Base Class Documenation: deslib.des.des_clustering.DESClustering

Param Distributions

0. "default"

```
defaults only
```

32.3.7 "des knn"

Base Class Documenation: deslib.des.des_knn.DESKNN

Param Distributions

0. "default"

```
defaults only
```

32.3.8 "deskl"

Base Class Documenation: deslib.des.probabilistic.DESKL

Param Distributions

0. "default"

```
defaults only
```

32.3.9 "desmi"

Base Class Documenation: deslib.des.des_mi.DESMI

Param Distributions

0. "default"

```
defaults only
```

32.3.10 "desp"

Base Class Documenation: deslib.des.des_p.DESP

Param Distributions

0. "default"

```
defaults only
```

32.3.11 "exponential"

Base Class Documenation: deslib.des.probabilistic.Exponential

Param Distributions

0. "default"

```
defaults only
```

32.3.12 "knop"

Base Class Documenation: deslib.des.knop.KNOP

Param Distributions

0. "default"

```
defaults only
```

32.3.13 "knorae"

Base Class Documenation: deslib.des.knora_e.KNORAE

Param Distributions

0. "default"

```
defaults only
```

32.3.14 "knrau"

Base Class Documenation: deslib.des.knora_u.KNORAU

Param Distributions

0. "default"

```
defaults only
```

32.3.15 "lca"

Base Class Documenation: deslib.dcs.lca.LCA

Param Distributions

0. "default"

```
defaults only
```

32.3.16 "logarithmic"

Base Class Documenation: deslib.des.probabilistic.Logarithmic

Param Distributions

0. "default"

```
defaults only
```

32.3.17 "mcb"

Base Class Documenation: deslib.dcs.mcb.MCB

Param Distributions

0. "default"

```
defaults only
```

32.3.18 "metades"

Base Class Documenation: deslib.des.meta_des.METADES

Param Distributions

0. "default"

```
defaults only
```

32.3.19 "min dif"

Base Class Documenation: deslib.des.probabilistic.MinimumDifference

Param Distributions

0. "default"

```
defaults only
```

32.3.20 "mla"

Base Class Documenation: deslib.dcs.mla.MLA

Param Distributions

0. "default"

```
defaults only
```

32.3.21 "ola"

Base Class Documenation: deslib.dcs.ola.OLA

Param Distributions

0. "default"

```
defaults only
```

32.3.22 "rank"

Base Class Documenation: deslib.dcs.rank.Rank

Param Distributions

0. "default"

```
defaults only
```

32.3.23 "rrc"

Base Class Documenation: deslib.des.probabilistic.RRC

Param Distributions

0. "default"

```
defaults only
```

32.3.24 "single best"

Base Class Documenation: deslib.static.single_best.SingleBest

Param Distributions

0. "default"

```
defaults only
```

32.3.25 "stacked"

Base Class Documenation: deslib.static.stacked.StackedClassifier

Param Distributions

0. "default"

```
defaults only
```

32.3.26 "stacking classifier"

Base Class Documenation: BPt.pipeline.Ensembles.BPtStackingClassifier

Param Distributions

0. "default"

```
defaults only
```

32.3.27 "voting classifier"

Base Class Documenation: BPt.pipeline.Ensembles.BPtVotingClassifier

Param Distributions

0. "voting classifier"

```
voting: 'soft'
```

The backend library for conducting hyper-parameter searches within the BPt is nevergrad, a library developed by facebook. They implement a whole bunch of methods, and have limited documentation explaining them. This page will try to break down the different avaliable options.

CHAPTER

THIRTYTHREE

RANDOM SEARCH

Background: https://en.wikipedia.org/wiki/Random_search

There are few optional parameters you may specify in order to produce different random search behavior.

middle_point Optional enforcement of the first suggested point as zero. Either,

- True : Enforced middle suggested point
- False : Not enforced

(default = False)

opposition_mode symmetrizes exploration wrt the center: (e.g. https://ieeexplore.ieee.org/document/4424748) - "opposite" : full symmetry - "quasi" : Random * symmetric - None

(default = None)

33.1 'RandomSearch'

```
Defaults Only
```

33.2 'RandomSearchPlusMiddlePoint'

```
middle_point: True
```

33.3 'QORandomSearch'

```
opposition_mode: 'quasi'
```

33.4 ORandomSearch

```
opposition_mode: 'opposite'
```

CHAPTER

THIRTYFOUR

ONE SHOT OPTIMIZATION

Implemented one-hot optimization methods which are 'hopefully better than random search by ensuring more uniformity'. The algorithms vary on the following parameters,

sampler Type of random sampling. Either,

- 'Halton' : A low quality sampling method when the dimension is high
- 'Hammersley' : Hammersley sampling
- 'LHS' : Latin Hypercube Sampling

(default = 'Halton')

scrambled Adds scrambling to the search

- True : scrambling is added
- False : scrambling is not added

(default = False)

middle_point Optional enforcement of the first suggested point as zero. Either,

- True : Enforced middle suggested point
- False : Not enforced

(default = False)

cauchy Use Cauchy inverse distribution instead of Gaussian when fitting points to real space Either,

- True : Use the cauchy ditribution
- False : Use a gaussian distribution

(default = False)

rescaled Rescale the sampling pattern to reach the boundaries. Either,

- True : rescale
- False : don't rescale

(default = False)

autorescale Perform auto-rescaling

- True : Auto rescale
- False : don't auto rescale

(default = False)

recommendation_rule Method for selecting best point. Either,

- 'average_of_best' : take average over all better then median
- 'pessimistic' : selecting pessimistic best

(default = 'pessimistic')

opposition_mode symmetrizes exploration wrt the center: (e.g. https://ieeexplore.ieee.org/document/4424748) - "opposite" : full symmetry - "quasi" : Random * symmetric - None

(default = None)

34.1 'HaltonSearch'

```
Defaults Only
```

34.2 'HaltonSearchPlusMiddlePoint'

```
middle_point: True
```

34.3 'ScrHaltonSearch'

```
scrambled: True
```

34.4 'ScrHaltonSearchPlusMiddlePoint'

```
middle_point: True
scrambled: True
```

34.5 'HammersleySearch'

```
sampler: 'Hammersley'
```

34.6 'HammersleySearchPlusMiddlePoint'

```
sampler: 'Hammersley'
middle_point: True
```

34.7 'ScrHammersleySearchPlusMiddlePoint'

```
scrambled: True
sampler: 'Hammersley'
middle_point: True
```

34.8 'ScrHammersleySearch'

```
sampler: 'Hammersley'
scrambled: True
```

34.9 'OScrHammersleySearch'

```
sampler: 'Hammersley'
scrambled: True
opposition_mode: 'opposite'
```

34.10 'QOScrHammersleySearch'

```
sampler: 'Hammersley'
scrambled: True
opposition_mode: 'quasi'
```

34.11 'CauchyScrHammersleySearch'

```
cauchy: True
sampler: 'Hammersley'
scrambled: True
```

34.12 'LHSSearch'

```
sampler: 'LHS'
```

34.13 'CauchyLHSSearch'

```
sampler: 'LHS'
cauchy: True
```

34.14 'MetaRecentering'

```
cauchy: False
autorescale: True
sampler: 'Hammersley'
```

34.15 'MetaTuneRecentering'

```
cauchy: False
autorescale: "autotune"
sampler: 'Hammersley'
scrambled: True
```

34.16 HAvgMetaRecentering

```
cauchy: False
autorescale: True,
sampler: "Hammersley"
scrambled: True
recommendation_rule: "average_of_hull_best"
```

34.17 AvgMetaRecenteringNoHull

```
cauchy: False
autorescale: True
sampler: "Hammersley"
scrambled: True,
recommendation_rule: "average_of_exp_best"
```

CHAPTER

THIRTYFIVE

ONE PLUS ONE

This is a family of evolutionary algorithms that use a technique called 1+1 or One Plus One. 'simple but sometimes powerful class of optimization algorithm. We use asynchronous updates, so that the 1+1 can actually be parallel and even performs quite well in such a context - this is naturally close to 1+lambda.'

The algorithms vary on the following parameters,

noise_handling How re-evaluations are performed.

- 'random' : a random point is reevaluated regularly
- 'optimistic' : the best optimistic point is reevaluated regularly
- a coefficient can to tune the regularity of these reevaluations

(default = (None, .05))

mutation The strategy for producing changes / mutations.

- 'gaussian' : standard mutation by adding a Gaussian random variable (with progressive widening) to the best pessimistic point
- 'cauchy' : same as Gaussian but with a Cauchy distribution.
- 'discrete' : discrete distribution
- 'fastga' : FastGA mutations from the current best
- 'doublefastga' : double-FastGA mutations from the current best (Doerr et al, Fast Genetic Algorithms, 2017)
- 'portfolio' : Random number of mutated bits (called niform mixing in Dang & Lehre 'Self-adaptation of Mutation Rates in Non-elitist Population', 2016)

(default = 'gaussian')

crossover Optional additional of genetic cross over.

- True : Add genetic crossover step every other step.
- False : No crossover.

(default = False)

35.1 'OnePlusOne'

```
Defaults Only
```

35.2 'NoisyOnePlusOne'

```
noise_handling: 'random'
```

35.3 'OptimisticNoisyOnePlusOne'

```
noise_handling: 'optimistic'
```

35.4 'DiscreteOnePlusOne'

```
mutation: 'discrete'
```

35.5 'DiscreteLenglerOnePlusOne'

```
mutation: 'lengler'
```

35.6 'AdaptiveDiscreteOnePlusOne'

```
mutation: "adaptive"
```

35.7 'AnisotropicAdaptiveDiscreteOnePlusOne'

```
mutation: "coordinatewise_adaptive"
```

35.8 'DiscreteBSOOnePlusOne'

```
mutation: "discreteBSO"
```

35.9 'DiscreteDoerrOnePlusOne'

```
mutation: "doerr"
```

35.10 'CauchyOnePlusOne'

```
mutation: "cauchy"
```

35.11 'OptimisticDiscreteOnePlusOne'

```
noise_handling: 'optimistic'
mutation: 'discrete'
```

35.12 'NoisyDiscreteOnePlusOne'

```
noise_handling: ('random', 1.0)
mutation: 'discrete'
```

35.13 'DoubleFastGADiscreteOnePlusOne'

```
mutation: 'doublefastga'
```

35.14 'FastGADiscreteOnePlusOne'

```
mutation: 'fastga'
```

35.15 'RecombiningPortfolioOptimisticNoisyDiscreteOnePlusOne'

```
crossover: True
mutation: 'portfolio'
noise_handling: 'optimistic'
```

CHAPTER

THIRTYSIX

CMA

This refers to the covariance matrix adaptation evolutionary optimzation strategy Background: https://en.wikipedia.org/wiki/CMA-ES

The following parameters are changed

diagonal To use the diagonal version of CMA (advised in large dimensions)

- True : Use diagonal
- False : Don't use diagonal

fcmaes To use fast implementation, doesn't support diagonal=True. produces equivalent results, preferable for high dimensions or if objective function evaluation is fast.

36.1 'CMA'

```
diagonal: False
fcmaes: False
```

36.2 'DiagonalCMA'

```
diagonal: True
fcmaes: False
```

36.3 'FCMA'

```
diagonal: False
fcmaes: True
```

Further variants of CMA include CMA with test based population size adaption. It sets Population-size equal to lambda = 4 x dimension. It further introduces the parameters:

popsize_adaption To use CMA with popsize adaptation

- True : Use popsize adaptation
- False : Don't...

covariance_memory Use covariance_memory

- True : Use covariance
- False : Don't...

36.4 'EDA'

```
popsize_adaption: False
covariance_memory: False
```

36.5 'PCEDA'

```
popsize_adaption: True
covariance_memory: False
```

36.6 'MPCEDA'

```
popsize_adaption: True
covariance_memory: True
```

36.7 'MEDA'

```
popsize_adaption: False
covariance_memory: True
```

CHAPTER

THIRTYSEVEN

EVOLUTION STRATEGIES

Experimental evolution-strategy-like algorithms. Seems to use mutations and cross-over. The following parameters can be changed

recombination_ratio If 1 then will recombine all of the population, if 0 then won't use any combinations just mutations

(default = 0)

popsize The number of individuals in the population

(default = 40)

offsprings The number of offspring from every generation

(default = None)

only_offsprings If true then only keep offspring, none of the original population.

(default = False)

ranker Either 'simple' or 'nsga2'

37.1 'ES'

```
recombination_ratio: 0
popsize: 40
offsprings: 60
only_offsprings: True
ranker: 'simple'
```

37.2 'RecES'

```
recombination_ratio:1
popsize: 40
offsprings: 60
only_offsprings: True
ranker: 'simple'
```

37.3 'RecMixES'

```
recombination_ratio: 1
popsize: 40
offsprings: 20
only_offsprings: False
ranker: 'simple'
```

37.4 'RecMutDE'

```
recombination_ratio: 1
popsize: 40
offsprings: None
only_offsprings: False
ranker: 'simple'
```

37.5 'MixES'

```
recombination_ratio: 0
popsize: 40
offsprings: 20
only_offsprings: False
ranker: 'simple'
```

37.6 'MutDE'

```
recombination_ratio: 0
popsize: 40
offsprings: None
only_offsprings: False
ranker: 'simple'
```

37.7 'NSGAIIES'

```
recombination_ratio: 0
popsize: 40
offsprings: 60
only_offsprings: True
ranker: "nsga2"
```

CHAPTER

THIRTYEIGHT

DIFFERENTIAL EVOLUTION

Background: https://en.wikipedia.org/wiki/Differential_evolution

In the below descriptions the different DE choices vary on a few different parameters.

initialization The algorithm/distribution used for the initialization phase. Either,

- 'LHS' : Latin Hypercube Sampling
- 'QR' : Quasi-Random
- 'gaussian' : Normal Distribution

(default = 'gaussian')

scale The scale of random component of the updates

Either,

- 'mini' : 1 / sqrt(dimension)
- 1 : no change

(default = 1)

crossover The crossover rate value / strategy used during DE. Either,

- 'dimension' : crossover rate of 1 / dimension
- 'random' : different random (uniform) crossover rate at each iteration
- 'onepoint' : one point crossover
- 'twopoints' : two points crossover

(default = .5)

popsize The size of the population to use. Either,

- 'standard' : max(num_workers, 30)
- 'dimension' : max(num_workers, 30, dimension +1)
- 'large' : max(num_workers, 30, 7 * dimension)

Note: dimension refers to the dimensions of the hyperparameters being searched over. 'standard' by default.s

(default = 'standard')

recommendation Choice of the criterion for the best point to recommend. Either,

- 'optimistic' : best
- 'noisy' : add noise to choice of best

(default = 'optimistic')

38.1 'DE'

```
Defaults Only
```

38.2 'OnePointDE'

```
crossover: 'onepoint'
```

38.3 'TwoPointsDE'

```
crossover: 'twopoint'
```

38.4 'LhsDE'

```
initialization: 'LHS'
```

38.5 'QrDE'

```
initialization: 'QE'
```

38.6 'MiniDE'

```
scale: 'mini'
```

38.7 'MiniLhsDE'

```
initialization: 'LHS'
scale: 'mini'
```

38.8 'MiniQrDE'

```
initialization: 'QE'
scale: 'mini'
```

38.9 'NoisyDE'

```
recommendation: 'noisy'
```

38.10 'AlmostRotationInvariantDE'

```
crossover: .9
```

38.11 'AlmostRotationInvariantDEAndBigPop'

```
crossover: .9
popsize: 'dimension'
```

38.12 'RotationInvariantDE'

```
crossover: 1
popsize: 'dimension'
```

38.13 'BPRotationInvariantDE'

```
crossover: 1
popsize: 'large'
```

CHAPTER

THIRTYNINE

ALGORITHM SELECTION

Algorithm selection works by first splitting the search budget up between trying different search algorithms, and the 'budget_before_choosing' is up, it uses the rest of the search budget on the strategy that did the best.

In the case that budget_before_choosing is 1, then the algorithm is a passive portfolio of the different options, and will split the full budget between all of them.

The parameter options refers to the algorithms it tries before choosing.

39.1 'ASCMA2PDEthird'

```
options: ['CMA', 'TwoPointsDE']
budget_before_choosing: 1/3
```

39.2 'ASCMADEQRthird'

```
options: ['CMA', 'LhsDE', 'ScrHaltonSearch']
budget_before_choosing: 1/3
```

39.3 'ASCMADEthird'

```
options: ['CMA', 'LhsDE']
budget_before_choosing: 1/3
```

39.4 'TripleCMA'

```
options: ['CMA', 'CMA', 'CMA']
budget_before_choosing: 1/3
```

39.5 'MultiCMA'

```
options: ['CMA', 'CMA', 'CMA']
budget_before_choosing: 1/10
```

39.6 'MultiScaleCMA'

```
options: ['CMA', 'ParametrizedCMA(scale=1e-3)', 'ParametrizedCMA(scale=1e-6)']
budget_before_choosing: 1/3
```

39.7 'Portfolio'

```
options: ['CMA', 'TwoPointsDE', 'ScrHammersleySearch']
budget_before_choosing: 1
```

39.8 'ParaPortfolio'

```
options: ['CMA', 'TwoPointsDE', 'PSO', 'SQP', 'ScrHammersleySearch']
budget_before_choosing: 1
```

39.9 'SQPCMA'

```
options: ['CMA', n_jobs - n_jobs // 2 'SQP']
budget_before_choosing: 1
```

CHAPTER

FORTY

COMPETENCE MAPS

Competence Maps essentially just automatically select an algorithm based on the parameters passed, the number of workers, the budget, ect. . .

40.1 'NGO'

Nevergrad optimizer by competence map., Based on One-Shot options

40.2 'NGOpt'

Nevergrad optimizer by competence map.

40.3 'CM'

Competence map, simplest

40.4 'CMandAS'

Competence map, with algorithm selection in one of the cases

40.5 'CMandAS2'

Competence map, with algorithm selection in one of the cases (3 CMAs).

40.6 'CMandAS3'

Competence map, with algorithm selection in one of the cases (3 CMAs).

40.7 'Shiva'

"Shiva" choices - "Nevergrad optimizer by competence map"

CHAPTER

FORTYONE

MISC.

These optimizers did not seem to naturally fall into a category. Brief descriptions are listed below.

41.1 'NaiveIsoEMNA'

Estimation of Multivariate Normal Algorithm This algorithm is quite efficient in a parallel context, i.e. when the population size is large.

41.2 'TBPSA'

Test-based population-size adaptation, for noisy problems where the best points will be an average of the final population.

41.3 'NaiveTBPSA'

Test-based population-size adaptation Where the best point is the best point, no average across final population.

41.4 'NoisyBandit'

Noisy bandit simple optimization

41.5 'PBIL'

Population based incremental learning "Implementation of the discrete algorithm PBIL" https://www.ri.cmu.edu/pub_files/pub1/baluja_shumeet_1994_2/baluja_shumeet_1994_2.pdf

41.6 'PSO'

Standard Particle Swarm Optimisation, but no randomization of the population order.

41.7 'SQP'

Scipy Minimize Base See: https://docs.scipy.org/doc/scipy-1.1.0/reference/generated/scipy.optimize.minimize.html Note: does not support multiple jobs at once.

41.8 'SPSA'

The First order SPSA algorithm, See: https://en.wikipedia.org/wiki/Simultaneous_perturbation_stochastic_approximation Note: does not support multiple jobs at once.

41.9 'SplitOptimizer'

Combines optimizers, each of them working on their own variables. By default uses CMA and RandomSearch's

41.10 'cGA'

Implementation of the discrete Compact Genetic Algorithm (cGA) https://pdfs.semanticscholar.org/4b0b/5733894ffc0b2968ddaab15d61751b87847a.pdf

41.11 'chainCMAPowell'

A chaining consists in running algorithm 1 during T1, then algorithm 2 during T2, then algorithm 3 during T3, etc. Each algorithm is fed with what happened before it. This 'chainCMAPowell' chains first 'CMA' then the 'Powell' optimizers. Note: does not support multiple jobs at once.

CHAPTER

FORTYTWO

EXPERIMENTAL VARIANTS

Nevergrad also comes with a number of Experimental variants, to see all of the different options run:

```
import nevergrad as ng
import nevergrad.optimization.experimentalvariants
print(sorted(ng.optimizers.registry.keys()))
```

CHAPTER

FORTYTHREE

FEAT IMPORTANCES

Determining Feature Importance is an important step in making sense of ML output. The following e-book provides an extensive background on different feature importance methods, in particular this section: https://christophm.github.io/interpretable-ml-book/agnostic.html

Different feat importances can be broadly considered as either global (one value per feature) or local (features can ussually be averaged over to produce a global measure of feature importance). This param is the 'scopes'. A feature importance can further be calculated on different parts of the data, or rather the split can differ. Splits can be either, train only, test only or both. Train only refers to calculating feature importances based on only the trained model, or with access to only how that model performs on the same data it was trained. For example: if computing permutation feature importance, train only would calculate this score on the training set, and therefore describe the behavior of the trained model, but not neccisarily how well those feature generalize. In this case if the model is overfit, then the train only feature importances would most likely not be meaningful. Test only is determined based on performance or predictions made on the test or validation set only. This method, using the example from before, would indicate how the trained model genralizes to new unseen data.

scopes The scopes in which the feature importance method calculates over.

- 'local' : One feature importance per data point
- 'global' : One feature importance per feature

split What portion of the data is used to determine feature importance

- 'train' : On only the training data
- 'test' : On only the testing or validation data
- 'all' : On both the training and the testing data

43.1 "base"

Base feature importance refers to the feature importance as calculated automatically by the underlying model. For example, for linear models, base feature importance reflects the beta weights of the trained model. For tree based models, e.g., random forest, the feature importances represent the calculated feature importances (gini split index in this case).

Base feature importances have scope == 'global' and split == 'train'.

Param Distributions

0. "default"

```
defaults only
```

43.2 "shap"

This computes Shap feature importance, which can be read about in more detail at: https://github.com/slundberg/shap
A good more general description is also found at: https://christophm.github.io/interpretable-ml-book/shap.html

This base "shap" setting uses split == 'test', so importances are determined from the test or validation set.

Shap generates both local and global feature importances.

The underlying shap method with change if a linear underlying model, tree-based or anything else. Linear and tree-based models run quickly, but take care in computing shap values for anything else! In this case, the kernel shap computer is used, which approximates the shap values and is hugely compute intensive!! See: https://shap.readthedocs.io/en/latest/ for more info on the TreeExplainer and KernelExplainer.

The parameters that can be changed in shap are as follows (descriptions from the shap documentation):

shap__global__avg_abs This parameter is considered regardless of the underlying model. If set to True, then when computing global feature importance from the initially computed local shap feature importance the average of the absolute value will be taken! When set to False, the average value will be taken. One might want to set this to True if only concerned with the magnitude of feature importance, rather than the sign.

(default = False)

shap__linear__feature_dependence Only used with linear base models. There are two ways we might want to compute SHAP values, either the full conditional SHAP values or the independent SHAP values. For independent SHAP values we break any dependence structure between features in the model and so uncover how the model would behave if we intervened and changed some of the inputs. For the full conditional SHAP values we respect the correlations among the input features, so if the model depends on one input but that input is correlated with another input, then both get some credit for the model's behavior. The independent option stays "true to the model" meaning it will only give credit to features that are actually used by the model, while the correlation option stays "true to the data" in the sense that it only considers how the model would behave when respecting the correlations in the input data.

(default = 'independent')

shap__linear__nsamples Only used with linear base models. Number of samples to use when estimating the transformation matrix used to account for feature correlations. Only used if shap__linear__feature_dependence is set to 'correlation'.

(default = 1000)

shap__tree__feature_perturbation Only used with tree base models. Since SHAP values rely on conditional expectations we need to decide how to handle correlated (or otherwise dependent) input features. The "interventional" approach breaks the dependencies between features according to the rules dictated by casual inference (Janzing et al. 2019). Note that the "interventional" option requires a background dataset and its runtime scales linearly with the size of the background dataset you use. Anywhere from 100 to 1000 random background samples are good sizes to use. The "tree_path_dependent" approach is to just follow the trees and use the number of training examples that went down each leaf to represent the background distribution. This approach does not require a background dataset and so is used by default when no background dataset is provided.

(default = 'tree_path_dependent')

shap__tree__model_output Only used with tree base models. What output of the model should be explained. If "margin" then we explain the raw output of the trees, which varies by model (for binary classification in XGBoost this is the log odds ratio). If "probability" then we explain the output of the model transformed into probability space (note that this means the SHAP values now sum to the probability output of the model). If "log_loss" then we explain the log base e of the model loss function, so that the SHAP values sum up to the log loss of the model for each sample. This is helpful for breaking down model performance by feature. Currently the probability and log_loss options are only supported when feature_dependence="independent".

(default = 'margin')

shap__tree__tree_limit Only used with tree base models. Limit the number of trees used by the model. By default None means no use the limit of the original model, and -1 means no limit.

(default = None)

shap__kernel__nkmean Used when the underlying model is not linear or tree based. This setting offers a speed up to the kernel estimator by replacing the background dataset with a kmeans representation of the data. Set this option to None in order to use the full dataset directly, otherwise the int passed will the determine 'k' in the kmeans algorithm.

(default = 10)

shap__kernel__nsamples Used when the underlying model is not linear or tree based. Number of times to re-evaluate the model when explaining each prediction. More samples lead to lower variance estimates of the SHAP values. The 'auto' setting uses nsamples = 2 * X.shape[1] + 2048.

(default = 'auto')

shap__kernel__l1_reg

Used when the underlying model is not linear or tree based. The l1 regularization to use for feature selection (the estimation procedure is based on a debiased lasso). The auto option currently uses "aic" when less that 20% of the possible sample space is enumerated, otherwise it uses no regularization. THE BEHAVIOR OF "auto" WILL CHANGE in a future version to be based on num_features instead of AIC. The "aic" and "bic" options use the AIC and BIC rules for regularization. Using "num_features(int)" selects a fix number of top features. Passing a float directly sets the "alpha" parameter of the sklearn.linear_model.Lasso model used for feature selection.

(default = 'aic')

Param Distributions

0. "base shap"

```
shap__global__avg_abs: False
shap__linear__feature_dependence: 'independent'
shap__linear__nsamples: 1000
shap__tree__feature_perturbation: 'tree_path_dependent'
shap__tree__model_output: 'margin'
shap__tree__tree_limit: None
shap__kernel__nkmean: 10
shap__kernel__nsamples: 'auto'
shap__kernel__l1_reg: 'aic'
```

43.3 "shap train"

See above "shap", this option simply changes the split to computing shap values on the training set. The parameters are the same.

43.4 "shap all"

See above "shap", this option simply changes the split to computing shap values on both the training and testing/validation set. The parameters are the same.

43.5 “perm”

This refers to computing feature importance through a permutation and predict strategy, For more info see: https://christophm.github.io/interpretable-ml-book/feature-importance.html

Note the following article may be of interest before deciding to use permuation feature importance: https://arxiv.org/pdf/1905.03151.pdf

This base “perm” setting using split == ‘test’, so importances are determined from the test or validation set.

The ‘perm__n_perm’ parameter determines the number of time each feature column is permuted.

Param Distributions

0. “base perm”

```
perm__n_perm: 10
```

43.6 “perm train”

See above “perm”, this option simply changes the split to computing permutation values on both the training and testing/validation set. The parameters are the same.

43.7 “perm all”

See above “perm”, this option simply changes the split to computing permutation values on both the training and testing/validation set. The parameters are the same.

CHAPTER

FORTYFOUR

EXTENSIONS

44.1 Winsorizer

class BPt.extensions.Scalers.**Winsorizer**(*quantile_range=(5, 95)*, *copy=True*)

This Scaler performs winzorization, or clipping by feature.

Parameters

- **quantile_range** (*tuple (q_min, q_max), 0.0 < q_min < q_max < 100.0*) – Default: (5.0, 95.0), the lower and upper range in which to clip values to.
- **copy** (*boolean, optional, default is True*) – Make a copy of the data.

44.2 FeatureSelector

class BPt.extensions.Feat_Selectors.**FeatureSelector**(*mask='sets as random features'*)

Custom BPt feature selector for integrating in feature selection with a hyper-parameter search.

Parameters **mask** (*{'sets as random features', 'sets as hyperparameters'}*) –

- 'sets as random features': Use random features.
- 'sets as hyperparameters': Each feature is set as a hyperparameter, such that the parameter search can tune if each feature is included or not.

44.3 SurfLabels

class BPt.extensions.Loaders.**SurfLabels**(*labels*, *background_label=0*, *mask=None*, *strategy='mean'*, *vectorize=True*)

This class functions simmilar to NiftiLabelsMasker from nilearn, but instead is for surfaces (though it could work on a cifti image too).

Parameters

- **labels** (*str or array-like*) – This should represent an array, of the same size as the data dimension, as a mask with unique integer values for each ROI. You can also pass a str location in which to load in this array (though the saved file must be loadable by either numpy.load, or if not a numpy array, will try and load with nilearn.surface.load_surf_data(), which you will need nilearn installed to use.)

- **background_labels** (*int, array-like of int, optional*) – This parameter determines which label, if any, in the corresponding passed labels, should be treated as 'background' and therefore no ROI calculated for that value or values. You may pass either a single interger value, an array-like of integer values.

  If not background label is desired, just pass a label which doesn't exist in any of the data, e.g., -100.

  ```
  default = 0
  ```

- **mask** (*None, str or array-like, optional*) – This parameter allows you to optional pass a mask of values in which to not calculate ROI values for. This can be passed as a str or array-like of values (just like labels), and should be comprised of a boolean array (or 1's and 0's), where a value of 1 means that value will be ignored (set to background label) should be kept, and a value of 0, for that value should be masked away. This array should have the same length as the passed *labels*.

  ```
  default = None
  ```

- **strategy** (*specific str, custom_func, optional*) – This parameter dictates the function to be applied to each data's ROI's individually, e.g., mean to calculate the mean by ROI.

  If a str is passed, it must correspond to one of the below preset options:

  - **'mean'** Calculate the mean with np.mean
  - **'sum'** Calculate the sum with np.sum
  - **'min' or 'minimum** Calculate the min value with np.min
  - **'max' or 'maximum** Calculate the max value with np.max
  - **'std' or 'standard_deviation'** Calculate the standard deviation with np.std
  - **'var' or 'variance'** Calculate the variance with np.var

  If a custom function is passed, it must accept two arguments, custom_func(X_i, axis=data_dim), X_i, where X_i is a subjects data array where that subjects data corresponds to labels == some class i, and can potentially be either a 1D array or 2D array, and an axis argument to specify which axis is the data dimension (e.g., if calculating for a time-series [n_timepoints, data_dim], then data_dim = 1, if calculating for say stacked contrasts where [data_dim, n_contrasts], data_dim = 0, and lastly for a 1D array, data_dim is also 0.

  ```
  default = 'mean'
  ```

- **vectorize** (*bool, optional*) – If the returned array should be flattened to 1D. E.g., if the last step in a set of loader steps this should be True, if before a different step it may make sense to set to False.

  ```
  default = True
  ```

44.4 SurfMaps

class BPt.extensions.Loaders.**SurfMaps**(*maps*, *strategy='auto'*, *mask=None*, *vectorize=True*)

Simmilar to NiftiMapsMasker from nilearn but for surfaces, and designed to work with BPt Loader.

This object calculates the signal for each of the passed maps as extracted from the input during fit, and returns for each map a value.

maps [str or array-like, optional] This parameter represents the maps in which to apply to each surface, where the shape of the passed maps should be (# of vertex, # of maps) or in other words, the size of the data array in the first dimension and the number of maps (i.e., the number of outputted ROIs from fit) as the second dimension.

You may pass maps as either an array-like, or the str file location of a numpy or other valid surface file format array in which to load.

strategy [{'auto', 'ls', 'average'}, optional] The stratgey in which the maps are used to extract signal. If 'ls' is selected, which stands for least squares, the least-squares solution will be used for each region.

Alternatively, if 'average' is passed, then the weighted average value for each map will be computed.

By default 'auto' will be selected, which will use 'average' if the passed maps contain only positive weights, and 'ls' in the case that there are any negative values in the passed maps.

Otherwise, you can set a specific strategy. In deciding which method to use, consider an example. Let's say the fit data X, and maps are

```
data = np.array([1, 1, 5, 5])
maps = np.array([[0, 0],
                 [0, 0],
                 [1, -1],
                 [1, -1]])
```

In this case, the 'ls' method would yield region signals [2.5, -2.5], whereas the weighted 'average' method, would yield [5, 5], notably ignoring the negative weights. This highlights an important limitation to the weighted averaged method, as it does not handle negative values well.

On the other hand, consider changing the maps weights to

```
data = np.array([1, 1, 5, 5])
maps = np.array([[0, 1],
                 [0, 2],
                 [1, 0],
                 [1, 0]])

ls_sol = [5. , 0.6]
average_sol = [5, 1]
```

In this case, we can see that the weighted average gives a maybe more intuative summary of the regions. In general, it depends on what signal you are trying to summarize, and how you are trying to summarize it.

mask [None, str or array-like, optional] This parameter allows you to optional pass a mask of values in which to not calculate ROI values for. This can be passed as a str or array-like of values (just like maps), and should be comprised of a boolean array (or 1's and 0's), where a value of 1 means that value will be ignored (set to 0) should be kept, and a value of 0, for that value should be masked away. This array should have the same length as the passed *maps*. Specifically, where the shape of maps is (size, n_maps), the shape of mask should be (size).

```
default = None
```

vectorize [bool, optional] If the returned array should be flattened to 1D. E.g., if this is the last step in a set of loader steps this should be True. Also note, if the surface data it is being applied to is 1D, then the output will be 1D regardless of this parameter.

```
default = True
```

INDEX

T

V

W