Chapter 12

Brain Predictability Toolbox

Sage Hahn, Nicholas Allgaier, and Hugh Garavan

Abstract

The Brain Predictability toolbox (BPt) is a Python-based library with a unified framework of machine learning (ML) tools designed to work with both tabulated data (e.g., brain-derived, psychiatric, behavioral, and physiological variables) and neuroimaging specific data (e.g., brain volumes and surfaces). The toolbox is designed primarily for 'population'-based predictive neuroimaging; that is to say, machine learning performed across data from multiple participants rather than many data points from a single or small set of participants. The BPt package is suitable for investigating a wide range of neuroimaging-based ML questions. This chapter is a brief introduction to general principles of the toolbox, followed by a specific example of usage.

Key words Machine learning, Python, Neuroimaging, Data science, Data visualization

1 Introduction

In general, the use of a toolbox such as Brain Predictability toolbox (BPt) imposes a practical trade-off between flexibility and ease of use. In the case of working with BPt, once the dataset and desired type of analysis are supported, then a number of analysis steps can be handled automatically, thus reducing opportunities for users to make careless errors. Alternatively, if a specific analysis isn't supported (e.g., deep learning classifiers), then BPt will be a poor choice (see Chapter 16 for an example of deep learning).

BPt is designed to be generalizable to different storage and computing requirements. In practice, data storage and computing requirements will depend on both the dataset of interest as well as predictive questions of interest. For example, performing machine learning on surface-projected data directly may require relatively large computational resources, but if the question or ML model of interest is simple, it could be run on a personal computer in a few hours. In general, BPt has been designed with single personal or workstation computing in mind and the vast majority of situations support this use case. However, this is not to say that BPt cannot be

Robert Whelan and Hervé Lemaître (eds.), *Methods for Analyzing Large Neuroimaging Datasets*, Neuromethods, vol. 218, https://doi.org/10.1007/978-1-0716-4260-3_12,

used by a more advanced user for more complex questions on large cloud-based computing clusters. Most functions within BPt allow for easy integration of multi-core processing to speed up potentially time-intensive ML modeling tasks, which tends to allow performing a greater range of analyses locally. Likewise, data storage requirements will obviously vary when dealing with a single csv file of a few hundred megabytes versus the raw fMRI files from a study with 10,000 participants (20 TB+).

2 Software and Coding

BPt is a Python 3.7+ based package that is tested regularly across all common operating systems (Windows, Mac, and Linux). Use of this package will therefore at the minimum require some proficiency and experience with Python and in setting up Python libraries. Prior experience with the standard data science Python libraries (e.g., pandas, numpy, scikit-learn) [see Resources] is encouraged but not strictly required. Likewise, some prior background knowledge on both neuroimaging and machine learning is expected as BPt tutorial material is not designed to be a user's first exposure to these topics. For new users, it is recommended that the library be used within a computation notebook (e.g., Jupyter notebook or Google Colab). These environments allow for an interactive and iterative approach to coding which is highly recommended when learning and exploring a new library or toolbox. Likewise, most available tutorial material is provided in this base format.

3 General Method

3.1 Inputs

Input data for the toolbox can take a wide range of forms, but generally speaking include outputs from a typical neuroimaging preprocessing pipeline (e.g., the example dataset used for this chapter), plus target and nuisance variables. The easiest data to work with are data already in tabular form (e.g., calculated mean values per region of interest). That said, the toolbox is capable of working with volumetric or surface projected structural or functional MRI (sMRI and fMRI, respectively) data as well. Other modalities, like EEG (electroencephalography), could also be analyzed using the toolbox, but in these cases, it may require additional formatting (as EEG requires quite different preprocessing steps).

There are no specific guidelines in terms of choice of preprocessing pipeline, or choice of parcellation size, atlas, voxel vs. vertex with respect to working with BPt. Instead, as BPt is a general utility toolbox, best practices with respect to all of these choices should be taken in consideration to the broader prediction-based

neuroimaging literature. For the most part, these decisions will depend on the specific modalities employed as well as the predictive target(s) of interest. That said, there are a number of benchmark papers that address these questions empirically, including for surface-based sMRI [1] and functional connectome [2].

A good way of conceptualizing the 'readiness' level of data for machine learning is to consider any transformations that can be computed based solely on a single data point (e.g., a participant's data) versus transformations that utilize information across the entire dataset. That is, in most cases, any participant-level analysis or transformations should be already applied prior to machine learning. Importantly, the BPt toolbox provides support for some of these common data preparation/processing steps, which include: organization of the data, utilities for exploratory data visualization, common transformations such as k-binning and binarization, automatic outlier detection, information on missing data, and other summary measures. This chapter describes use of the interface to access common operations. Additional, more specific features exist as well (e.g., a built-in function to save a whole table of descriptive variables straight to a .docx file and built-in smart merging of index names; *see* Fig. 1).

3.1.1 Sample Size

No specific minimum number of participants are required, but when performing machine learning based experiments larger sample sizes are highly preferred (for a more detailed discussion on why [3]).

3.1.2 Missing Data

Missing data within predictive-based neuroimaging is a common occurrence given the 'messiness' of real-world data. BPt includes utilities both to identify (*see* Fig. 2) and purge existing datasets of missing data or alternatively if necessary to impute values properly within a machine learning pipeline (*see* Fig. 3). Supported strategies

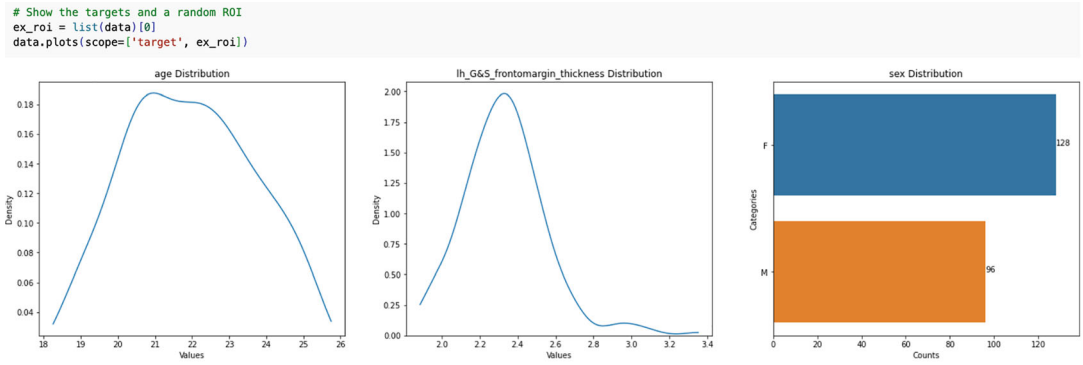


Fig. 1 Example showing build in dataset function for visualizing input data as a collage of plots

```
df.nan_info()
```

```
Loaded NaN Info:
There are: 23848 total missing values
24 columns found with 22 missing values (column name overlap: ['avg'])
23 columns found with 21 missing values (column name overlap: ['avg'])
20 columns found with 19 missing values (column name overlap: ['_surfavg'])
17 columns found with 20 missing values (column name overlap: ['avg'])
11 columns found with 23 missing values (column name overlap: ['avg', '_t'])
```

Fig. 2 Example of built-in function for providing information on patterns of loading missing data

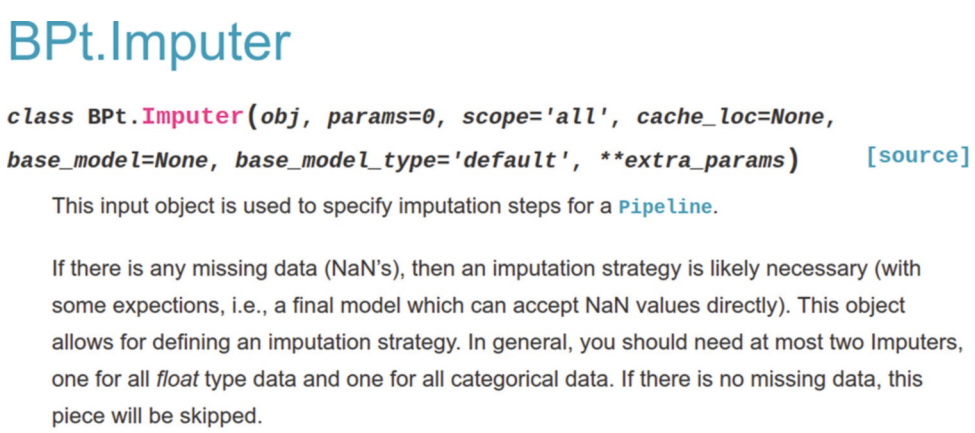


Fig. 3 Screenshot of Imputer pipeline piece documentation

for imputation within a pipeline include mean and median imputation in addition to more complex strategies such as multiple rounds of iterative imputation.

3.2 Data Structure

BPt guides the user in how to structure and answer a question of interest within a predictive framework. Given the inherent vastness of this topic, it is important to note that there is no single 'right' way of doing things, and instead what we present here is a set of general recommendations in which the underlying library has been designed to follow.

3.2.1 Frame a Question

The very first step is to frame a research question of interest in terms of a prediction. For example, if our question of interest is to investigate age-related changes in cortical thickness, then a simple predictive re-framing could be: "how well can cortical thickness predict a participant's age?". What if we had longitudinal data per participant? Then maybe we could ask, how well does cortical thickness predict age at time point 1, what about time point 2, and so on?

```
import BPt as bp

# Load
data = bp.read_csv('quick_start.csv',
                   index_col='participant_id',
                   targets=['age', 'sex'])

# Set sex as a binary variable
data = data.to_binary('sex')
```

Fig. 4 Example of how data saved in a csv can be quickly loaded, and information around which columns are input data and which are target variables quickly set. Likewise, this example shows how columns can be easily transformed, in this case the variable 'sex' is binarized

Note
The key pieces of information to identify after composing a question of interest are: What are the input variables to the prediction? What variable(s) are being predicted? Furthermore, are there any other variables which might influence this prediction in an undesirable way (i.e., potential confounding variables).

3.2.2 *Prepare Data in BPt*

Once a question of interest has been identified, we load it into a Dataset object (*see* Fig. 4), which is a Python class based on the popular Pandas DataFrame. The key point here is that the Dataset object is inherently designed to enforce an explicit organization structure based on the question of interest. The idea is that each column of the Dataset class—where data points within the column are either single values or external references to (e.g., to a saved sMRI file)—are given a role: 'data', 'target' or 'non input'. These roles correspond to the variables used as input to a machine learning algorithm ('data'), the target variables that are predicted ('target') and everything else, including the potential confounding variables ('non input').

There are some pre-modelling steps that, depending on the dataset and the question, might also be explored at this stage, and can be performed using the Dataset object directly. For example, users may want to: generate exploratory plots of the different features in the dataset, remove any data based on status as an outlier, decide if missing data should be kept and imputed, or dropped, apply any pre-requisite transformations that should be applied to the data? (e.g., conversion from strings 'Male', 'Female' to 0 and 1's).

3.2.3 Define a ML Pipeline

A machine learning pipeline is not just the choice of ML model, it is the full set of transformations to the data prior to input to an ML algorithm. This is, in a lot of ways, the area with the most researcher degrees of freedom, as we can think of both the presence or absence of a transformation, as well as the choice of model and that model's parameters as all 'hyper-parameters' of the broader ML pipeline. These could be choices like what brain parcellation to use, to z-score each feature or not, which type of fMRI connectivity metric to use, the type of ML estimator, the parameters associated with that estimator, etc. The number of permutations grows quite rapidly, so in practice how should the researcher decide? We recommend treating each possible 'hyper-parameter' according to the following set of options.

Note
If a parameter is important to the research question, test and report the results by each possible value or a reasonable set of values of interest that this parameter might take. For example, let's say we want to know how our prediction varies by choice of parcellation, so we repeat our full ML experiment with three different parcellations, and report the results of each. Otherwise, if not directly important or related to the question of interest the researcher can either: (1) fix the value ahead of time based on a priori knowledge or best estimate or (2) assign the value through some nested validation strategy (e.g., train-validation/test split or nested K-fold). In general, option 1 is preferable, as it is simpler to both implement and conceptualize fixing a value ahead of time. That said, setting values through nested validation can be useful in certain cases, for example, it is often used for setting hyper-parameters specific to an ML estimator. In other words, option 2 is used as a way to try and improve down-stream performance, with an emphasis on 'try,' as it is difficult in practice to correctly identify the choices which will benefit from this approach.

While designing an ML pipeline can be daunting and introduce lots of researcher degrees of freedom, it is also the area most amenable to creativity. As long as proper validation, as discussed in the next section, is kept in mind, testing and trying new/different pipelines can be an important piece of ML modeling. This becomes especially important when the researcher starts to consider ML modeling in the context of potential confounds, where potential corrections for confounds are themselves steps within the pipeline. That said, especially as a newer researcher, it may be a good

```
results = bp.evaluate(pipeline='ridge_pipe', dataset=data, target='age')
```

Fig. 5 Screenshot showing how a default pipeline can be easily selected when defining a pipeline within an evaluation loop, in this case a default pipeline based on a regularized ridge regression

```
import BPt as bp

pipe = bp.Pipeline([bp.Scaler(obj='robust'),
                    bp.Model(obj='ridge',
                             params=1,
                             param_search=bp.ParamSearch(n_iter=60))])
```

Fig. 6 Customized creation of pipelines

idea to start by replicating previous strategies from the literature that have been found to work well. Default pipelines can be easily specified within BPt (*see* Fig. 5) or alternatively, we can easily customize the creation of pipelines (*see* Fig. 6).

3.2.4 Select and Evaluate According to a Validation Strategy

In order for the results from a ML-based predictive experiment to be valid, some sort of cross or external validation is essential. So how do we decide between say a training-test split between two matched samples and K-fold cross validation on the whole sample? In short, it depends. There is no silver bullet that works for every scenario, but the good news is that for the most part it really shouldn't matter! The most important element to properly using an external validation strategy isn't between threefolds versus ten-folds, but instead is in how the chosen strategy is used. That is to say, the validation data should only be used in answering the main predictive question of interest. If instead the current experiment isn't related to the primary research question, that is to say, the result will not be reported, then the validation data should not be used in any way. Let's consider an explicit example of what *not* to do: Let's say we decide to use a threefold cross validation strategy, predicting age from cortical thickness, and we start by evaluating a simple linear regression model, but it doesn't do very well. Next, we try a random forest model, which does a little better, but still not great, so we try changing a few of its parameters, run the threefold cross validation again, change a few more parameters, and after a little tweaking eventually get a score we are satisfied with. We then report just this result: "a random forest model predicted age, R2=XXX." The issue with the example above is, namely, one of over-using the validation data. By repeatedly testing different models with the same set of validation data, be it through K-fold or a left-aside testing set, we have increased our chances of obtaining an artificially high performance metric through chance alone (i.e., this is a phenomenon pretty similar in nature to p-hacking in classical statistics). Now in this example the fix is fairly

```
cv = bp.CV(splits=3, n_repeats=1, stratify='sex')
```

Fig. 7 Example of defining a cross-validation strategy where a three-fold validation is performed, and further the ratio of 'Males' and 'Females' as defined in variable 'sex' are preserved within every training and validation set

```
cv = bp.CV(splits='site')
```

Fig. 8 Example showing how a validation strategy for performing leave-site-out cross validation can be easily defined

easy. If we want to perform model selection and model hyper-parameter tuning, we can, but as long as both the model selection and hyper-parameter tuning are conducted with nested validation (e.g., on a set-aside training dataset). Fundamentally, it depends on what our ultimate question of interest is. For example, if we are explicitly interested in the difference in performance between different ML models, then it is reasonable to evaluate all of the different models of interest on the validation data, as long as all of their respective performances are reported.

There are of course other potential pitfalls in selecting and employing validation strategies that may vary depending on the underlying complexity of the problem of interest. For example, if using multi-site data, there is a difference between a model generalizing to other participants from the same site (random split k-fold validation) versus generalizing to new participants from unseen sites (group k-fold validation where site is preserved within fold). While choice of optimal strategy will vary, BPt provides an easy interface for employing varied and potentially complex validation strategies, such as internal cross-validation (*see* Fig. 7) or leave-site-out (*see* Fig. 8).

4 Interpreting and Reporting Results

Results from every machine learning based evaluation in BPt return a special results object called 'EvalResults' (*see* Fig. 9). This object stores by default key information related to the conducted experiment, which allows the user to then easily access or additionally compute a range of useful measures. Listed below are some of the available options.

Base common machine learning metrics are provided, across regression, binary, and multi-class predictions, for example R^2, negative mean squared error, ROC AUC (Receiver Operating Characteristic Area Under Curve) [see Glossary], balanced accuracy, and others. In the case of employing a cross-validation strategy like K-fold, these metrics can be accessed either per fold, or averaged across multiple folds (or even the weighted average across folds of different sizes).

```
EvalResults
------------
r2: 0.1027 ± 0.0454
neg_mean_squared_error: -2.83 ± 0.4594

Saved Attributes: ['estimators', 'preds', 'timing', 'estimator', 'train_subjects',
'val_subjects', 'feat_names', 'ps', 'mean_scores', 'std_scores', 'weighted_mean_scores',
'scores', 'fis_', 'coef_', 'cv']

Available Methods: ['to_pickle', 'compare', 'get_X_transform_df', 'get_inverse_fis',
'run_permutation_test', 'get_preds_dfs', 'subset_by', 'get_fis', 'get_coef_',
'permutation_importance']

Evaluated With:
target: age
problem_type: regression
scope: all
subjects: all
random_state: 1
```

Fig. 9 Example showing a string representation of an EvalResults object, with information on saved attributes and methods

```
results.get_preds_dfs()[0]
```

participant_id	predict	y_true
sub-0001	21.909845	25.50
sub-0005	22.126259	24.75
sub-0012	21.909031	22.75
sub-0017	21.716770	20.50
sub-0019	22.436428	21.25

Fig. 10 Example showing how predictions can be accessed from the results object

Raw predictions made per participant in the validation set (s) can be accessed in multiple formats (*see* Fig. 10 for an example) and can be useful in performing further analysis beyond those implemented in the base library (e.g., computing new metrics or feature importances).

In the case that the underlying machine learning model natively supports a measure of feature importance (e.g., beta weights in a linear model), then these importances can be directly accessed (*see* Fig. 11). Additionally, feature importances can be estimated regardless of underlying pipeline through a built-in permutation-based feature importance method. When working with neuroimaging objects directly (e.g., volumetric or surface representations of the data), users can back-project feature importances into their original space.

```
fis = results.get_fis().mean()
fis
```

```
lh_G&S_cingul-Ant_thickness           0.041987
lh_G&S_cingul-Mid-Ant_thickness      -0.107472
lh_G&S_cingul-Mid-Post_thickness     -0.115151
lh_G&S_frontomargin_thickness         0.022932
lh_G&S_occipital_inf_thickness       -0.003825
```

Fig. 11 Example showing how averaged feature importances can be quickly accessed

```
p_values, null_values = results.run_permutation_test(n_perm=10,
                                                     blocks=data['sex'], within_grp=True,
                                                     plot=True)
```

Fig. 12 Example showing how a constrained permutation test can be performed, where target labels are only permuted for participants with the same sex label

```
from neurotools.plotting import plot

plot(results.get_fis().mean())
```

Fig. 13 Example code, for plotting feature importances from ROIs automatically onto a set of brain surfaces

The results of a single evaluation, regardless of cross-validation method, can be investigated further in order to ask questions around the statistical significance of results and/or the potential influence of confounds on results. One of the most powerful tools for this type of analysis is a permutation test, wherein the analysis is repeated but with the target labels shuffled. An important extension to this base method is the ability to restrain the shuffling of target labels according to an underlying group or nested group structure (*see* Fig. 12 for an example).

Another available method related to probing the significance of results, is the ability to statistically compare between two and more similar results objects, that perhaps vary on choice of a meaningful hyper-parameter. It can also be useful in some instances to visualize the predictions made in other ways, for example, through ROC plots from a binary or multi-class analysis, or plots showing the residuals from regression prediction. Feature importances from BPt are further designed to be easily visualized through the related python package, from the same maintainers as BPt, bp-neurotools (*see* Fig. 13). This package contains one-line automatic plotting functions that handle a number of different cases (e.g., plotting ROIs, brain surfaces, brain volumes, or collages of different combinations).

When working with neuroimaging data files directly (e.g., performing machine learning on surfaces), BPt includes utilities that allow the user to back-project feature importances back into the original native space. This can be useful, along with the already mentioned neuroimaging specific plotting utilities, for visualizing results.

Note
When it comes to presenting a final set of results within a manuscript or project write up, there is no one-size fits all solution. Instead, how one reports results will depend fundamentally on the question(s) of interest. In practice, the typical advice is that all metrics from experiments related to questions should be reported. Likewise, all related experimental configurations tested should also be reported, the key point being that the user should do their best to accurately and fairly present their results. As tempting or desirable as publishing a very accurate classifier may be, authors should take care not to overstate their findings. This principle holds in the context of null findings as well, where it is valuable to highlight the areas where predictive models fail.

5 General Pitfalls

There are of course general pitfalls to be aware of when performing any type of analyses on observational data, which are not specific to this library itself but are prudent to keep in mind. Perhaps the most general is that despite machine learning, deep learning, or other variations, results will typically be correlational, not causal, in nature.

While machine learning can be a useful tool for identifying null findings, for example when a model is not predictive, the nature of predictive modeling means we cannot ever be fully confident. In other words, just because one model (or full pipeline/set of steps) isn't predictive does not mean that another one may produce a positive result. In practice, it is typically sufficient in the case of null findings to show that a representative range of pipelines all fail to predict the outcome.

The over-use of cross-validation or 'double-dipping' is a particularly insidious and sometimes hard to detect issue within machine learning and the broader literature [4]. These types of mistakes are often responsible for overly optimistic or inflated accuracy. Further, the conceptual difficulties with employing cross-validation correctly can multiply in the case of nested cross-

validation, so potential users should be very careful if attempting to implement a custom-designed cross validation scheme. The cleverly titled "I tried a bunch of things: The dangers of unexpected overfitting in classification of brain data" provides a good, expanded description on this issue more broadly [5].

Be wary of results that look 'too' good. There are many different mistakes in machine learning which can lead to over-confident results, including problems with the data, mis-using cross-validation, and a whole host of other tricky issues. When encountering a situation like this, the best course of action is typically to perform small checks (building them into analysis code, whenever possible). These include little things such as printing the shape of a dataset, or maybe performing some assertion on the expected distribution of variables (e.g., confirm values for age in years are greater than 0 and less than 100). These types of checks are generally low effort and in the long run can be helpful in detecting small but disastrous bugs.

Interpreting, and in some cases overinterpreting, feature importances is a common problem. In practice, different measures of feature importance may have different drawbacks and constraints, and it is therefore a good idea to make sure one first understands a given importance's potential limitations. For example, in the case of multivariate linear models we refer readers to the excellent tutorial made available by scikit-learn. Hooker et al. [6] outline well other potential issues in interpreting some other common formulations of feature importance [6]. A more general discussion around interpretability in machine learning by Kaur et al. [7] may also be of interest [7].

6 Step-by-Step Example https://colab.research.google.com/drive/1vFGw8HtpDeLCbDmYUiLKwa5JQwsiiUnF?usp=sharing

Intro/Setup

Within this example notebook, we will investigate as our test question of interest: Can cortical thickness ROIs predict participant age?

First though, let's download the brain predictability toolbox and additional neurotools package to this collab instance.

Note running this cell this first time will install and then crash the instance, after this we can just run it as normal.

```
try:
 import BPt as bp
except ImportError:
 !pip install brain-pred-toolbox
```

(continued)

```
!pip install bp-neurotools
# We need to force a restart of the runtime/kernel
# because of the version of matplotlib we are using
exit()
```

This example is designed to be run from the already prepared quick_start file, which went through a few simple steps to combine separately saved thickness ROIs into a single file, as well as to add age and sex columns. We will download this file to this instance directly, now.

```
!wget https://raw.githubusercontent.com/sahahn/methods_-
series/master/ds002790/quick_start.csv
--2022-10-31 17:01:10--https://raw.githubusercontent.
com/sahahn/methods_series/master/ds002790/quick_start.
csv
Resolving raw.githubusercontent.com (raw.githubusercon-
tent.com)... 185.199.108.133, 185.199.109.133,
185.199.110.133, ...
Connecting to raw.githubusercontent.com (raw.githubuser-
content.com)|185.199.108.133|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 304024 (297K) [text/plain]
Saving to: 'quick_start.csv.1'

 quick_start.csv.1 0%[ ] 0 --.-KB/s quick_start.csv.1
100%[====================>] 296.90K --.-KB/s in
0.02s
2022-10-31 17:01:10 (13.8 MB/s) - 'quick_start.csv.1'
saved [304024/304024]
```

Preparing Our Dataset

As a first step, we will prepare our data into a BPt Dataset object. If you are already familiar with the python library pandas, then you might notice that this object looks awfully similar to the pandas DataFrame object – and you would be right! The Dataset class is built directly on top of the DataFrame class, just adding some extra functionality/special behavior for working with the BPt.

In this minimal example, this step will be rather simple. There are 3 different 'roles' that columns within our Dataset can take. The first is called 'data,' which by default every loaded column will be specified as, these are going to be our features that are used to predict some variable of interest (The X variable in a scikit-learn style setup). The second key role is

(continued)

'target,' which is going to be any of our feature(s) which we want to predict (using the columns as input variables). The last is 'non input' which as the name suggests are any variables which we do not want to ever use directly as an input/data variable. In this example, we will treat both age and sex as targets, and not use the 'non input' role.

Here we load data directly from a prepared csv, which gives us a BPt Dataset object. In loading the csv, we also specify a series of additional arguments, these are the file paths of the csv, which column we want to be treated as the index, then also which columns we want in different roles (where remember that by default every loaded variable is of role 'data' unless otherwise specified).

import BPt as bp

```
# Load
data = bp.read_csv('quick_start.csv',
                   index_col='participant_id',
                   targets=['age', 'sex'])


# Set sex as a binary variable
data = data.to_binary('sex')
```

Show the first five rows of our Dataset

```
data.head()
 lh_G&S_frontomargin_thickness  lh_G&S_occipital_inf_th-
ickness \
participant_id
sub-0001     1.925    2.517
sub-0002     2.405    2.340
sub-0003     2.477    2.041
sub-0004     2.179    2.137
sub-0005     2.483    2.438


 lh_G&S_paracentral_thickness  lh_G&S_subcentral_thick-
ness \
participant_id
sub-0001     2.266   2.636
sub-0002     2.400   2.849
sub-0003     2.255   2.648
sub-0004     2.366   2.885
sub-0005     2.219   2.832
```

(continued)

```
 lh_G&S_transv_frontopol_thickness \
participant_id
sub-0001 2.600
sub-0002 2.724
sub-0003 2.616
sub-0004 2.736
sub-0005 2.686


  lh_G&S_cingul-Ant_thickness  lh_G&S_cingul-Mid-An-
t_thickness \
participant_id
sub-0001 2.777 2.606
sub-0002 2.888 2.658
sub-0003 2.855 2.924
sub-0004 2.968 2.576
sub-0005 3.397 2.985


 lh_G&S_cingul-Mid-Post_thickness \
participant_id
sub-0001 2.736
sub-0002 2.493
sub-0003 2.632
sub-0004 2.593
sub-0005 2.585


 lh_G_cingul-Post-dorsal_thickness \
participant_id
sub-0001 2.956
sub-0002 3.202
sub-0003 2.984
sub-0004 3.211
sub-0005 3.028


 lh_G_cingul-Post-ventral_thickness ... \
participant_id ...
sub-0001 2.925 ...
sub-0002 2.868 ...
sub-0003 2.972 ...
sub-0004 2.428 ...
sub-0005 3.361 ...
```

(continued)

```
 rh_S_postcentral_thickness \
participant_id
sub-0001 2.038
sub-0002 1.882
sub-0003 2.066
sub-0004 1.930
sub-0005 1.938


 rh_S_precentral-inf-part_thickness \
participant_id
sub-0001 2.425
sub-0002 2.513
sub-0003 2.410
sub-0004 2.241
sub-0005 2.445


  rh_S_precentral-sup-part_thickness  rh_S_suborbi-
tal_thickness \
participant_id
sub-0001 2.324 2.273
sub-0002 2.429 2.664
sub-0003 2.579 3.494
sub-0004 2.296 3.092
sub-0005 2.218 3.712


 rh_S_subparietal_thickness rh_S_temporal_inf_thickness \
participant_id
sub-0001 2.588 2.548
sub-0002 2.676 2.220
sub-0003 2.375 2.625
sub-0004 2.641 2.622
sub-0005 2.360 2.402


 rh_S_temporal_sup_thickness \
participant_id
sub-0001 2.465
sub-0002 2.291
sub-0003 2.497
sub-0004 2.487
sub-0005 2.442
```

(continued)

```
 rh_S_temporal_transverse_thickness age sex
participant_id
sub-0001 2.675 25.50 1
sub-0002 2.714 23.25 0
sub-0003 2.674 25.00 0
sub-0004 2.556 20.00 0
sub-0005 1.864 24.75 1
```

[5 rows × 150 columns]

There are other steps we could potentially perform here as well, e.g., let's look to see if there are any extreme outliers. Like you can find other available options for different common encodings or filtering.

```
data = data.filter_outliers_by_std(scope='float',
n_std=10)
data.shape
(224, 150)
```

We can see that no data was dropped with a strict filter of 10 standard deviations. We can also confirm before moving on that there is no missing data in this prepared dataset.

If there were any missing data, this would print something

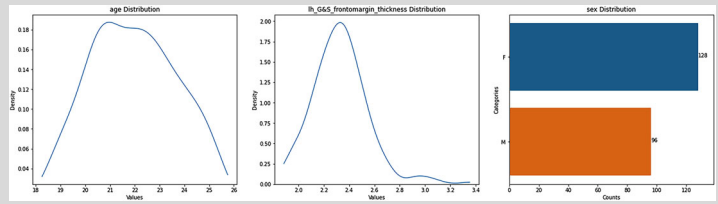
data.nan_info()

Visualize Features

BPt includes a few different utilities for easy visualizations, in particular, our Dataset class has some built in plotting functions. In the example below, we specify what variables/columns we want to plot with a special scope argument. We can see the distributions of variables with plotting methods plot and plots.

Show the targets and a random ROI

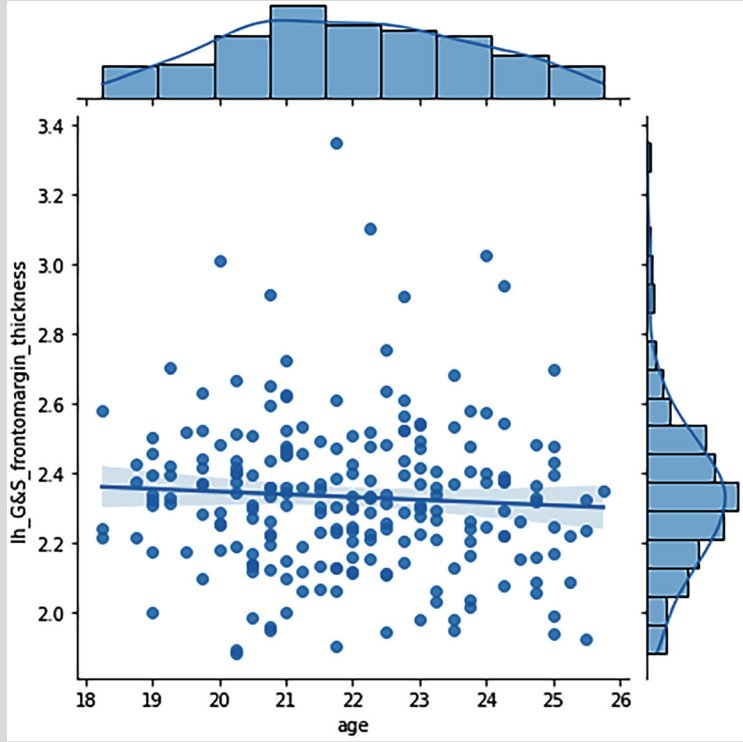
```
ex_roi = list(data)[0]
data.plots(scope=['target', ex_roi])
```



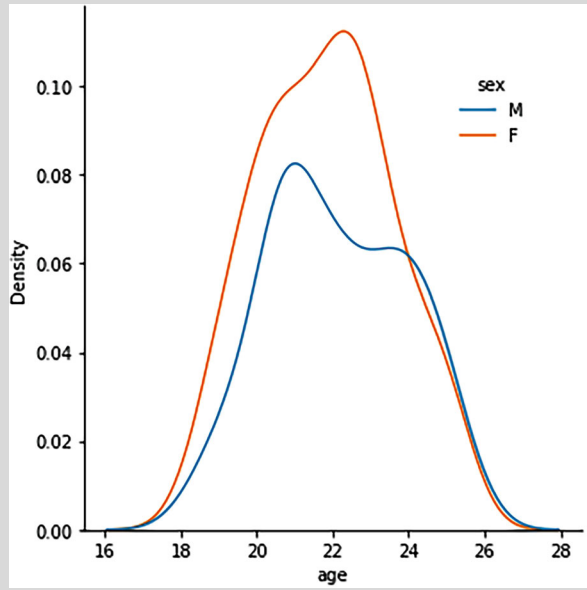
(continued)

Another built in plotting option we can use is for visualizing bi-variate relationships between variables, with plot_bivar. Note that internally these plotting functions make use the library seaborn which is addition to matplotlib for creating more complex out of the box plots.

```
# Plot age vs sex and the roi
data.plot_bivar(scope1='age', scope2=['sex', ex_roi])
```



(continued)



Machine Learning

Next, we will jump in directly to a minimal machine learning example. In this case, our question of interest is can our thickness ROIs predict age, which we have already setup within our dataset by virtue of specifying all cortical thickness ROI's as role='data' and age with role='target,' though since we have two loaded targets, we could also optionally make sure that age is predicted by passing target='age.'

To run the experiment itself, we are going to use the evaluate method from BPt. This method allows us to perform a number of different ML evaluations and can be customized according to a large number of different parameters. In this case though, we are going to provide the bare minimum input needed, and let the default settings take care of everything else.

In particular, we specify that we want to use a default ML pipeline from BPt called 'ridge_pipe' which is a pre-defined pipeline based on a regularized ridge regressor w/nested hyper-parameter search, and also we let the function know our dataset.

```
results = bp.evaluate(pipeline='ridge_pipe', dataset=da-
ta, target='age')
Predicting target = age
Using problem_type = regression
```

(continued)

```
Using scope = all (defining a total of 148 features).
Evaluating 224 total data points.
{"version_major":2,"version_minor":0,"model_i-
d":"66a13cc8677948f583ebc060aec166c4"}


Training Set: (179, 148)
Validation Set: (45, 148)
Fit fold in 1.8 seconds.
r2: 0.1416
neg_mean_squared_error: -3.27


Training Set: (179, 148)
Validation Set: (45, 148)
Fit fold in 1.8 seconds.
r2: 0.1444
neg_mean_squared_error: -2.19


Training Set: (179, 148)
Validation Set: (45, 148)
Fit fold in 1.7 seconds.
r2: 0.1131
neg_mean_squared_error: -3.44


Training Set: (179, 148)
Validation Set: (45, 148)
Fit fold in 1.8 seconds.
r2: 0.0944
neg_mean_squared_error: -2.66


Training Set: (180, 148)
Validation Set: (44, 148)
Fit fold in 5.6 seconds.
r2: 0.0198
neg_mean_squared_error: -2.59
```

We can see from the verbose output above that five different training and validation sets was evaluated. This is because the default cross-validation behavior is to run a K-Fold cross validation with 5 folds.

(continued)

In essence that brief example was designed to be as implicit as possible or mostly rely on default values. That said, we could run the same exact evaluation, but this time explicitly providing a number of the default arguments as:

```
bp.evaluate(pipeline='ridge_pipe', dataset=data
 target='age', scorer=['r2', 'neg_mean_squared_error'],
 scope='all', subjects='all',
 problem_type='regression',
 n_jobs=1, random_state=1, cv=5,
 progress_bar=True, eval_verbose=1)
```

Next, let's look at the returned results object, an instance of EvalResults which we saved in variable results.

results

EvalResults

---

r2: 0.1027 ± 0.0454

neg_mean_squared_error: -2.83 ± 0.4594

Saved Attributes: ['estimators', 'preds', 'timing', 'estimator', 'train_subjects', 'val_subjects', 'feat_names', 'ps', 'mean_scores', 'std_scores', 'weighted_mean_scores', 'scores', 'fis_', 'coef_', 'cv']

Available Methods: ['to_pickle', 'compare', 'get_X_transform_df', 'get_inverse_fis', 'run_permutation_test', 'get_preds_dfs', 'subset_by', 'get_fis', 'get_coef_', 'permutation_importance']

Evaluated With:

target: age

problem_type: regression

scope: all

subjects: all

random_state: 1

This object saves by default a large amount of potentially useful information from the experiment. This includes the mean evaluation metrics, actual estimator objects, predictions made, information on feature importance, and more.

For example, we can look at the more 'raw' object of what exactly we just ran.

results.estimator

```
BPtPipeline(steps=[('mean float',
 ScopeTransformer(estimator=SimpleImputer(), inds=Ellip-
sis)),
 ('median category',
```

(continued)

```
 ScopeTransformer(estimator=SimpleImputer(strategy='me-
dian'), inds=[])),
 ('robust float',
 ScopeTransformer(estimator=RobustScaler(quantile_range=
(5, 95)), inds=Ellipsis)),
 ('one hot encoder category',
 BPtTransformer(estimator=OneHotEncoder(handle_unknow...
```

BPtModel(estimator=NevergradSearchCV(estimator=–Ridge(max_iter=100, random_state=1, solver='lsqr'), param_distributions={'alpha': Log(lower=0.001, upper=100000.0)}, ps={'cv': BPtCV(cv_strategy=CVStrategy(), n_repeats=1, splits=3, splits_vals=None), 'cv__cv_strategy': CVStrategy(), 'cv__cv_strategy__groups': None, 'cv__cv_strategy__stratify': None, 'cv__cv_strategy__train_only_subjects': None, 'cv__n_repeats': 1, 'cv__only_fold': None, 'cv__random_state': 'context', 'cv__splits': 3, 'dask_ip': None, 'memmap_X': False, 'mp_context': 'loky', 'n_iter': 60, 'n_jobs': 1, 'progress_loc': None, 'random_state': 1, 'scorer': make_scorer(r2_score), 'search_only_params': {}, 'search_type': 'RandomSearch', 'verbose': 0, 'weight_scorer': False}, random_state=1), inds=Ellipsis))])

Yikes, that's a handful... another way of looking at the pipeline we ran is to look at it in the BPt pipeline object syntax rather than the raw scikit-learn style.

Essentially when we pass 'ridge_pipe' as our pipeline, it will grab some default code from BPt.default.pipelines, we can also import it directly.

from BPt.default.pipelines import ridge_pipe

```
ridge_pipe
Pipeline(steps=[Imputer(obj='mean', scope='float'),
 Imputer(obj='median', scope='category'), Scaler(obj='-
robust'),
 Transformer(obj='one hot encoder', scope='category'),
 Model(obj='ridge',
  param_search=ParamSearch(cv=CV(cv_strategy=CVStrategy
()),
 n_iter=60),
 params=1)])
```

As a careful user might note that in this case a number of these steps are actually redundant, e.g., we have no missing data, so no need for imputation and we have no categorical data, so no need for one hot encoding. The beauty here is that

(continued)

if not needed, or if out of scope given a certain input, these pipeline steps are just skipped. This is helpful for designing re-usable pipelines, that are robust to different types of inputs (e.g., includes categorical variables or not).

We can also look at some other options available when working with a result's object, for example, viewing the raw predictions.

```
results.get_preds_dfs()[0]
 predict y_true
participant_id
sub-0001 21.909845 25.50
sub-0005 22.126259 24.75
sub-0012 21.909031 22.75
sub-0017 21.716770 20.50
sub-0019 22.436428 21.25
sub-0020 22.145191 20.00
sub-0029 22.262978 21.75
sub-0030 21.805216 20.50
sub-0032 22.008379 20.50
sub-0034 22.045389 24.75
sub-0035 22.154486 23.25
sub-0036 22.001040 24.00
sub-0039 22.097433 22.25
sub-0040 21.570295 19.75
sub-0045 22.572697 25.00
sub-0052 21.528109 19.00
sub-0059 21.977768 23.25
sub-0063 22.722687 25.25
sub-0068 21.829220 22.25
sub-0070 22.035213 18.75
sub-0074 21.457249 20.25
sub-0079 22.089703 20.25
sub-0085 21.702505 23.00
sub-0086 21.735174 21.00
sub-0092 22.166224 21.50
sub-0100 22.488585 23.75
sub-0103 21.614632 21.50
sub-0107 21.745026 19.25
sub-0108 21.288305 21.25
sub-0120 22.015606 22.75
sub-0121 21.495790 21.75
sub-0125 22.300081 23.00
sub-0152 22.030699 23.75
sub-0153 21.711060 19.00
sub-0156 21.993868 20.75
```

(continued)

```
sub-0161 21.311602 23.50
sub-0163 22.200386 22.25
sub-0167 21.918657 24.75
sub-0169 21.757229 25.00
sub-0188 22.429842 23.75
sub-0189 21.426121 20.50
sub-0192 21.896299 22.25
sub-0196 21.228672 19.25
sub-0198 22.514311 25.00
sub-0204 21.584047 19.75
```

Visualizing Results

Let's say we want to look at feature importances. Because the model we ran was a regularized ridge regression, the importances we are going to be looking at are the beta weights from the model. Further, because we ran a 5-fold CV, we want to look at the beta weights as averaged across each of our 5 models:

```
fis = results.get_fis().mean()
fis
lh_G&S_cingul-Ant_thickness 0.041987
lh_G&S_cingul-Mid-Ant_thickness -0.107472
lh_G&S_cingul-Mid-Post_thickness -0.115151
lh_G&S_frontomargin_thickness 0.022932
lh_G&S_occipital_inf_thickness -0.003825
 ...
rh_S_suborbital_thickness -0.033135
rh_S_subparietal_thickness -0.026897
rh_S_temporal_inf_thickness 0.008185
rh_S_temporal_sup_thickness -0.004152
rh_S_temporal_transverse_thickness -0.080884
```

Length: 148, dtype: float32

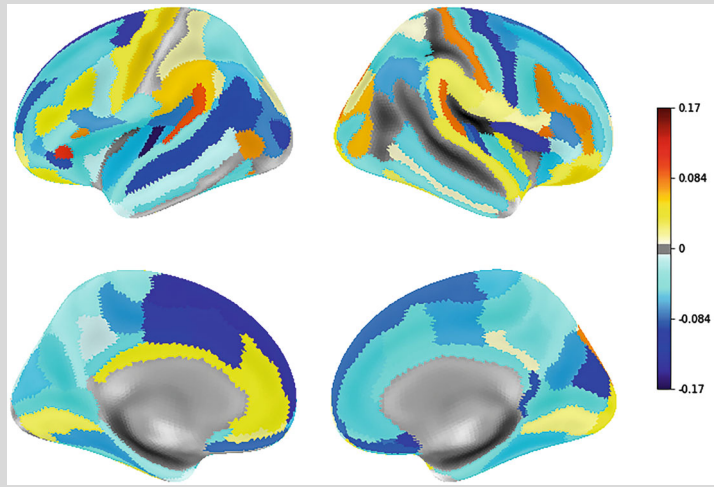
Next, let's plot our results. For this, we will use an 'automagical' plotting function, plot, from the library neurotools (a library designed to complement BPt, but with a less ML focus).

from neurotools.plotting import plot

```
plot(fis)
Downloading latest neurotools_data to/root/neurotools_da-
ta
Downloaded data version = 1.2.5 complete!
Current version saved at: /root/neurotools_data/neuro-
tools_data-1.2.5/data
```

(continued)

```
If you move this directory, make sure to update saved
location in data ref at /usr/local/lib/python3.7/dist-
packages/neurotools/data_ref.txt.
```



This plotting function tries to basically automate everything. This includes, as an important caveat, an automatic conversion from ROI names to their associated parcellation – which for now only supports a small number of underlying parcellations, including of course our current freesurfer based destr. parcellation.

Next, we will plot the same feature importances again, but this time go a little further to add some customization. We will customize here by adding a user-defined threshold in which to not show results under and also go a little deeper and add it as a part of a collage of plots. We will also save the figure with matplotlib.

import matplotlib.pyplot as plt
from neurotools.plotting import plot_bars

```
# Initialize two subplots on the same row
fig, axes = plt.subplots(nrows=1, ncols=2,
 figsize=(18, 6),
 gridspec_kw={'wspace': 0})
```

# Share threshold
threshold = .1
# Make a bar plot, with fi values before each fold
plot_bars(results.get_fis(), threshold=threshold, ax=axes[0])

(continued)

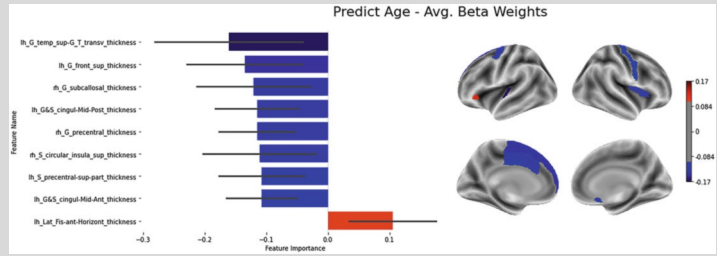
```
# Use the same plot as before, but with some extra arguments

plot(fis,
 threshold=.1,
 space='fsaverage5', # Note could change to fsaverage for higher resolution
 ax=axes[1]
 )

# Add a title to the whole figure

plt.suptitle('Predict Age - Avg. Beta Weights', fontsize=22)


plt.savefig('example.png', dpi=100)
```



Permutation Test

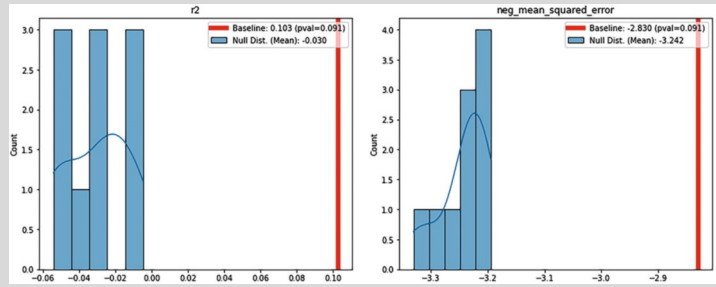
There are of course other useful things we can do with this result's object. One useful one is to be able to easily run permutation tests as a way of estimating the significance of our results. In the context of results generated from cross-validation and if significance tests are desired, than permutation based methods are preferred.

While this is useful, we can also easily extend this idea with a powerful extension, that is, constraining the permutations in a meaningful way. For example, we will run 10 permutations, but with the added specification that values within the target only be allowed to be swapped with other participants of the same sex. Now what we are testing is a more specific null model, one where any potential sex-age effects will be preserved within our null distribution. In this version, we are essentially testing to see if sex effects are driving our observed R2. If the null dist mean is still the same as before, it is likely not, but if it is higher, than to some degree it might be. This

(continued)

type of constrained permutation test is especially useful with multi-site data, where a variable representing site is passed.

```
p_values, null_values = results.run_permutation_test
(n_perm=10,
 blocks=data['sex'], within_grp=True,
 plot=True)
```



Other ML Models

There are plenty of other choices for ML models we could have used besides the regularized linear ridge regression, even while staying within the default pipelines made available by BPt. That said . . .

Testing a bunch of different models is a common area where essentially too many researcher degrees of freedom are often introduced. In order for the results from a ML-based predictive experiment to be valid, some sort of cross or external validation is essential. In this example, we have used a 5-fold cross-validation on the entire available dataset. What that means in practice is that the only ML experiments we want to run this full 5-fold cross validation on, are those which are directly related to our main question of interest – or in other words – we should be reporting any results which we test using this full 5-fold cross validation.

So let's say we do test multiple models using this full 5-fold CV. Let's test additionally the default elastic-net based pipeline and the default gradient boosting based pipeline. To do this, we will use an object from BPt called Compare which allows the same evaluate function from earlier to perform a comparison between a few different options.

```
compare_pipes = bp.Compare(['ridge_pipe', 'elastic_-
pipe', 'gb_pipe'])
```

(continued)

```
all_results = bp.evaluate(pipeline=compare_pipes,
 dataset=data, eval_verbose=-2, n_jobs=1)


all_results.summary()
```

In this case, now that we have run and tested three different models, we have a perfect example of what not to do, which is, just reporting the ridge regression results. Or more broadly speaking, just reporting the best performing results. Instead, if we were reporting this result in a paper or write-up, we NEED to include information related to all of the experiments we ran, if those experiments made use of our main validation strategy! Not reporting this information is similar to p-hacking.

Let's say though we don't want to have choice of ML model fill up our results, but we still want to find a high performing model. One common strategy for this is done with nested cross-validation, the easiest being a front-end global train-test split. I'll show a quick example of that below (again under the assumption that this was done INSTEAD of what we already did in this notebook).

```
# Make a copy of our dataset
data_tr_test = data.copy()


# Split it into train and test sets
data_tr_test = data_tr_test.set_test_split(size=.2, ran-
dom_state=4)
data_tr_test
```

Re-run the same compare style analysis from before, but now with subjects = 'train.' So, explicitly, in this alternate analysis we are performing 5-fold CV, but only on the subset of subjects we labelled as part of the training set.

```
all_results = bp.evaluate(compare_pipes, dataset=da-
ta_tr_test,
 subjects='train', eval_verbose=-2)
all_results.summary()
```

Then, the next step of this work-flow would be once the correct pipeline is identified, our new validation test is training on the full training set, and testing on the set of subjects that

(continued)

we said was the test set. And we do it with just our best identified 'ridge pipe.'

```
results = bp.evaluate('ridge_pipe', dataset=data_tr_t-
est, subjects='all', cv='test')
results
```

Now, we could report just these ridge regression results, with our checks between different pipelines conducted safely on a training set. That said, in this particular example – given the small sample sizes involved, a train-test approach, where our test set only has 45 subjects is likely a pretty bad idea... or rather, we should expect large error bars. We can formalize this intuition by returning to working with the full dataset and essentially as our CV strategy is simulating a repeated train and test split of 20%.

This is a custom CV object that sats, use a random test split of 20% 10 times.

cv = bp.CV(splits=.2, n_repeats=10)

results = bp.evaluate('ridge_pipe', dataset=data, cv=cv, eval_verbose=0)

Plot the distribution of scores with library seaborn

import seaborn as sns

sns.displot(results.scores['r2'], kde=True)

Seriously don't double dip

I want to make sure this point is very clear, so next we will look at an ever more extreme example than just trying a few models and just reporting the best one. In this case we will be repeating a 3-fold CV, but with a bunch of different random splits.

for random_state in range(10):

```
 score = bp.evaluate('ridge_pipe', data, cv=3, random_-
state=random_state,
 eval_verbose=0, progress_bar=False).score
 print(random_state, score)
```

First thing to notice is that cross-validation, especially with small-ish sample sizes is not perfect. Depending on just luck of different random split, we can observe some variability in mean explained variance.

The second, is that a dishonest person could use a strategy like the one above and run and just report the following:

bp.evaluate('ridge_pipe', data, cv=3, random_state=0,

```
 eval_verbose=0, progress_bar=False)
```

(continued)

The 'cheating' part here, is that we essentially tried 10 different things using our full dataset, then chose a result from those 10, didn't report the others. That is to say, running 10 repeats of different 3-fold CV is actually not a bad thing – as long as you report all of them (well, the average).

Well, this example is obvious, it can be easy to accidently end up doing something similar. Say for example you try one model, and then another, and then with slightly different features, then maybe you go back to a different model, etc...

Predict Sex

The other thing to note is that all of the BPt style objects when possible are generic to problem type also. So that means, even if we switch to a binary prediction, we can still use the same code from the ridge pipe. Let's try that now:

from BPt.default.pipelines import ridge_pipe

We can pass either the default str, or the object itself for pipeline

Also note that now that we have two targets, we need to specify which one we want to predict

Let's also add one more parameter, n_jobs, which let's use multi-process our evaluation

```
results = bp.evaluate(pipeline=ridge_pipe,
 dataset=data,
 target='sex',
 n_jobs=2)
results
```

Now if we look at the fully composed scikit-learn style estimator again:

results.estimator

We see that there are some changes from before, e.g., now our base model is a LogisticRegression and the set of parameters it searches over are different

```
param_distributions={'C': Log(lower=1e-05,
upper=1000.0), 'class_weight': TransitionChoice([None,
'balanced'])}
```

Where before the default hyper-parameter search parameters were:

```
param_distributions={'alpha': Log(lower=0.001,
upper=100000.0)}
```

(continued)

On the end of the user, specifying these hyper-parameter distributions is done when building the model as just: Model (obj='ridge,' params=1, ...)

Where 1 refers which default distribution to select (see all choices for all supported models here: https://sahahn.github.io/BPt/options/pipeline_options/models.html).

A more advanced used could also manually specify this choice as well, for example:

```
# Make a copy of the default ridge pipeline
our_ridge_pipe = ridge_pipe.copy()


# Look in our pipeline at where the param distribution is
saved
our_ridge_pipe.steps[-1].params
# Replace it with one of our choosing
our_ridge_pipe.steps[-1].params = {'C': bp.p.Log(low-
er=1, upper=1000)}


# Then re-run
results = bp.evaluate(pipeline=our_ridge_pipe,
 dataset=data,
 target='sex',
 n_jobs=2)
results
```

Something to keep in mind of course when performing custom hyper-parameter tuning like this is not to abuse it, re-running different choices until by chance you get a high performing results. This of course is another area like with what we discussed before, and is why a lot of time in practice it can be helpful to just use default settings, either defaults from BPt, or defaults from other libraries as we will see in the next example:

We can also just as easily use custom sklearn objects as either a step in the pipeline – or instead of the pipeline.

```
from sklearn.linear_model import LogisticRegressionCV
# We just need to wrap it in a Model object, so BPt
# knows how to handle it correctly
sk_model = bp.Model(LogisticRegressionCV())

results = bp.evaluate(pipeline=sk_model,
 dataset=data,
 target='sex',
```

(continued)

```
 mute_warnings=True # Mute ConvergenceWarning's
 )
results
```

Conclusion

BPt as a library is still a work in progress, and may not be as polished as some other available libraries, but I hope it still may be useful for some people.

If you are interested, I encourage you to check out the User Guide on the documentation page, which includes a number of other Full Examples. Likewise, the github repository method_series contains a work in progress larger tutorial, which will feature the use of BPt and neurotools in analyzing the three AOIMIC datasets, across a number of different analysis.

I also encourage anyone interested in contributing code, ideas, bugs, etc... to check out the project github and/or open an issue with your question or comment.

7 Sustainability

The software is currently hosted on github at https://github.com/sahahn/BPt as well as through the python PIP repository under 'brain-pred-toolbox.' Users are welcome to submit any code/improvements to the library. Users are also welcome to comment with any suggestions or features they would like to see implemented.

8 Conclusions

Frameworks like BPt can be helpful for some projects, but require a tradeoff between flexibility to usefulness. That said, we hope this library can be as useful a tool as possible moving forward and welcome any suggestions, feedback, or bug reports on the library's github page (https://github.com/sahahn/BPt).

References

1. Hahn S, Owens MM, Yuan D, Juliano AC, Potter A, Garavan H, Allgaier N (2022) Performance scaling for structural MRI surface parcellations: a machine learning analysis in the ABCD Study. Cereb Cortex 33:176–194. https://doi.org/10.1093/cercor/bhac060
2. Dadi K, Rahim M, Abraham A, Chyzhyk D, Milham M, Thirion B, Varoquaux G,

Alzheimer's Disease Neuroimaging Initiative (2019) Benchmarking functional connectome-based predictive models for resting-state fMRI. NeuroImage 192:115–134. https://doi.org/10.1016/j.neuroimage.2019.02.062

3. Varoquaux G (2018) Cross-validation failure: small sample sizes lead to large error bars. NeuroImage 180:68–77. https://doi.org/10.1016/j.neuroimage.2017.06.061
4. Button KS (2019) Double-dipping revisited. Nat Neurosci 22:688–690. https://doi.org/10.1038/s41593-019-0398-z
5. Hosseini M, Powell M, Collins J, Callahan-Flintoft C, Jones W, Bowman H, Wyble B (2020) I tried a bunch of things: the dangers of unexpected overfitting in classification of brain data. Neurosci Biobehav Rev 119:456–467. https://doi.org/10.1016/j.neubiorev.2020.09.036
6. Hooker G, Mentch L, Zhou S (2021) Unrestricted permutation forces extrapolation: variable importance requires at least one more model, or there is no free variable importance. Stat Comput 31:82. https://doi.org/10.1007/s11222-021-10057-z
7. Kaur H, Nori H, Jenkins S, Caruana R, Wallach H, Wortman Vaughan J (2020) Interpreting interpretability: understanding data Scientists' use of interpretability tools for machine learning. In: Proceedings of the 2020 CHI conference on human factors in computing systems. Association for Computing Machinery, New York, pp 1–14